

Data Warehousing for Biomedical Informatics

Data Warehousing for Biomedical Informatics

Richard E. Biehl



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2016 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20151006

International Standard Book Number-13: 978-1-4822-1522-9 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

To Janet

Contents

Preface.....	xvii
Author	xxi

1 Biomedical Data Warehousing.....	1
Nature of Biomedical Data	1
Nature of Warehoused Data	4
Business Requirements	7
Functional Requirements	8
Data Queries, Reports, and Marts	9
Data Issues and Hypotheses.....	10
Performance and Control Data	11
Never-Finished Warehouse	11
Organizational Readiness	12
Implementation Strategy	13
Warehouse Project.....	14

SECTION I ALPHA VERSION

2 Dimensional Data Modeling	17
Evolution of Data Warehouses	17
Relational Normalization.....	18
Dimensional Design.....	19
The Star Schema	22
Dimensions and Subdimensions	24
Dimensional Granularity.....	26
Transposing Dimensional Schema	29
Anticipating Dimensions.....	32
Affinity Analysis	33
Dimensions as Supertypes.....	34
Reverse Engineering.....	37
3 Understanding Source Data	39
Implicit versus Explicit Data	40
Semantic Layers.....	42
BFO Continuants.....	44

BFO Occurrents	47
BFO Temporal Regions	48
BFO Spatiotemporal Regions	50
BFO Processual Entities	51
BFO Process Aggregate	52
BFO Process Boundary	52
BFO Processual Context	52
Information Artifacts	53
IAO Material Information Bearer	54
IAO Information Content Entity	55
IAO Information Carrier	56
Biomedical Context	57
OBI Processes	58
OBI Process Aggregates	61
OBI Processual Contexts	63
Clinical Picture	64
Ontological Levels	67
Epistemological Levels	68
Observations	68
Hypotheses	70
Conclusions	71
4 Biomedical Warehouse.....	75
Biomedical Star	76
Biomedical Facts	78
Master Dimensions	79
Organization Dimension	80
Study Dimension	81
Caregiver Dimension	82
Interaction Dimension	83
Subject Dimension	84
Object Instantiation	85
Reference Dimensions	86
Diagnosis Dimension	86
Pathology Dimension	87
Procedure Dimension	90
Treatment Dimension	91
Material Dimension	92
Facility Dimension	93
Accounting Dimension	94
-Omics Dimension	95
Almanac Dimensions	96
Geopolitics Dimension	96
Calendar Dimension	99

Clock Dimension.....	100
Environment Dimension.....	102
Structure Dimension	102
System Dimension.....	103
Unit of Measure Dimension.....	103
Analysis Dimensions.....	105
Annotation Dimension.....	105
Quality Dimension	106
Control Dimensions	106
Metadata Dimension	107
Data Feed Dimension	109
Data State Dimension.....	109
Operation Dimension.....	109
Requirements Alignment	109
5 Star Dimension Design Pattern	117
Structure of a Dimension.....	117
Master Data: Definition Tables	118
Slowly Changing Dimensions	121
SCD Example.....	125
Source Keys: Context and Reference Tables	129
Fact Participation: Group and Bridge Tables	134
Interconnections: Hierarchy Tables	137
Natural Hierarchies	140
Connecting to Facts	142
Dimension Navigation.....	143
6 Loading Alpha Version	149
Throw-Away Code	149
Selecting and Preparing Sources	150
Generating Surrogate Keys	156
Simple Dimensions and Facts.....	159
First Source: Patient Master Data	159
Alternative G–B–H Processing	162
Second Source: Patient Address Facts	164
Quick Fact Queries	170
Recap of Simple ETLs	172
Complicated Dimensions and Facts	175
Third Source: Basic Lab Results.....	175
Finalizing Alpha Structures.....	188
V&V of Alpha Version.....	189
Metadata-Fact Counts	189
Dimension–Subdimension Counts.....	190
Dimension-Fact Counts	190
Recreating Sources	191

SECTION II BETA VERSION

7 Completing the Design	197
Unit of Measure.....	198
UOM Scale.....	199
UOM Class.....	199
UOM Unit	200
UOM Measure	201
UOM Language	201
UOM Value	202
Metadata Mappings	208
Metadata Contexts.....	208
Metadata Reference	209
Metadata Definition.....	209
Targeting Metadata	211
Property Metadata	212
Implicit UOM Metadata	213
Superseding Metadata	213
Codeset Translation Metadata	214
Flat Hierarchy Metadata	215
Metadata Examples	215
First Alpha Source: Patient Master Data	216
Second Alpha Source: Patient Address Facts.....	218
Third Alpha Source: Basic Lab Results.....	222
Control Dimensions	224
Data State	224
Operation	225
Data Feed	226
Reinitializing the Warehouse	226
Empty Warehouse	226
System Rows.....	227
Data States	228
Units of Measure	229
Configuration Data.....	229
Context Entries	230
Default Dimensionality.....	231
Metadata Loading	232
8 Data Sourcing	233
Source Mapping Challenges	233
Coverage and Seamlessness.....	234
Functional Normalization.....	238
Qualities of Facts.....	243
Lab Result Facts	244

Allergy Assessment Facts	247
Financial Charge Facts	250
Fact Superseding	254
Dimensionalizing Facts	257
Sourcing Your Data	259
Selecting Columns.....	260
Selecting Rows.....	263
Selecting Time.....	264
9 Generalizing ETL Workflows	267
Standardizing Source Data.....	267
Dataset Controls	268
Transaction Controls	272
Source Data Values	275
Source Data Intake Jobs	275
SDI Design Pattern.....	277
Source Data Consolidation	282
External versus Internal Sourcing	283
Single Point of Function	284
Single Point of Failure	286
ETL “Pipes”.....	287
Reference Pipe.....	288
Definition Pipe	289
Fact Pipe	290
Checkpoint, Restart, and Bulk Loading.....	290
Metadata Transformation	291
Codeset Translations	292
General versus Functional Transformations.....	294
Resolving “~All~” Entries.....	296
Early versus Late Binding.....	297
Data Control Pipe.....	299
ETL Job	300
Data Control Layers.....	300
Data Control Points	304
Assigning Master IDs to ETL Layers	305
Wide versus Deep Data	306
10 ETL Reference Pipe.....	309
Metadata Transformation	312
Codeset Translation.....	314
Reference Processing.....	316
Reference Composite	317
Reference Staging	319
Beta Limitations.....	320

Resolve References	321
Unknown or Broken Keys	323
Alias Entry Collisions	324
Unresolved References	327
Distinct Composites	328
Surrogate Assignments	329
Transactional Redistribution	330
Alias Propagation	331
Reference Entries	332
New References	332
Alias Entries	332
Bridges and Groups	334
Hierarchy Entries	334
New Self-Hierarchies	335
Fiat Hierarchies	335
Natural Hierarchies	336
Terminus Resolution	339
Fiat Hierarchy Cascade	340
11 ETL Definition Pipe	343
Processing Complexities	343
Metadata Transformation	347
Deep and Wide Staging	348
Example Master Loads	352
Insert New Definitions	356
New Orphans	359
Orphan Auto-Adoption	360
Definition Change Processing	361
Slowly Changing Dimensions	361
Multiple Simultaneous Transactions	364
Building SCD Transaction Sets	367
Staging Existing Definitions	367
Assigning Deep Row Numbers	368
Distribute Deep Non-SCD Updates	369
Assigning Relative Wide Row Numbers	370
Applying Transactions to Dimensions	372
Obtain New Dimension IDs	373
Auto-Adopt Orphan Definitions	374
Insert New Definitions	374
Update Existing Definitions	375
Performance Concerns	375

12 ETL Fact Pipe.....	377
Metadata Transformation	378
Generating Sourced Facts	378
Generating Factless Facts.....	380
Staged Facts.....	382
Bridges and Groups	383
Bridge Staging	384
Group Staging	386
Group Lookups	387
New Group Surrogates.....	389
Distribute Surrogates to Group IDs	390
Distribute Group IDs to Bridges.....	390
Insert New Groups.....	390
Insert New Bridges	391
Build Facts	391
Bridge Pivot	392
Value Alignment.....	394
Finalize Dimensions.....	394
Unit of Measure.....	395
Implicit UOM	396
Assigning the Implicit UOM.....	397
Calendar and Clock.....	398
Organization.....	399
Optional Dimensions	400
Set Control Dimensions	400
Data State	400
Datafeed	401
Insert Fact Values	401
Superseding Facts.....	402
Metadata Review	403
13 Finalizing Beta.....	407
Audit Trail Facts	407
Datafeed Dimension	410
Verification and Validation.....	411
Structural Verification.....	412
Context Tables	412
Reference Tables.....	413
Definition Tables	415
Bridge Tables	416
Group Tables	416

Fact Table.....	417
Metadata	418
Preparing for Gamma	419
SECTION III GAMMA VERSION	
14 Finalizing ETL Workflows.....	423
Alternatively Sourced Keys	425
Sourcing Compound Natural Keys	425
Sourcing Warehouse Surrogates	427
Alternatives in the Reference Pipe.....	428
Sourced Metadata.....	431
Standard Data Editing.....	432
Value Trimming and Cleanup.....	433
Timestamp Handling.....	433
Data Type Checking.....	433
Range Checking	436
Value-Level UOM	437
Undetermined Dimensionality.....	441
ETL Transactions.....	445
Target States	446
Superseded Facts.....	447
Continuous Functional Evolution	450
15 Establishing Data Controls	451
Finalizing Warehouse Design	451
Database Statistics	451
Application Layer Issues.....	452
Indexing and Partitioning	453
Outrigger Tables.....	454
Fact Tables	457
Redaction Control Settings.....	462
Data Monitoring	465
Pipeline Counts	466
System Rows.....	466
Unexpected and Undesired Values.....	467
Orphan Tracking	469
Surrogate Merges.....	473
Security Controls.....	475
Implementing Dataset Controls	476
Warehouse Support Team.....	479
16 Building out the Data	481
Minimize Data Seams	481
Shifting toward Metrics	486
Metrics Process Design	487

New Dimension and Fact Tables.....	488
Metric Fact Table	490
Control Fact Table	491
Metric Subdimension.....	492
Control Subdimension.....	495
Display Subdimension.....	496
Populating Metric Values	496
Populating Control Values	498
Populating Displays.....	500
Metric Aggregation	502
Metric–Control Fact Views.....	507
17 Delivering Data.....	511
Warehousing Use Cases.....	511
User Analysts.....	512
HIPAA Controller.....	512
Data Owners	513
Data Governance.....	514
Institutional Review Board.....	515
Support Teams.....	515
Data Administration	516
Auditor	516
Data Sources.....	517
Standards Bodies.....	517
Privacy-Oriented Usage Profiles	518
Metadata Browsing.....	520
Cohort Identification	524
Fact Count Queries	528
Timeline Generation	529
Business Intelligence.....	539
Alternative Data Views	541
Flattened Groups	541
Flattened Facts.....	547
External Data Marts	548
i2b2	548
Provider Dimension	550
Patient Dimension	553
Visit Dimension	553
Concept Dimension.....	557
Fact Table.....	560
18 Finalizing Gamma.....	565
Business Requirements	565
Technical Challenges	566
Configuration Management	566

Job Scheduling	568
Database Tuning	569
Functional Challenges.....	570
Patient Merge/Unmerge	570
HIPAA Redaction.....	572
IRB Integration.....	573
Going Live	574
SECTION IV RELEASE 1.0	
19 Knowledge Synthesis.....	577
Fact Counts.....	577
Fact Counts by Metadata.....	578
Ranked Dimensioned Facts	579
Correlation Analysis	584
Derivative Data.....	585
Inter-event Timings	586
Census Data	587
Timeline Analysis.....	588
Nonpatient Timelines	590
Statistical Analyses	593
Descriptive Statistics.....	593
Statistical Process Control.....	594
Semantic Annotation.....	595
20 Data Governance.....	605
Organizing for Governance	605
Data Governance Group.....	606
Data Owners	607
Data Administration	608
Warehouse Support Team.....	609
Governance Opportunities	610
Immediate Decisions.....	610
Near-Term Control.....	610
Medium-Term Growth.....	611
Release Management.....	612
Source Application Feedback.....	612
Expanded Data Sharing	613
Long-Term Evolution	614
Process Maturity	615
Translational Medicine	615
Index	617

Preface

For over three decades, data warehouses have been impacting the way we think about and manipulate data, enabling us to analyze and integrate data across our organizations in ways never imagined while designing the operational databases and applications from which our warehoused data were collected. The data warehousing movement has only in recent years crossed over into the biomedical sector. Much of what was learned about data in other sectors is applicable in healthcare, but biomedicine and healthcare offer challenges and complexities never faced in the commercial sector. Everything I learned about data modeling, storage, and querying prior to entering the healthcare sector only served as a starting point for what I would need to know to design and build biomedical data warehouses. I'd never go back; the power of data to promote effective healthcare is too astounding.

Many biomedical or healthcare institutions today are making their own attempts to design and develop data warehouses, usually with some form of clinical or operational focus. One problem with this approach is that, owing to a general lack of IT maturity in the healthcare sector generally, these implementations are often based on concepts and designs that are already seen as largely obsolete in the broader data warehousing community or are being constrained by perceived technological weaknesses that actually haven't been of paramount concern in database technology for many years. This book offers an up-to-date model that will allow your organization to jump-start, or realign, your implementation efforts, giving you a successful data warehouse implementation that is both quickly implemented and is state of the art.

To be effective, this book covers much more than generalized data warehousing. It specifically addresses biomedical data warehousing for healthcare—a discipline that requires its own knowledge base and design patterns. Biomedical and healthcare warehousing not only requires all of the basic fundamentals of data warehousing in any field, but also requires privacy and consent controls that are unprecedented in other fields, and the integration of imaging and genomic data as modalities that explode data volumes and controls. With the right guidelines, you can build biomedical data warehouses that take these things into account. Implementations can be completed in months that used to take years or that were not even attempted because of their anticipated

complexity. This book is about how to do that effectively by applying the learning that I have achieved in numerous healthcare data warehousing settings in recent years, building upon a warehouse design that I have been tuning for over 30 years.

A biomedical data warehouse is a very complex undertaking for any information technology function or group to take on. This book can't make your implementation easy, but it will make the journey much safer and more enjoyable.

What to Do with This Book

I want you to do more than simply read this book. I want you to use this book as your data warehouse architecture guide so you actually build and implement the data warehouse that I describe here. Where I can, I've attempted to provide you with the code needed to build the database and the ETL needed to load that database. You'll need to make some adjustments to make this material meet your local requirements and to conform to the type and version of database and SQL language in your installation, but otherwise this material is meant to be directly implemented. To do what I'm expecting of you, you'll need to do the following:

1. Assemble a small agile team for rapid and focused warehouse implementation. The design I offer here is fairly generic, so you don't want too big a team because they'll end up tripping over each other. There aren't as many distinct components to this design as you might think as you start reading, so your instinct to establish a fairly large team to implement a fairly large-scale data warehouse needs to be resisted until you see how the generic architecture presented here unfolds.
2. Identify how to eliminate database and functional dependency project bottlenecks. If you are like me, you've grown up in a world where you want to know all of the users' requirements before you commit to a particular database design. This book turns that notion upside down and describes a database design that is actually quite independent of what any of your users might want to do with it. By decoupling the database from the functions, implementation steps that used to be sequentially dependent become both reduced and functionally isolated. Everything in your implementation can happen in parallel, allowing for very short implementation sprints.
3. Plan a multi-iteration project approach that delivers functionality quarterly. If you've spent years wanting to build a data warehouse for your organization, it will seem odd to plan an implementation that will have users querying the warehouse within a few months, but that's what you need to do. Part of the power of this generic design is the ability to deliver quickly, but that benefit won't materialize unless you plan it that way. Don't fall into the traditional trap of thinking that the warehouse will take years to build. If you do, it will

become a self-fulfilling prophecy. Get your warehouse operationally testable within a few months and into production in less than a year.

4. Diagnose organizational and resource issues that might stand in the way. I'll talk about these throughout the book, particularly in Chapter 1, and you'll need to manage them proactively. If you reflect on your own reaction to implementing this warehouse in less than a year, imagine how others in your organization will react. One of your bigger challenges in implementation will be in developing the confidence of your organization that you can actually do it. Absent that confidence, you won't receive the support you actually need to succeed.
5. Select the most powerful clinical data sources for the initial implementation. Implementing quickly is a key concern, but implementing something really useful quickly will be your key success factor. I've seen warehouse projects fail simply because the data they chose to load early wasn't interesting enough to maintain the organization's interest and commitment. That first early implementation must give users an ability that they don't already have. It's shocking how many times data warehouses are implemented, sometimes at great cost and effort, and they aren't able to do anything better than what users could already do with existing systems and resources. The first implementation needs to blow people away.

I assume you wouldn't be reading this book if you didn't have a genuine interest in some form of data warehouse in a biomedical or healthcare setting. If you aren't sure about moving forward, Chapter 1 will offer a set of generic requirements that might help you get your organization or management moving in the data warehousing direction. The journey will be shorter than you expect, but it will also be intense. Let this design do most of the work for you, and you enjoy the results.

Author

Richard E. Biehl is an information technology consultant with 37 years of experience, specializing in logical and physical data architectures, quality management, and strategic planning for the application of information technology. His research interests include semantic interoperability in biomedical data and the integration of chaos and complexity theories into the systems engineering of healthcare. Dr. Biehl holds a PhD in applied management and decision science and an MS in educational change and technology innovation from Walden University, Minneapolis, Minnesota. He is a certified Six Sigma Black Belt (CSSBB) and a Software Quality Engineer (CSQE) by the American Society for Quality (ASQ), Milwaukee, Wisconsin. Dr. Biehl is a visiting instructor at the University of Central Florida (UCF), Orlando, Florida in the College of Engineering and Computer Science (CECS), teaching quality and systems engineering in the Industrial Engineering and Management Systems (IEMS) Department.

Chapter 1

Biomedical Data Warehousing

Data warehousing, as a technical discipline within information technology, has been evolving over the past 30 years and has emerged as fairly stable and mature only within the past decade. A key driver of this maturation is the development of large-scale database technologies that can handle the terabytes of data that a modern data warehouse is expected to hold. Without these technologies, today's data warehouses wouldn't be possible, and while the technical evolution continues, standardized models of large-scale data warehouses are emerging, and best practices are being identified.

The development of informatics in the biomedical sector has been going through a separate but parallel evolution over the past 20 years. Extensive health records once trapped in paper-based charts have shifted into electronic health record (EHR) databases that make information available far beyond the point of care. Analysis of clinical practice and outcomes data has allowed informaticists to develop metrics and practice guidelines across the continuum of care. Data now follow the patient instead of being contained within a provider's practice. As the range of data included in analysis grows, data warehousing has become essential to combining all of the data being generated and stored across the biomedical sector.

This book offers a standardized model for implementing a healthcare data warehouse. It is based on an evolving design that I have been implementing for 35 years, exclusively in the healthcare sector for the past decade. It combines best practices from the technical disciplines related to data warehousing with emerging concepts in biomedical informatics, a challenging field that is growing along with the increased use of information technology in healthcare.

Nature of Biomedical Data

The factor that most differentiates biomedical data warehousing from other functional disciplines of data warehousing is the data. Biomedical data are among the most complex forms of data you will ever need to warehouse, and

that complexity arises in areas that are difficult to anticipate if you have not spent time working in the biomedical sector. Very few of these complexities are truly unique to biomedicine or healthcare. An experienced data warehousing specialist will have run into many of them, and even optimized solutions for quite a few of them. In aggregate, the issues that make biomedical data different make it *extremely* different—in ways that require considerable learning to be able to understand and resolve.

The first and most immediate difference in biomedical data is that so much of them are textual. Traditional data warehouses are built around data aggregation as a core function, with detailed facts typically in the form of some quantity that can be aggregated. Total sales can be aggregated easily because the warehoused facts include order quantities and extended prices. Average order value can be aggregated using the same data. Because the data are quantitative, the warehouse is typically used to create time-series descriptive statistics against almost any set of stored data facts. Exceptions to this pattern are rare, and sometimes nonexistent. Many data warehouse teams would not bother storing the exceptional textual data in the warehouse, typically arguing that there would be no point since the warehouse primarily exists to aggregate quantitative data.

Biomedical data are very different. Yes, healthcare institutions have operational and financial aspects that generate quantitative data in the same ways as any other institution doing data warehousing. Patient accounts have orders, fulfillments, charges, bills, payments, and reimbursements that can be stored, tracked, aggregated, and analyzed in the data warehouse. Materials and supplies must be ordered, received, inventoried, disbursed, and scrapped. Facilities must be stocked, scheduled, maintained, and managed. Staff must be hired, scheduled, and managed; and productivity analyzed. These are some of the reasons that biomedical data warehouses are so complex. They start out already containing as much operational complexity as the data warehouses built and used in any other sector.

All of this complexity is inherent in biomedicine *before* any clinical or biological data are added to the warehouse, and the complexity of this additional data is even greater than the underlying complexity of all of that operational data. The biomedical data warehouse is already among the most complex of warehouses even before any healthcare data are added. Operationally, the customers who receive the services are not the ones who pay for those services. The physicians who provide those services usually don't work for the institutions within which those services are provided. The nurses who actually provide most of the care typically work for the institution itself, although they are accountable to those physicians who are independent. These factors make a healthcare operation much more complex than most traditional industrial organizations, all before discussing any actual clinical and research biomedical data.

Your data start to get really interesting as you start to add the actual clinical and biological data that are the cornerstones of biomedical informatics. You'll keep data at different levels of detail for millions of subject categories that all

need to be managed. Patients as individuals are the most obvious category of subjects in the warehouse, but a lot of clinical data is actually about portions of those individuals drawn as specimens. Patients are also grouped into cohorts for research or epidemiological purposes. A data warehouse will store and track biomedical data at all three of these levels, and seamlessly integrate data across these levels as needed.

The biomedical facts in your warehouse will vary from simple numeric quantities to full-fledged textual reports. You'll parse a lot of the arriving data to make it as discrete as possible, but much of the data will stay in textual form because of the ways they were entered into the source systems from which they are received. Pathology, radiology, and many laboratory reports arrive as large text blocks. Nursing and physician notes arrive as free-form text. The hospital registrar notes a presenting patient's principal complaint as a small amount of text. All of these data serve an immediate purpose in the source clinical systems because those systems are designed to present that text to the human eye of the clinician who can correctly interpret and use that information even if it isn't spelled correctly, or even if it was entered in the incorrect field on a screen by a system user. If a user types the systolic and diastolic blood pressures in the wrong order, no clinician will confuse the two numbers. All of these text data, as well as the sloppiness of much of the data in healthcare, present tremendous challenges to data warehousing.

Another area of complexity in healthcare data arises because much of what happens in healthcare never gets entered into any application system from which it can be sourced into the warehouse. Many text reports, particularly most that arrive from outside of the organization's boundary, are scanned into systems within the organization as PDFs that can be viewed, but not queried. As health information technology grows and matures, this category of complexity is reducing in frequency, but it still occurs and is likely to continue into the foreseeable future. The result is incomplete data: fulfillments without orders, orders not fulfilled or cancelled, or charges without fulfillments. The chain of evidence from observation to order to fulfillment to result to outcome is often broken in healthcare systems in ways that would be rare in industrial corporate systems. Any warehouse design dependent on these chains being maintained would fail immediately in the healthcare sector, creating challenges for the design of biomedical data structures. Again, frequency of this issue is decreasing slowly as health information technology systems and practices improve, but the sector still has a long way to go to close this gap.

Finally, if all of these complexities haven't already scared you away from biomedical data warehousing, privacy remains the single biggest contributor to complexity. Biomedical data are considered highly sensitive by the patients who are represented in the data, and the privacy of the data is of paramount concern. In the United States, the Health Insurance Portability and Accountability Act (HIPAA) regulates who can see and use healthcare data. Similar regulations govern institutions outside of the United States. These regulations require that,

in addition to simply storing the biomedical data, the warehouse must also store enough information about the data to be able to enforce privacy constraints. Patient consent data, institutional review board (IRB) data, operational use cases, and procedure or data profiles all must be available at the time of information retrieval in order to determine if the warehouse should provide certain data in response to a query. This set of requirements alone would make the biomedical data warehouse among the most complex of warehouses even if all of the other distinct healthcare requirements were absent.

Nature of Warehoused Data

Data warehousing is based largely on an analogy. In the physical world, a warehouse consolidates products or materials that arrive from many disparate places and producers, and holds them in storage until they are needed by some customer or user. The warehouse itself need not look anything like the production sites from which the products and materials are collected. Instead, it is optimized for expected storage duration and ease of retrieval. The ability to retrieve materials or products doesn't require information about how or where they were made; it only requires knowledge of the *dimensionality* of the warehouse, meaning that for example Bin A74G is in Section A, Row 7, Stack 4. The entire inventory in the warehouse is stored and retrieved through that four-dimensional structure.

The digital nature of a data warehouse is different from the physical nature of a logistics warehouse. In a logistics setting, you have to go to the section in order to get to the row, go to the row to get to the stack, and get to the stack to get to the bin you need. The bin is the intersection of four dimensions, but you can only get there through one specific physical pathway. You can't get to the bin any other way. In a digital data warehouse, the facts (bins) are at the intersection of many dimensions, and there isn't any single preferred pathway for getting to them. You can traverse the dimensions of the warehouse using any of the pathways that make sense to your purpose, and you can use them in any order. There is no preferred pathway, and that distinction is what makes the dimensional data warehouse so much more powerful than the traditional relational data warehouse.

I started developing data warehouses around 1980. The scope and scale of those early warehouses were small compared to what we implement today, but the reasons we built them were the same. We needed access to data along pathways that our existing computer systems were unable to support. The big database technologies at the time were hierarchical: data defined either explicitly as part of some other data, or implicitly as part of some data through a foreign key. Customer records encompassed sales call records, which encompassed the orders for those customers, each of which encompassed the shipments required to fulfill those orders (Figure 1.1a). The data were stored along that specific

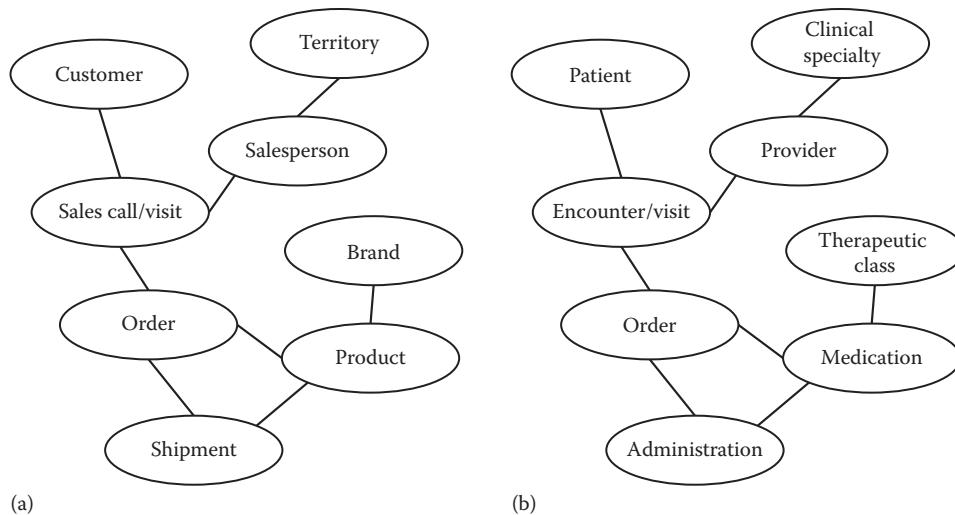


Figure 1.1 Nature of data pathways over time: (a) 1980 and (b) 2010.

hierarchic pathway and, as in the logistics warehouse, could be accessed easily as long as your access pathway mirrored that hierarchy.

The database technologies at the time were very good with hierarchies. We could jump into the data for an order without having to know the customer or sales call, because we were jumping in using the standard pathway, simply accessing the data for a specific branch of the hierarchy. Problems arose when we needed to access data along different pathways that required retrieval of data from across different branches of the hierarchy. If we wanted to look at product data across all shipments or orders, those data were deeply buried throughout the existing databases, and those databases weren't designed or built to retrieve information that way. The short-term answer was to build new systems to analyze product data, and then copy all of the data from the first database into the new database. The new database wouldn't be able to answer customer-related questions easily, but it could be optimized to answer product questions, including summarizing data by brand. These new systems weren't data warehouses. They were new *application systems* that would be managed for years as data silos in the organization, and there were a lot of them.

While marketing was building brand-product applications and databases, the sales organization was building its own applications and databases. They needed the data from the original system organized by salesperson so they could roll-up the data into territories and regions. Since the original system couldn't produce data that way, another silo system would be built with its own data copied from the original system. It couldn't be used to answer customer or product questions, but it provided easy access to sales data in the way the sales organization needed to see it. Throughout the 1970s and 1980s, organizations added more and more distinct silos of data throughout their systems environments. During this same period, we realized that many of these silos were being built for the same reasons, and with extremely similar data access

and analytical requirements. The differences tended to be in the dimensionality of the datasets, and the pathways needed for routine access. Early data warehouses allowed many of those needs to be met without having to build a new silo system for each new need. One data warehouse with an appropriate data architecture could do it all.

The technologies on which the early data warehouses were built have continued to evolve, and today they are more powerful and capable than ever. As storage and access technologies have become more routine, the field of data warehouse design has expanded to include a broader range of database designs and options. Over time, the field converged on two general categories of data warehouses: relational, that keep data in a large set of tables that obey the rules of relational algebra; and dimensional, that keep data in fewer tables that differentiate the facts being stored from the dimensional reference data that describes those facts. The database designs that formed the foundation of these two warehouse types have matured beyond anything imaginable in those early years.

The first data warehouse I implemented was in the energy sector in 1979. It was a seven-dimensional warehouse, although the term *dimensional* hadn't come into common use at that time. It fit today's definition of a dimensional warehouse because it had just one fact table that was surrounded by the seven dimensions through which data could be accessed or aggregated. Over the next 25 years I continued to implement financial and logistics data warehouses in pharmaceutical, aerospace, agriculture, publishing, insurance, petrochemical, and various manufacturing enterprises. The designs kept getting more sophisticated as the tools continued to improve and the general models matured. The books of Ralph Kimball in the early 1990s helped me standardize many of the concepts and conventions in my designs. In a sense, I can trace every data warehouse I've ever built back to that first small energy-oriented data warehouse. The databases and tools keep getting more powerful, allowing our idea of what constitutes a good data warehouse to keep evolving and maturing. Every new data warehouse is an outgrowth of those earlier implementations.

When I moved into the healthcare sector in 2006, I was overwhelmed by the complexity of biomedical data, but I was also struck by the extent to which I recognized many of the data challenges as being very similar to those I had encountered in other sectors. For example, patients had encounters which contained orders that resulted in administrations or fulfillments (Figure 1.1b), and this pattern mirrored the customer-sales-order-shipment hierarchy that I had implemented so many times before. Electronic health record (EHR) systems captured all of that data along hierachic pathways, and accessing the data any other way was difficult. Looking at medications across orders or administrations, or aggregating those data by therapeutic class, was almost impossible without creating a new data silo for that purpose. Grouping encounters by provider, or aggregating those data by clinical specialty, was just as difficult. Perhaps one

benefit of the healthcare sector being decades behind industry in fully embracing information technology was that the proliferation of all those data silos could be avoided, and more integrated solutions could be inherited from those other more mature sectors.

This book describes specific ways to design and build a data warehouse that can solve many problems in biomedical informatics. Data warehouse implementation is a significant information technology initiative. It should be managed by IT staff who have the best practices and organizational mind-set to design and construct complex information systems. Building databases, programming loads, implementing query tools, and providing an operational environment are best left to the IT professionals. Organizations that implement data warehouses typically have very high expectations for what using them can achieve. To meet these expectations, the implementation of the warehouse needs to be taken seriously across the enterprise as a significant information systems initiative. In addition to IT skills, the implementation requires a level of commitment and perseverance not usually available in the user community.

Business Requirements

Organizations considering data warehousing must start by defining the business requirements to be met by their implementation. Most of the institutions I've worked with have thought their business requirements were fairly unique, or at least distinct, compared to other organizations, but I've found that the actual requirements these organizations have been trying to identify and satisfy are quite stable and predictable. Prioritization of requirements might change from institution to institution, but the requirements themselves usually fall into seven basic categories:

1. *Integrated data repository*: Achieve data integration across a complicated data landscape, both *within* the institution (i.e., from EHR out to ancillary systems) as well as *across* institutions (i.e., continuity of care, medical home, accountable care)
2. *New and diverse access paths*: Provide access to data along any pathway, and at any level of detail, without regard to how the data were received
3. *Enhance data quality*: Improve upon problems encountered in the data, or at least prevent harm from their use by identifying weaknesses in the data
4. *Early access to complicated data*: Make it easier to access complex data through a layered approach that doesn't require users to have complete knowledge of the data before access and use can begin
5. *Scalable to include more sources*: Be able to load and handle any data source identified in the future, typically requiring the warehouse to be able to store sources that might actually contradict each other

6. *Semantic pathways into data:* Support access to data through conceptual avenues that might have been unknown or unsupported by the systems that provided the data
7. *Longitudinal phenotypic data:* Orient data from operations and clinical practice so actual patient phenotype data can be derived over extended time frames

I typically encounter all seven of these requirements at my client organizations, although not all seven are defined at the beginning of each initiative. When one or more is not being articulated by a client, I try to initiate discussions that will elicit them. If I focus exclusively on implementing the first few, I later find the latter few much harder to implement. The last one (e.g., phenotype) is particularly hard to accomplish if it isn't considered when trying to satisfy the first (e.g., integration). It's fine if the earlier requirements get higher priority in the first implementation wave, but it's important that all seven be considered when planning long-term strategy, and even short-term data mapping. From the very beginning, the warehoused data should feel like it describes patients and their states of health, not the contents of computer systems and the clinical activity that populated them.

Functional Requirements

To meet your organization's business requirements, you'll build and implement a biomedical data warehouse so your institution can begin doing effective warehouse-based biomedical informatics. The question becomes: What functional capabilities must the data warehouse include in order for your data warehousing activities to conform to the business requirements you've defined? As with business requirements, I've found that the functional requirements for a data warehouse are fairly predictable and stable across institutions. The biggest implementation problem that I've seen is defining functional requirements that are too narrow. Focusing too narrowly on immediate needs results in a technical implementation that meets those needs sufficiently, or even well, but can't be expanded to include additional functional requirements that were side-stepped in the early, narrow push for implementation (i.e., defects of *omission*, in quality assurance jargon).

Functionally, the biomedical data warehouse is a comprehensive set of datasets and tools that provides encompassing access to patient-centric data from across the applications and databases used throughout any major institution, whether community hospital, academic medical center, or research center. At its essence, the warehouse collects data facts from across the various systems in the biomedical setting, organizes them, and makes them available to users throughout the institution in ways not available without centralization and consolidation. During the centralization processes of loading data into the warehouse, additional value is added to the data in the form of verification and

validation of the data itself, as well as systematic analysis to identify trends and patterns in the data that might otherwise remain unrecognized. There are three main functional pathways that must be enabled by the implementation of the warehouse in order for all of this to happen.

Data Queries, Reports, and Marts

Paramount among the functional requirements is the need to receive reference and factual data from data sources and make those data available to user-analysts through queries and reports. Many organizations have only this functional requirement in mind when they discuss implementing a data warehouse. While I don't consider it the *only* requirement, I'll grant that it is a warehouse's *primary* functional requirement.

The details of this requirement generally focus on the quality and completeness of the query results on the user side, and the correctness of the data on the data source side. By defining these primary requirements functionally and in detail, the queries and reports delivered to users need to be *good* results and not just *any* results. I've seen too many warehouses deliver bad results because the requirements for goodness weren't adequately defined, and the technology perspective took over as criteria for quality and project compliance.

The queries and reports produced by a data warehouse should be managed in a controlled way so similar queries against similar data always produce similar results. The result sets should be meaningful to the user, which means that the data need to *actually* describe the real-world circumstances that it *claims* to describe. Data ambiguity commonly arises when unrecognized measurement error is present in warehoused data, such as when disparities exist between clinical and system timestamps in the source applications, or system data have been loaded into the warehouse as proxy for clinical data. Data must be as clear as possible, both in the data presented, and in any erroneous or suspended data not included. Results must be compliant, both with the standards against which they were defined, as well as within the privacy regimes within which access is granted. Also, data appearing in query results must be traceable back to the sources so users can access those sources for further details or simple validation.

On the source side, the functional requirements include making sure that the reference data brought into the warehouse are stable over time, and aligned with industry and organizational standards. If any local codes that lack stability are being brought into the warehouse, such as new lab test procedure codes being assigned to small variations in how a test is performed, the power to look at data longitudinally can be lost because of the temporal fragmentation created by the unstable reference data. The facts received against reference data must also be compliant with any reference definitions or standards that have been loaded. Noncompliance issues commonly arising at the sourced level include qualitative data received where quantitative data were expected, or quantitative data in

unknown or unexpected units of measure. The user functional requirements can't be met if the source inputs lack conformance to these input requirements.

Data Issues and Hypotheses

Recognizing that there will be failures in meeting requirements in the primary data pathway from data source to user analyst, a secondary pathway is needed that offers a feedback loop for data owners to work toward improving the quality and usefulness of data. As a data warehouse specialist, I consider this the more *important* pathway through the warehouse because it supports the continuous improvement of both data and the relationships among the data. If this secondary pathway is working well, meeting requirements in the primary pathway continuously improves because the data and the way they are organized are constantly being refined and made more robust.

The nature of this functional feedback loop is that by looking at the data already in the data warehouse, data issues that weren't or couldn't be identified at the source analysis level can be discerned. By looking at patterns and inconsistencies in the data, the data warehouse can be used to develop hypotheses about how the data either should have been loaded, or might be improved in order to better meet the primary requirements. These issues and hypotheses are communicated to the appropriate data owners for investigation and resolution.

Data owners are part of the network of support personnel that provide for data governance. Chapter 20 deals with data governance in the completed data warehouse, and how the various people providing that support interact. There will typically be a group of people who are close enough to some of the data to be identified as owners of those data, either because they have a strong relationship to the biomedical functions supported by those data, or to the management of the applications systems from which the data are drawn. As long as the hypotheses generated by the data warehouse are relevant to the data owner's area of expertise, timely enough to be useful to analysts using the warehouse, actionable in terms of functionality available in the warehouse implementation, and traceable to the involved data and their sources, the data owner is empowered to define corrective actions to be implemented by the data warehouse support team.

My experience is that this feedback loop is always embraced by the organization that implements it, precisely because the process of implementing a biomedical data warehouse always uncovers extensive data quality problems in the source data being loaded. At its worst, poor quality and corrupt data can prevent a warehouse from meeting its basic user requirements. The need for this kind of feedback loop should lessen over time, but since the ongoing evolution of the warehouse usually involves sourcing and loading new data from additional systems, there is always a new source of poor quality data to be dealt with. Data management by data owners needs to be a central function of every biomedical data warehouse.

Performance and Control Data

The functional requirements for the biomedical data warehouse involve a third important pathway for support personnel and resources to continuously improve and optimize the actual functioning of the data warehouse. As with the data owner pathway, this pathway is also a feedback loop where performance data coming out of the warehouse is used by the support team to adjust the performance and control settings of the warehouse.

The control data produced by the warehouse need to be auditable so it can be verified and tested by the support team over time, actionable so options exist for making adjustments to the warehouse based on what is discerned, and compliant with already existing settings and policies governing data warehouse performance and control. The warehouse team needs to be able to make changes to the warehouse that help it remain stable for any users, and work toward optimizing actual performance for those users. Some of those changes will require policy updates or resource enhancements to be made by organizational leadership.

Never-Finished Warehouse

The reality is that I've never finished a data warehouse. I've completed many warehouse *projects*, but never to the extent that I could assert that all known requirements could be satisfied, or that all known design options had been exercised. In that respect, the data warehouse is more of a product-line than a stand-alone product. Product-lines evolve over an extended period of time, usually many years, introducing new or optional features that continuously expand the range of requirements being satisfied. Under such an evolutionary approach, it's hard to ever claim that the product is finished. The points at which different organizations stop implementing their data warehouses vary, but the reason they stop is quite consistent: diminishing returns.

No organization implements a data warehouse just for the sake of doing so, and it tends to implement the more important requirements first. This means that with each stage of implementation, the requirements that have already been met were more important than the requirements that haven't yet been met. The value of each incremental extension of the warehouse diminishes as the project team works its way down the requirements list. Inevitably, the value of the next incremental release is exceeded by the value to be attained by allocating the team to some other endeavor, and the warehouse implementation will end.

One way to keep an ongoing commitment to continual evolution of the warehouse is to work to expand the value of what the warehouse can provide. Part of this comes from working with the data governance function to continually identify new opportunities to source more data, or enlist new users to take advantage of the existing data to do new things. The warehouse environment is not a closed system in which the value of the next requirement

to be met will diminish over time. Work with your governance group to add new and exciting requirements to the list over time. Rather than diminishing, the value of each incremental improvement can be increased, with the warehouse becoming an investment that the stakeholders want to add resources to, not one they want to cut their expenses on. The warehouse can be something that will never be finished, but that comes to be seen as a strength, not a weakness.

Organizational Readiness

This book is my attempt to give you everything you need to implement a very powerful and flexible biomedical data warehouse, but these instructions alone can't guarantee success. In addition to following the design that I offer here, you'll need to have other capabilities established in your organization that will support and promote a successful implementation project. These capabilities, taken collectively, form the organizational readiness prerequisites to implementing my design.

Your organization will need to have a strong and viable data governance function in place in order to be successful at data warehousing. The team you put together for data governance doesn't need to be technical, because they guide everything about the data warehouse except the technology. Their initial responsibilities include developing a release plan for the warehouse roll-out over the short, medium, and long term. They'll decide which data sources across your institution should be loaded into the data warehouse, and in what order. The first few of these impact the development effort because they set the scope for the first release, but the release strategy rolls out over a period of many years. Four to seven sources should be assigned for the initial release, 2–3 sources per calendar quarter over the next few years, and 10–15 sources each year for the following several years. Having a plan in place—regardless of how many times it will be changed over the intervening years—allows for establishing the appropriate breadth of focus for the first-year development effort. If you have a plan for loading hundreds of data sources over the next 5 years, it is very unlikely that characteristics of the first few sources will bias the design and implementation process for your new warehouse.

With a release plan in hand, the development team's emphasis will be on making all of the analysis, design, and build activities for the data warehouse Extract-Transform-Load (ETL) as repeatable as possible in support of the release strategy. Repeatability implies an improved level of process maturity, and higher process maturity is often achieved, in part, by specialization of functions and responsibilities across the IT function. In a data warehousing environment, data administration is a core area that benefits from specialization. If your organization doesn't have a data administration group to support data standards and conventions, your warehouse team will most likely run into obstacles whenever data definition or integration issues arise among different sources being targeted for the warehouse. While the warehouse team can make choices and decisions that

will aid implementation, they aren't in a position to make and enforce those choices across the institution. Many warehouse requirements will necessitate clarifying definitions of data fields or derivations, or the way they are captured or used across the clinical landscape. The impact of optimal solutions for those requirements might require changes throughout the organization, and the warehouse team will have neither the time nor authority to drive some of those changes. A strong data administration function can enable those changes, and take a great deal of organizational pressure off the data warehouse implementation team.

An initial production-version warehouse should be implemented as soon as possible, *preferably within a year*. Implementing the warehouse into production prior to getting these readiness functions up and running—while sometimes seeming like an expeditious approach—always results in longer-term delays in implementation. I outline my basic governance model in the last chapter of this book, not because it is unimportant and can wait until the end, but because appreciating the breadth and depth of that function requires knowing much of what this book covers. You'll read it last, but should do it first. Without effective data governance and administration, a 1-year data warehouse implementation won't happen.

Implementation Strategy

The approach I take in this book presumes that you will implement a series of incremental releases of the warehouse, starting with an internal Alpha version at 3 or 4 weeks into the project, a more functional but still internal Beta version at 3 months, a fully functional but not production-ready Gamma version at 7 months, and the final production Release 1.0 at about 1 year. Your actual timeline will vary from these benchmarks for a variety of reasons, but the essential flow should still work for you. If your organization tends to try to implement full versions of systems in one iteration, this incremental release approach will seem foreign at first, but I strongly recommend it. Your warehouse will be very complex, and probably unlike any other systems your organization has implemented. An iterative approach reduces the risks associated with learning curves and ambiguous requirements, delivering function early in order to maintain user and management interest and commitment.

The implementation approach in this book aims for a fully functional warehouse based on a flexible generic architecture that is divorced from the requirements of its source applications. However, implementing a generic architecture around a new conceptual model that is initially unfamiliar can be extremely difficult. Therefore, this book presumes that your project will start out with fairly hard-coded brute-force architecture. This allows a small subset of the warehouse to be built very quickly in the Alpha version, even while the team is assimilating the more detailed aspects of the design. By the end of the book, that investment will have been incorporated or converted into the completely generic architecture.

Warehouse Project

The most immediate challenge in moving forward is typically putting the appropriate human resources in place to begin work. The immediate question will be the extent to which the existing organizational data team can be shifted to work on the new warehouse project while also meeting the needs of supporting any legacy reporting or pre-warehouse environments in the near-term. I've seen organizations hire completely new staff for their data warehouse teams, but I've also seen organizations hire new staff to back-fill positions so existing resources could be allocated to the warehouse development effort.

I usually work with very small implementation teams, often only a single developer. I've also worked with teams that were far too large for effective development and productivity. The core resource that I recommend for a data warehouse implementation is a three-to-five person (full-time equivalent, or FTE) project team, dedicated over the entire project time frame:

- Data Warehouse Architect (0.5 FTE, heaviest up front)
- Data and Functional Analysts (1–2 FTE)
- ETL Developers (1–2 FTE)
- Application Developer (0.5 FTE, intermittent)

In addition to this direct team, additional resources should be available to the implementation effort on an as-needed basis:

- Project Manager (0.5 FTE, continuous)
- Database Administrator (0.5 FTE, intermittent)
- Data Administrator (0.5–1 FTE)
- Integration Specialist (0.25 FTE)
- Business Intelligence Designer (0.5 FTE, intermittent)
- Organizational Change Agent (0.5 FTE, focused on Governance)

Based on my experiences at multiple biomedical clients, I recommend preventing this team from growing large. An initial growth in team size often creates the feeling of faster progress, but inevitably involves learning curve and coordination issues that have caused me to scale back the team to an effective core. My implementations have been more effective with core, integrated small teams. If I can't have my optimum three-to-five team members, I'd rather have fewer than more.

I recommend an iterative development lifecycle that implements functionality early and often, and I've organized this book around that approach. Barring major obstacles and readiness issues, the process takes about a year. If you stumble along the way, at least the early versions start providing real value to users. The chapters that follow define a strategy for implementing Alpha, Beta, and Gamma versions of the warehouse along the route to a final Release 1.0 production version.

ALPHA VERSION



The Alpha version of a healthcare data warehouse is the one that implements just enough of the basic design pattern to illustrate its core capabilities, while loading a small sampling of limited data for demonstration purposes. The point isn't to provide production-ready functionality, but to enable an easier way for everyone involved to visualize what the new warehouse paradigm entails by actually looking at a core subset of it that can be seen and examined. I usually allow 3–4 weeks for its development. Some people refer to this earliest version as a proof of concept; I typically reserve that term to describe a working subset of the Alpha version that I try to have available after the first week (yes, *week*) of development.

Chapter 2 introduces the basics of dimensional modeling, along with some history to support my assertion that the star schema architecture is the best choice for most data warehouses. Chapter 3 offers an ontological framework for analyzing source datasets into three semantic layers that define the biomedical data that we'll eventually load into our warehouse. I then offer my logical design for a biomedical data warehouse in Chapter 4; one that enables the design principles laid out in Chapter 2, while supporting the semantic requirements defined in Chapter 3. Chapter 5 then proceeds to define the physical design for the dimensions of the warehouse, so it can be built both generically and very quickly. Finally, Chapter 6 explains how to load the Alpha version of the warehouse, so users can start executing queries against early data.

The Alpha version will typically be queried by members of the development team during the second month of the project and by a broader pilot user group during the third month of the project. Additional Alpha version loading and testing can continue in parallel as the warehouse team moves on to the development of the Beta version of the warehouse.

Chapter 2

Dimensional Data Modeling

The starting point for the Alpha version of the data warehouse is the decision to implement the data warehouse as a dimensional model instead of the historically more common relational model. The two approaches are more interchangeable than many people realize—and we'll discuss that interchangeability later—but the actual warehouse needs to be developed firmly in one, or the other, model. Attempting to have it both ways—some dimensional functions and features side by side with relational model functions and features—can create more problems than it solves, and requires a level of expertise that is not usually available to most biomedical or healthcare organizations. At the very least, it would be inappropriate in a proof-of-concept Alpha version. For that reason, we'll implement the data warehouse as a dimensional model; so let's explore exactly what that means.

Evolution of Data Warehouses

A general multidimensional data warehouse contains two types of tables: fact tables and dimension tables. A fact table is a very large centralized table that contains specific measurable data of interest to the enterprise. The fact tables are defined, and surrounded, by a collection of dimension tables that represent the natural dimensions within which the facts are defined. The dimensions are said to provide the context for the information in the central fact tables.

By contrast, a relational data warehouse represents the historically traditional database implementation of the data warehouse concept. While the warehouse contains many tables, no explicit separation is made between fact tables and dimension tables. Enterprise information is contained throughout the warehouse. Because the data are distributed, more knowledge of the data is needed in order to build queries, and the database can only be optimized for certain families of queries. Queries that access data through less optimized paths tend to cost more, run longer, and can be much more difficult to design.

Relational Normalization

Historically, the relational view of a database has been determined by putting data through an analytical process of normalization. The rules of normalization attempt to get data into optimal positions within the database schema in order to obtain good access performance during maintenance activities. A lot of normalization makes sense, regardless of your intended database structure. The baseline rule of placing data into tables in which the data are dependent upon the key, the whole key, and nothing but the key of the tables—the rules most commonly associated with first, second, and third normal forms—makes a lot of sense. The problem I've always had with normalization is that the level to which it is taken is arbitrary, and the dependency between a data value and its associated table key is subjectively determined. We end up putting data where we want it, and using some of the rules of normalization to justify our choices. We conveniently ignore other rules of normalization that tell us our design is wrong.

Here's an example: Lab results. A typical lab results table in a database will have a column for the actual lab result, and a column for some form of abnormalcy flag or indicator, and one or two columns for the reference range. The key for the table will likely include the patient, encounter, some order information, and usually an identifier for the lab test. The lab result, abnormalcy indicator, and reference range all arrive in a single transaction, so database designers usually end up putting them into a single table. At a certain level, the three columns are all dependent upon the patient, encounter, order, and lab test keys; so the design is justified by the rules of data normalization.

The problem is that, while the design does follow some of the rules of normalization, it violates others. First, the lab test might have been recorded against the wrong encounter (but for the right patient, hopefully). Reassigning the order to the right encounter shouldn't require updating the order results; so the result columns aren't actually dependent upon the encounter key. The original lab order might be dependent on the encounter, but the results aren't. The results aren't directly dependent upon the patient either. They are for the patient, but it is the encounter that connects the data to the patient, not each individual result. Correcting an invalid patient identifier using a patient merge transaction shouldn't require updating all of that patient's lab results. Likewise, the abnormalcy indicator isn't just dependent on the keys of the table. It is also dependent upon the reference range value. To normalize that indicator, we need an additional table that includes the reference range in the key. Since the actual result isn't dependent upon the reference range, we won't be able to store the result value and abnormalcy indicator in the same table.

Fully normalized, the only reference to the patient would be in the encounter data. The only reference to the encounter would be in the order data. The result values would all reference the order, and the abnormalcy indicators would reference the order with the reference range. If we want to view all of these data together, we simply join the tables together as relational databases

are intended to be used. I'm not saying that we have to run out today and redesign all of our lab result tables. We will always have lab result tables that include patient and encounter identifiers, and list the abnormalcy indicator right alongside the result value. I'm simply pointing out that the claim that relational databases are normalized is usually only a half-truth. In fact, we've got a term for these kinds of trade-offs: denormalization. We denormalize our data whenever keeping it normalized doesn't give us the results we desire. We put the abnormalcy indicator alongside the result because we know the reference range isn't going to change after the fact, so the lack of functional dependency between the two values won't impact our use of the data. We put the patient and encounter in the result table because we never look at the lab result without that data, and we don't want to have to always join several tables to get back to the patient data.

The normalization–denormalization approach will always result in the most-used tables having more foreign keys than would be justified by the identification needs of the data in those tables, and will often combine data values that would not be put together if only the rules of normalization were the guide. This pragmatic approach to data design is fine with me, as long as we don't try to pretend it is what it isn't. I want to show you a star architecture based on dimensions and facts. It's a powerful model, and the fact that it selectively obeys some of the rules of normalization while discarding others shouldn't be held against it. I want a level playing field.

Dimensional Design

The dimensional data warehouse offers significant improvements and benefits over the older normalized relational approach. To add new types of data from new source systems to a relational data warehouse typically requires designing new tables and columns in the database that will store the new data. This is caused by our nominal adherence to normalization rules that typically demand that new kinds of data be placed into tables with different keys and functional dependencies than the data we already have. In the worst-case scenario, new data might be in conflict with the warehouse's existing design, necessitating a complete redesign of table keys or indexes; or a foregoing of the new data altogether because of its difficulty or impact. This usually requires a full information technology project to design the changes and coordinate the implementation. By contrast, dimensional data warehouses are remarkably stable over time. Typically, most new data can be added to a dimensional warehouse without any database changes at all. It still typically requires an IT resource to implement new data, but the effort required, and impacts endured, are typically several orders of magnitude smaller than the same change in a normalized relational warehouse environment.

There is one exception to what I just said. The dimensions in our warehouse will have specific columns defined for filtering queries upon

operation. A patient might have a date of birth, or a drug might have a therapeutic class. If new filtering columns are identified after implementation, they would ultimately require database changes in order to be implemented. This happens only rarely. Compared to the hundreds or thousands of values that can be added to the warehouse fact tables without requiring database changes, any new dimension filter columns are small. Also, I'll share a strategy for adding those columns first as non-database-changing facts, to be followed by a simple algorithm for migrating any new filter columns into the dimension tables at the same time; mitigating any impact any new columns might otherwise have. The details of these filtering columns can wait until we see the dimension design pattern in Chapter 5.

The ease with which information can be added to the dimensional warehouse is what provides for the excellent flexibility of this approach and enables a migration path for users that avoids the major expense and effort of a "big bang" implementation. The warehouse can start small, with only a few critical data feeds; with new data being added in successive releases with minimal impact on early users. This also supports changes and shifts in the actual source application systems that provide data to the warehouse; whether shifting from diverse uncoordinated systems toward a more centralized enterprise solution, or through the addition of specialized niche applications that are needed to round out the data in the warehouse. Release schedules are typically measured in months, where similar changes in a traditional relational warehouse can take years.

Because of the stability of the dimensional warehouse model, the costs incurred to implement new releases of the warehouse over time, usually with each release adding additional data feeds to the warehouse, are much lower than might be anticipated by comparing a release development to more traditional data warehousing or information systems projects. Once the startup costs of the first release have been incurred, two major elements of future costs have been eliminated.

First, the actual database structures that support the warehouse are extremely generic and stable. It is extremely unusual to require extensive database changes as new releases are introduced over time. Most of the data warehouses that I have built on the model being described here have not required any redesign activity after the first release. An initial production version that might take in half a dozen data sources can grow over time to over 100 data sources without changing the database structure that was implemented in the initial release.

The stability of the dimensional model can be illustrated with a simple example. Imagine needing to store four simple values from a vital signs event ([Figure 2.1](#)). In a normalized relational model, those four values are likely to be represented by four columns in a table, most likely one specifically tailored as a Vital Signs table, possibly as a subtype of some form of general Observation table. The exact configuration of tables and their relationships will depend upon the detailed data normalization strategy of the design team.

The figure consists of two parts, (a) and (b). Part (a) shows a relational warehouse model with four columns: Diastolic BP, Systolic BP, Oral temp., and Heart rate. Each column contains a single value: 98, 62, 98.4, and 72 respectively. Part (b) shows a dimensional warehouse model with two columns: Observation (dimension) and Value (fact). The Observation column lists the four vital signs, and the Value column lists their corresponding values: 98, 62, 98.4, and 72.

	Observation (dimension)	Value (fact)
Diastolic BP	98	
Systolic BP	62	
Oral temp.	98.4	
Heart rate	72	

Figure 2.1 Simplified illustration of relational warehouse model (a) versus the dimensional warehouse model (b) for four simple vital signs facts.

In the dimensional model, those four values will be stored as independent facts in a fact table. Each value will be in its own row. Each row will share overall dimensionality (e.g., patient, encounter, provider, date, and time—*not* shown in figure) of the other three facts. Differentiating them will be the Observation dimension, which captures the definition of the actual value embodied by the fact. In the relational model, the observations were *implicit* in the column names. In the dimensional model, those observations are *explicit*, and can be searched and queried accordingly. As explicit data, a user need not know of the existence of any particular Observation type in order to be able to query the data. In the relational model, a user needs to be aware of the existence of each column in order to query them correctly.

The real value of the dimensional model is quickly made evident when the need arises to add new data to the model. Imagine needing to add Respiration, Pain Level, Radial Pulse, and Tympanic Temperature to the example in Figure 2.1. The relational model requires four new columns to be added to the table. The dimensional model requires four new rows to be added to the Observation dimension. The former could take days or weeks depending upon the complexity of the database and its performance parameters, while the latter takes only minutes.

Second, the ETL architecture needed to load data from multiple sources into the warehouse also stabilizes during the first release. Those four new columns in the example above are completely unknown to any existing ETL code. New ETL code is needed for every newly sourced column. In the dimensional model, since there are no new columns, there's no need for new ETL (within the limitations of the generic architecture we develop for the ETL in the Beta version of the warehouse).

As a result of this initial investment in database design and ETL architecture, subsequent warehouse releases require far less marginal investment than might be seen in other warehouse or system architectures. Relational warehouses, in particular, typically require additional tables and indexes for each new data feed implementation; resulting in new database design and ETL architecture investments being required for each subsequent warehouse release. A full-scale

relational biomedical warehouse will eventually include several thousands of tables, while a dimensional version of the same data will typically max out at fewer than 200 tables. There are a little over 150 tables in the data warehouse design described in this book.

The Star Schema

Dimensional data warehouses are large aggregate databases that bring together an enormous amount of consolidated subject-oriented enterprise data for aggregation and analysis. The size and scale of today's data warehouses are largely enabled by the advances in database and storage technologies that have taken place over the past 30 years, while the semantics of data warehouses have evolved in parallel under different driving forces.

A central paradigm of dimensional warehousing is the separation of fact data from dimensional data, with facts comprising the quantitative and observational data of interest, and the dimensions comprising the variety of contexts in which those facts are collected and understood. The facts could be said to exist in the center of a cloud of dimensions that was often discussed by drawing a star and labeling the points of the star according to the desired dimensionality of the facts that would be included (Figure 2.2). In those early days of warehousing (e.g., the late 1970s), the five-pointed star seemed appropriate because database technologies were only mature enough to handle about five dimensions in a reasonable design, and the larger warehouses that would require more than five simple dimensions were beyond the cost-effective storage technologies of the day.

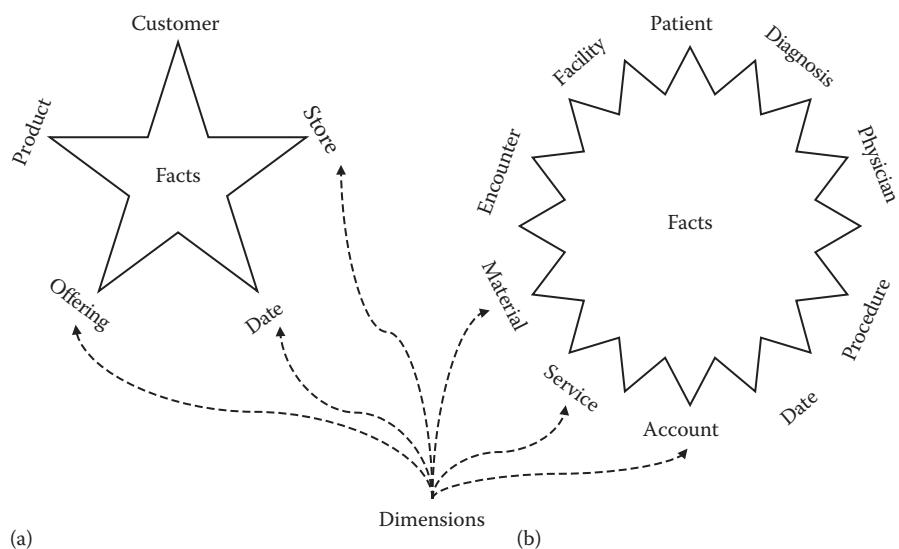


Figure 2.2 Evolution of the “star” schema paradigm in data warehousing: (a) a traditional sales data view with a five-pointed star (c. 1987) and (b) a much higher dimensional clinical data star with 15×20 dimensions (c. 2007).

I recall the design discussions around my first star schema warehouse in 1979. The requirements were dominated by five dimensions: product, location, market, calendar, and metadata. Problems arose because some of our facts needed a second product dimension in order to store facts about product conversion, such as bills of material. We needed to know what raw products were being consumed in the production of which finished products. We also had some facts where we needed a second location dimension so we could define both source and destination locations for import or export facts. That gave us requirements for seven dimensions, and it took months of debate and argument before we finally committed ourselves to attempting the implementation in that form. In the end, we rested our justification on the notion that only a small percentage of facts would need those extra two dimensions, and that we'd be able to tune our database to handle the strain of those extra foreign keys and indexes.

Today's warehouses routinely exceed the five-dimensional limitation of the early traditional stars, but the vernacular naming of the star schema persists. The hard part of dimensional modeling isn't the complexity of having more dimensions. That's actually made fairly easy these days by the maturity of the database technologies, and by the availability of standard design patterns for the construction of those dimensions. The hard part today—actually brought about by our technical ability to have many more dimensions—is actually figuring out what dimensions we require. Healthcare data is very complex in its own right and determining the dimensions we need requires us to deal with that complexity. In many ways, data warehouses were easier to design when we could only have a handful of dimensions.

Figure 2.3 illustrates a very simple healthcare dimensional warehouse model. It's a star schema even though over time we stopped drawing the actual star graphic as the number of dimensions (i.e., star points) has grown. This example

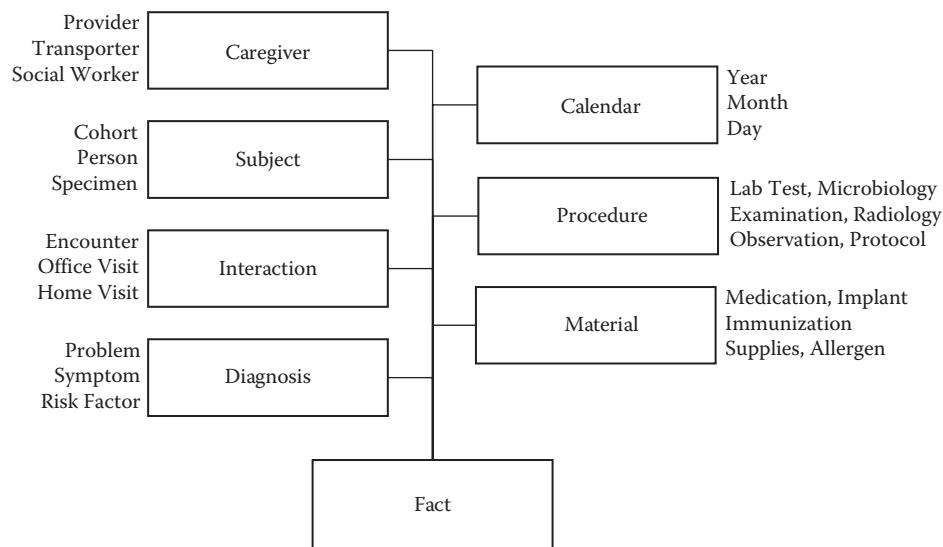


Figure 2.3 Example of simple healthcare dimensional star schema.

is a seven-dimensional star with a single fact table. The dimension boxes in the figure aren't database tables, they're representing the dimensions. The actual table structure of each dimension in the database is not depicted in the figure at all. The words written just outside each dimension box in the figure should be interpreted as some of the types of data that are expected to reside within the dimension, and will typically be the subdimensions of the dimension. The list of subdimensions in the figure for each dimension should never be interpreted as exhaustive. There will often be more than double the number of subdimensions shown in figures depicting a dimensional data warehouse.

Subdimensions coexist within the same dimension, and are typically defined by slightly different types of detailed data stored in the dimension for those subdimensions. For example, the Subject dimension in the figure is shown to include Cohorts, Persons, and Specimen. This means that when a fact connects to the Subject dimension, it could be a fact about an individual patient, a cohort of patients, or a specimen associated with a patient. Each subdimension might be described by different data that are distinct to the subtype, and the differences of which don't affect the ability of a fact to connect to the dimension. For example, a Person in Subject might have a Date of Birth that the other two subdimensions wouldn't use. A Specimen in Subject might have a Tissue Type that the others might not use. It is the attempt to normalize all of these data that causes the relational warehouse alternative to end up with so many more tables than the dimensional model.

As the design of dimensional star schema data warehouses has matured in recent decades, the problems of design have shifted from the technical arena back into the biomedical arena. Two major analytical challenges face the data warehouse designer: What should be the dimensions and subdimensions of the warehouse, and what level of detail, or grain, should each dimensions support? It used to be that the answers to these questions could be derived from the limitations of the database technologies involved. Today's technologies offer no automatic limits, so the designs have to actually be based on an analysis of the biomedical and organizational requirements for which the warehouse is being developed. That's good for biomedicine, but it's also hard.

Dimensions and Subdimensions

The dimensions of the data warehouse embody concepts that provide the semantic context for all of the facts to be stored in the warehouse. Each distinct concept constitutes a subdimension, or logical dimension, of the warehouse. Logical subdimensions are typically collected into physical dimensions within the database technology that implements the warehouse. The foreign keys within a warehouse fact table point to the physical dimensions, and within the physical dimensions some form of categorizing property identifies the appropriate logical subdimension. In the context of this discussion, referring to dimensions as physical, and subdimensions as

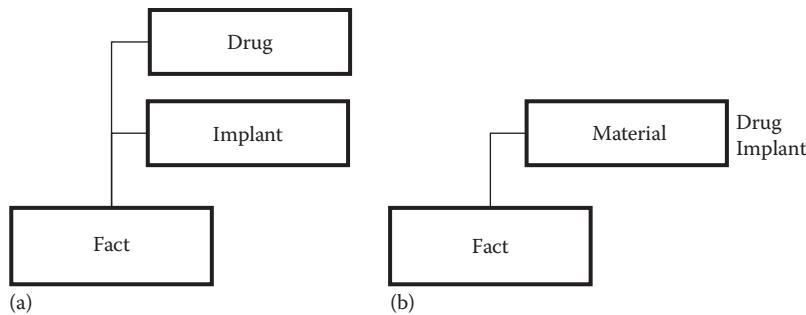


Figure 2.4 Candidate Drug and Implant dimensionality: (a) two separate dimensions and (b) one dimension with two subdimensions.

logical, is basically redundant; but sometimes we simply want to emphasize the physical and logical nature of the dimensionality.

An example can illustrate: If a clinical data warehouse needs to store facts about drugs and implants, then these constitute two logical subdimensions of the warehouse. Note that I didn't say that they *might* constitute two subdimensions. They do, by definition. The only question is whether those two subdimensions will be included in the same physical dimension or not. Figure 2.4 illustrates those two design options. Each could be implemented as a separate physical dimension, but more likely will be consolidated into a single physical dimension (e.g., Material) containing both Drug and Implant logical subdimensions.

The distinction is somewhat arbitrary but is extremely important to the design of a particular data warehouse. The number of subdimensions needs to be appropriate to the scope of the entire warehouse and the problem domains it is meant to address. The number of dimensions is constrained by the database technology being used to implement the data warehouse. While a typical data warehouse might have 50–100 logical subdimensions, no commercially available database technology can effectively handle a fact table design with 50–100 foreign keys. We typically see complex data warehouses today being comprised of 15–30 physical dimensions, across which are spread the 50–100 logical subdimensions needed to properly represent the data. Diagrams of star schema warehouses typically include representative notations about subdimensions outside the boxes that represent the dimensions, although the notations are rarely exhaustive.

While the dimension design pattern that will be covered in Chapter 5 determines the structure of each of the various warehouse dimensions, it ultimately will say nothing about the content or semantic meaning of those dimensions. The pattern is neutral with respect to defining a grouping of concepts into the dimensions. The expected 15–20 dimensional star schema, with 3–8 subdimensions defined for each dimension, could just as easily have been 50–100 distinct dimensions without violating the design pattern. While a 20-dimensional star is surely preferable to a 100-dimensional star when designing and building a relational database, a technology bias shouldn't be the basis for the decision. The number of dimensions should be determined as naturally

as possible from the problem domain and requirements being addressed by the implementation of the warehouse. Of the permutations of dimensionality that might make sense for a particular data warehouse, one of them should be determinable as the best.

Dimensional Granularity

Once the question of physical dimensions and logical subdimensions has been settled, there remains the design question of what the appropriate level of detail, or grain, is needed in each dimension. This issue couples closely with the issue of how to dimensionalize the various facts that are within the scope of the warehouse because the grain of the dimensions often becomes a critical factor in correct modeling of facts against the chosen dimensions. In order to understand granularity, let's use location in space and time as an example. The four-dimensional star schema depicted in Figure 2.5 can be used to store facts about the four-dimensional space and time within which things on our planet exist. The perspective matters, because the three spatial dimensions define space in terms of the surface of the planet, not some other arbitrary x-y-z coordinate system. We'll see that this places some limitations on what can be stored in this warehouse.

Let's start with dimensionalizing a simple fact: The location of a hospital. We easily recognize that a building exists someplace on the planet, so can be located using longitude and latitude; but at what grain should the longitude and latitude be defined? There's some ambiguity both in the dimension, and in the fact. Dimensionally, we could store both at a grain of degrees. Factually, our hospital might cover several acres, so we have to know exactly what our fact is meant to locate. We can decide the fact first: Whenever we store the location of something, we'll think in terms of the center of that thing. If the scale of the thing (e.g., acres) is small relative to the grain we choose for the dimension (e.g., miles vs. feet), the choice of center can be shown to be reasonable.

The choice of dimension grain gets complicated because the dimension eventually has to support lots of different kinds of facts. For our hospital location fact, keeping both longitude and latitude in terms of degrees might be an option. A degree of latitude is 69.11 miles. A degree of longitude at the Equator is 69.11 miles, getting smaller as you move north or south until it drops to zero

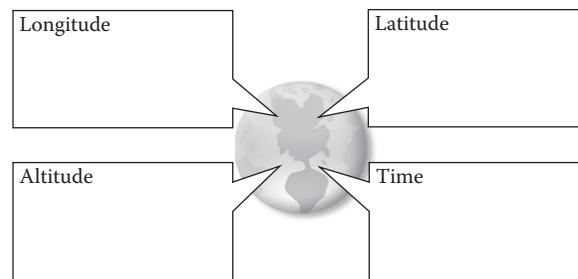


Figure 2.5 Example of four-dimensional design model.

at the poles. This relation means that the grain of a latitude value for a location will always be equal to, or larger, than the grain of the longitude value for that location. Therefore, presuming that we'll eventually choose the same grain for both of these dimensions, it's the grain of the longitude dimension that we should be concerned with when dimensionalizing our location facts. Degrees will probably not be sufficient to meet reasonable requirements. Knowing the location of a multi-acre site to only within approximately 69 miles doesn't seem like it will be that useful. Increasing the detail in the dimension to be minutes (1.15 miles each) or seconds (0.02 miles each) will give more reasonable and desirable results. At seconds of arc, the dimensions will give us the location of the center of our hospital to within 100 ft of accuracy.

As dimensional modelers, we always want to be challenging our design to ensure what we implement will meet all requirements; including those we don't yet know about. If seconds of arc gives us 100 ft accuracy, that's fine for something measured in acres; but what about smaller things? Suppose someone wants to record the location of the individual buildings on our hospital campus, or even the specific receiving bays at some of those buildings. Now we'd be talking about things much smaller than acres, sometimes things only a few yards across. Anticipating this, we might want to include enough detail in our latitude and longitude dimensions to be able to capture tenths of seconds of arc (about 10 ft) or hundredths of seconds of arc (about a foot).

Now we can dimensionalize the center of our hospital location to within a foot of longitude and latitude. Are we done? No. The schema includes two other dimensions that we haven't yet used, so we should analyze them for applicability to our fact. Does the center of our hospital location have altitude? Of course it does, and that dimensionality could be added to our fact. Other things being equal, we'd likely use a grain of feet in our altitude dimension in order to match the level of detail already available in our latitude and longitude dimensions.

Now that we've taken our fact to a third spatial dimension, we also want to revisit our designation that our fact will be based on the center of the location represented in the fact. The center made sense for a two-dimensional surface like our hospital campus, but what happens when we add altitude? For two-dimensional surfaces like our campus, the altitude can be the actual altitude on earth of that center point. For three-dimensional objects like buildings, the altitude is now a vertical line segment above that two-dimensional center point. How might we limit it? I suggest using the bottom of the object in question, or its lowest altitude. That means that the altitude of a building is the altitude of the earth at that point. The altitude of the first floor is the same as for the building, but the altitude of all other floors is the height of each floor's surface. In populating the dimension, altitude typically has two forms of representation: feet above sea level, and feet above ground. The former is typically used for geographic data, while the latter is typically used for structural data. As long as the former is available for the ground surface at each longitude and latitude, the latter measure is easily converted into the former.

Finally, what about time? Does our hospital location fact need time in its dimensionality? It probably does. While the geographic location on which we find our hospital is a static location, there tends to be some dynamics to the location of things at any geographic location. Our hospital campus hasn't always existed in its current form. Twenty years ago we might have been downtown, and now we're on a bigger piece of land near the edge of the city. In this longer-term perspective, time matters. Even in our current campus, if we've acquired edge properties over the years, the center point of the campus has been moving; and as we conjecture on the location of our buildings and smaller physical assets, many of those come and go on even shorter timescales. So, let's include time in our fact dimensionalizing; using the time at which the fact became true as our dimensionalization point.

As with the spatial dimensions, we need to decide upon an appropriate level of detail for our time dimension. It's natural to include all of the elements of timescale that we normally encounter in data. At the higher grain, we capture what our calendar would tell us: Year, Month, and Day. As we add more detail, we pick up what our clocks would tell us: Hour, Minute, Second. For our hospital location facts, we'd probably be happy knowing the day that the fact became true. We'd include the date and time detail in our discussion of the dimension primarily because we know there will be many facts in our efforts that will need more than just the date.

So, we've now designed our dimensional model to a sufficient grain to store our hospital location facts at the level of detail we've determined would maximize its value: latitude, longitude, and altitude to the nearest foot, and time to the day. This dimensional design will give us the capability to store those facts; but it doesn't mean we'll actually store them that way. Storing facts depends on more than our capability in the dimensional model. It also depends upon our ability to actually source the data. The data source might not have the necessary precision to take advantage of all of the dimensional capabilities we've modeled. Suppose we get a dataset that has a location for all of our facilities, but that dataset only has the longitude and latitude to the minute of arc, doesn't include any altitude, and only includes a cornerstone year. We'll be able to store those facts, but they won't be as informative as what we'd be able to store with a more precise dataset. That's okay. Our data warehouse can't store data we don't have. As dimensional modelers, we want to ensure that our model has the capability to store what we do have, and what we're likely get in the future if each dataset continues to improve.

The process of designing dimensions for a star schema data warehouse works much the same as this regardless of the domain of interest. As a complex domain, biomedical data present many interesting challenges when our analysis tries to anticipate the dimensions that will be needed to store the myriad data types and sources typically encountered in healthcare practice and biomedical research. Chapter 4 presents my standard dimensional model for biomedical data. First let's review the types of issues and questions that we

typically want to explore when coming up with this kind of model. Basically, where do the dimensions come from?

Transposing Dimensional Schema

The data warehouse designer must determine the optimum number of physical dimensions and logical subdimensions that will be included in a design. The number of physical dimensions is a *performance* concern, and the number of logical subdimensions is a *domain* concern. Fortunately, there are algorithmic mechanisms for changing the number of physical dimensions in a warehouse design so the logical view of the data remains unchanged. As a result, these two concerns can be treated separately on most initiatives.

Early data warehouse designs of low dimensionality can be changed to become higher-dimensional warehouses without losing the domain perspective supported by the logical subdimensions contained in those physical dimensions. This provides a pathway (see [Figure 2.6](#)) from early implementations to later implementations as the warehouse domain expands over time and database technologies improve to allow better performance at higher dimensionality. A future-oriented expanding pathway removes the pressure on the designer to get the first-generation warehouse design perfect.

[Figure 2.6](#) illustrates the pathway from lowest to higher dimensionality. Each warehouse will vary along the continuum from one extremely generalized dimension that can define and store any dimensional reference, to a collection of very specific dimensions that each store definitions that share semantic context and meanings.

The warehouse at the extreme abstraction end of the scale would be a warehouse with only a generalized single dimension. In my design, I call this dimension the Annotation dimension because it allows me to annotate my facts with anything I want. It is sometimes referred to as the *Thing* dimension because we can make it about anything (and Thing will actually show up as a sensible abstraction when we get to semantic ontologies in Chapter 3). What is extreme here isn't that we have an Annotation dimension, but that we have *only* an Annotation dimension. Absolutely everything we want to say about our facts would have to be an entry in this one dimension.

If we want to record that a patient was administered a 5 mg dose of a specific drug during a specific clinic visit by a specific physician, then the dimension would have to include a row for the actual permutation of that patient, visit, physician, and drug. If we think about all of the things we might want to store in our data warehouse, the idea of a single dimension having to include all of the reasonable permutations of all of the things that we might want to keep facts about becomes ridiculous. We simply can't do it. The list of things we're interested in is already quite long, and the length of the list of permutations of those things is astronomically long. The single generalized

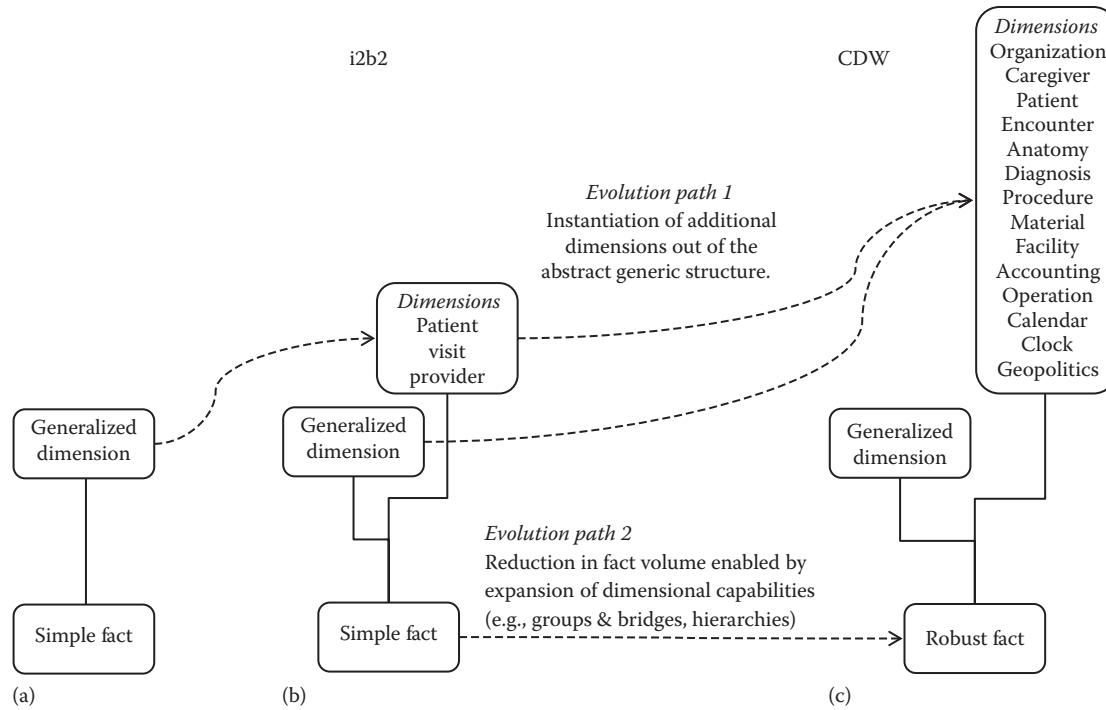


Figure 2.6 Dimension Maturity Pathway: (a) one-dimensional warehouse with all logical dimensions generalized into the single physical dimension, (b) the i2b2 design, with three specific logical dimensions moved into their own physical dimensions, and (c) a clinical warehouse with 14 physical dimensions, and everything else left in the generalized dimension. The generalized dimension is always present, and embodies any logical dimensions that don't map into one of the defined physical dimensions. The number of logical dimensions remains constant.

dimensional model at the left end of the continuum is simply too extreme to constitute an effective design alternative for our data warehouse.

To move to the right on the warehouse continuum, we look for elements in the generalized dimension that are most driving the large number of permutations. We might note that it seems reasonable to have a list of drugs in our generalized dimension, but it isn't reasonable to have the permutations of those drugs for every patient, visit, and provider in the dimension. We might create separate independent dimensions for those things, and thus reduce the permutation complexity of what remains in the generalized dimensions. Our patient-visit-provider-drug permutations list would be reduced to just a drug list. The facts we would store in this new model would expand from being dimensionalized to just the general Annotation dimension to being dimensionalized against all four dimensions: Patient, Visit, Provider, and Annotation. For our drug administration fact, the Annotation only needs to serve as the drug list.

As we analyze the original generalized Annotation dimension to find new dimensions that can be extracted and made independent, we might find any number of common data classes to discuss, and we'll discuss many more before we're finished. I chose these four for a very specific reason: A data warehouse

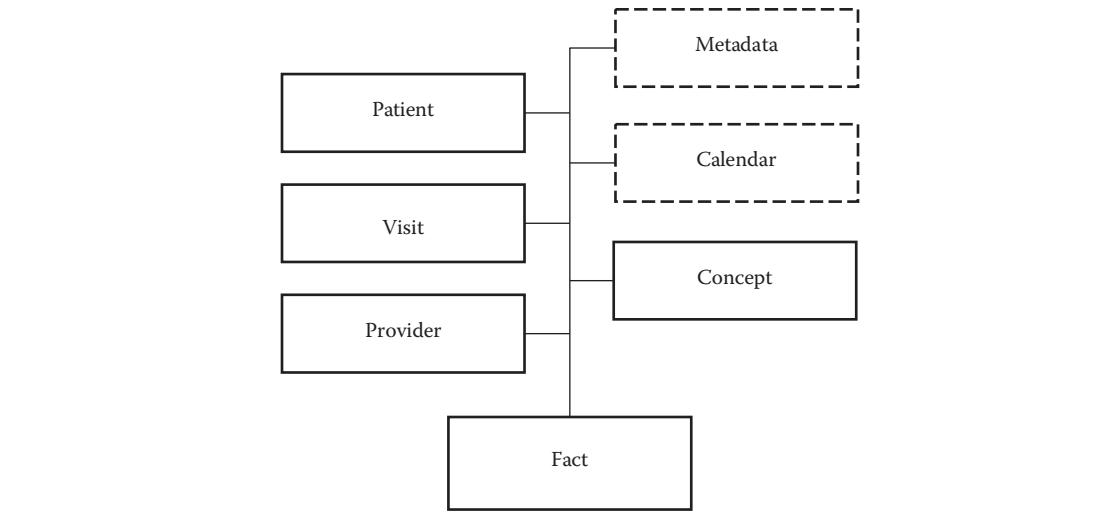


Figure 2.7 Dimensionality of i2b2 data warehouse, four explicit dimensions, and two implicit dimensions “hidden” in the i2b2 fact table.

that encompasses these four dimensions is readily available as an open source solution. The Informatics for Integrating Biology and the Bedside (i2b2) warehouse in Figure 2.6b is an open source warehouse design that specifically instantiates only these few dimensions commonly seen in a large-scale biomedical data warehouse. These four dimensions are illustrated in Figure 2.7, along with two other relatively hidden dimensions in the i2b2 model. By hidden, I mean that the four main dimensions are those that are usually depicted in documents that describe the i2b2 structure. Also, the Calendar and Metadata dimensions don't have their own sets of master tables as we typically expect to see for a dimension in a data warehouse.

Both of these hidden dimensions are implemented as columns in the key of the i2b2 fact table. The Calendar dimension is implemented as a date column, which means that facts associated with temporal periods other than date will be a challenge to represent depending upon the date processing capabilities of the database manager on which the warehouse is implemented. What I'm calling the Metadata dimension is actually defined in i2b2 as a Modifier column in the fact table. This Metadata allows a fact row to better identify exactly what is being represented by the fact (hence metadata as a description of the data), allowing multiple modified facts to be instantiated against the same five-dimensional set of values in the Patient, Visit, Provider, Calendar, and Concept dimensions. Without the Metadata dimension, the Concept dimension would need to be exploded one more time to create permutations among the values already in the Concept dimension and the values being placed in the Modifier column.

Alternatively, the design of the fact table could be altered from having just one generic fact value column to having multiple fact value columns, one for every instance of a modifier value. The alternative of having a fact table with multiple columns versus only a single generic value column is a key design decision in

any dimensional data warehouse. The multiple column option is referred to as a *wide* fact table, and the single column option is referred to as a *narrow* fact table. The i2b2 design is based on the narrow fact table design, as is the entire warehouse that this book outlines. Whenever you have a fact table with a generic fact value column, the metadata for what is actually placed in a given instance of that fact value has to be stored somewhere. That's what the Metadata dimension (under whatever name it has been given) is for. It allows lots of different kinds of facts to ultimately reside in the same column in the database in a way that those values can be used effectively.

The clinical warehouse design to the right in Figure 2.6c illustrates a typical warehouse star design in which many more dimensions have been instantiated. Viewed correctly, these can all be seen as shared designs along a continuum, and not conflicting alternative designs. The differences are in the number and types of dimensions that are instantiated out of the generic dimension. The challenge is to instantiate the right dimensions correctly. An algorithm for moving data among data marts of different dimensionality is described in Chapter 17.

Anticipating Dimensions

Having built a rationale for developing our data warehouse using the dimensional star schema paradigm, we will be anxious to begin anticipating and defining the various dimensions that will comprise our star model. The notion that we will be able to algorithmically take star designs of different dimensionality and transform them into each other offers some assurance that our initial set of dimensions doesn't have to perfectly reflect our ultimate long-term future. However, we don't want to start out over-relying on that risk mitigation assurance. While the risk of migrating star models can be made small in a technical sense, the risk of confusing our user analyst community through versioned transformations remains high. Our users will heavily invest themselves in learning the star model we develop, so we want the initial star to be as close to their long-term set of requirements as possible.

There are a variety of requirements elicitation techniques that are commonly used by data analysts and designers when defining new database structures. The ones used most frequently to define data warehouse dimensions are affinity analysis, supertype–subtype modeling, and reverse engineering. These techniques, while useful, share a common pitfall of being oriented toward information systems artifacts rather than the real-world artifacts represented in those systems. A danger of basing the design on information systems artifacts is that any weaknesses in those existing artifacts is likely to be translated into the new data warehouse. New analytical techniques related to semantic ontologies are becoming a popular and powerful way to represent requirements for data warehousing. Chapter 3 is devoted to semantic modeling of data, but first let's review some of those more traditional modeling techniques.

Affinity Analysis

If we designed a one-dimensional warehouse, that dimension would need to list all of the necessary concepts; however, the dimensionality would be insufficient to take advantage of the star schema dimensional model espoused here. A simple affinity principle could be used to split the one dimension into many dimensions: Divide up the list of concepts into affinity groupings so the concepts within each resulting dimension share more in common with each other than any of them do with the concepts placed into the other affinity groupings. This process will guarantee a multi-dimensional warehouse but will exhibit high variability in the number of dimensions depending upon the conceptual sensitivity used in the affinity analysis. Figure 2.8 illustrates some clusters of clinically related data that might be identified through some form of research or brainstorming during the very early stages of a data warehouse development project. Each cluster represents a potential perspective against which a user might want to analyze or aggregate data in the data warehouse, so each is a potential warehouse dimension.

Because the number of clusters quickly exceeds the number of physical dimensions anticipated for the warehouse, each cluster is considered to be a logical subdimension of one or more physical dimensions that now need to be identified. My own experience has led me to a small collection of heuristics that I find helpful in the early stages of trying to identify dimensions for the warehouse:

- Clusters of concepts that represent systems of some form will typically reside in different physical dimensions if they represent systems of different scale: societal, organismic, or mechanical. For example, societal systems like

Diseases	Case workers	Addresses	Buildings
Lab tests	Departments	Hospitals	Charges
Beds	Countries	Surgeries	Adjustments
Allergens	Patients	Pathologies	Implants
States	Physicians	Cath lab	Units
Credits	Rooms	Cities	Payments
Symptoms	Specimen	Research cohorts	Bills
Drugs	Transporters	Vital signs	Radiology exams
Nurses	Implants	Supplies	Group practices

Figure 2.8 Examples of logical groupings that might be identified as candidate warehouse dimensions during a brainstorming of project-relevant data.

hospitals, group practices, and their associated *departments* can be grouped together as Organizations, while organismic systems like *patients, physicians*, and *case workers* belong in a different dimension.

- Different clusters that focus on people, places, or things don't typically resolve into the same physical dimension. As a result, *patients, countries*, and *drugs* would not be expected to end up as subdimensions of the same dimension.
- Clusters that represent internal and controllable data don't typically map into the same dimension as external data that might be very noisy and out of our control. For this reason, *physicians*, which we can control, and *patients*, which are typically noisy and beyond our control, aren't typically mapped to the same dimension even though both are clusters of people.
- Clusters representing physical things aren't usually mapped into the same dimensions as logical or conceptual things. For this reason, we might group physical objects like *rooms, beds*, or *devices*; but it's difficult to also include conceptual constructs like *accounts* or *insurance plans* in the same dimension.
- Clusters that represent things that we do (typically *verbs*) are usually mapped into a separate dimension from things that we have (typically *nouns*). For this reason, processes like *lab tests, radiology exams*, and *surgeries* typically end up in different dimensions than physical objects like *drugs, implants*, or *supplies*.
- Clusters that represent attributes that describe things of interest, like *diagnoses*, typically end up in different dimensions than things that are used or needed for things of interest, like *rooms* and *beds*.

These heuristics are always subject to interpretation, and they won't always produce the same result when practiced by different warehouse development teams; but they go a long way toward reducing the high levels of variability in design that can be seen in dimension identification done using more *ad hoc* analysis. An example of a dimensional design that might result from these heuristics for the clusters in [Figure 2.8](#) can be seen in [Figure 2.9](#). In this design alternative, some of the heuristics listed above have been used to combine the logical subdimensions into candidate physical dimensions. The geopolitical clusters for countries, states, cities, and addresses have been combined into a Geopolitics dimension. People who provide care are combined into a Caregiver dimension. The clinical diagnoses, pathologies, problems, and symptoms are all combined in a single Diagnosis dimension. The design process continues with the rest of the candidate dimensions.

Dimensions as Supertypes

Designers familiar with data normalization will feel comfortable with some of these choices because the heuristics mimic some of the rules enforced by those practices, particularly the relegating of certain subordinate *subtype*

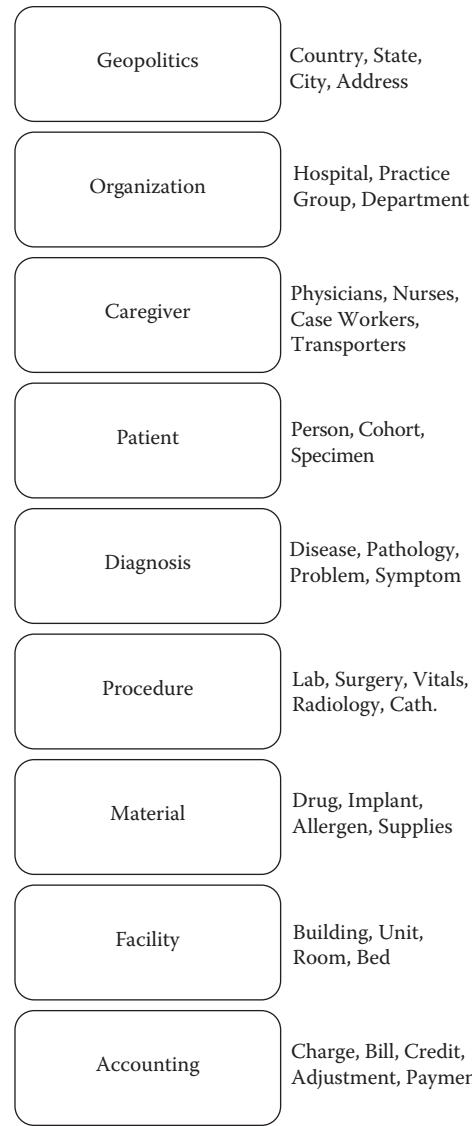


Figure 2.9 Examples of physical dimensions with their associated logical subdimensions that might result from an affinity-based and heuristic analysis.

entities to their encompassing *supertype* entities. When the characteristics of the various subdimensions look very similar, we typically don't resist combining those things into a single dimension. For example, all of the caregivers have names, professional specialties, and phone numbers. Normalization would lead these to be defined as subtypes of a common supertype entity, which is highly analogous to our physical dimension (supertype) and logical subdimension (subtype) distinction here. However, there are examples in Figure 2.9 that seem to be direct contradictions of normalization principles, and they will be problematic to designers whose mindset is relational database design rather than star schema design. While normalization experience can be helpful in this process, it is not our goal that our star schema be completely normalized.

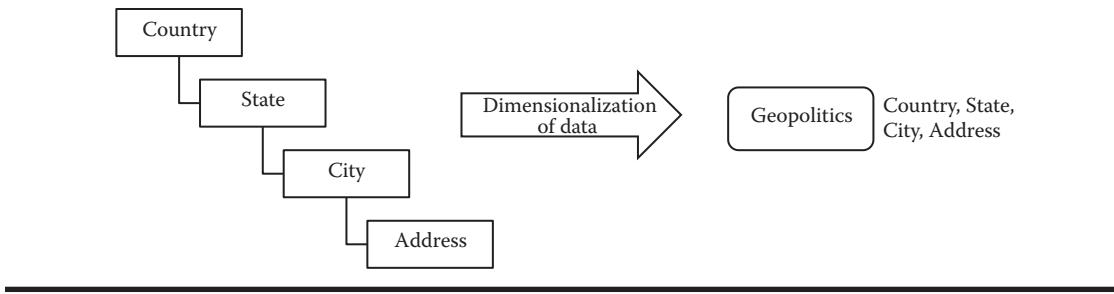


Figure 2.10 Normalization view of dimensionalized subdimensions.

A simple normalization challenge is Geopolitics. A relational designer will question how a single dimension table can define everything from a country down to an address, particularly since the more granular subtypes require foreign keys to the higher-level subtypes. Address is dependent on city, which is dependent on state, which is dependent on country. Normalization would require those subtypes to be implemented separately, as depicted in [Figure 2.10](#). What might have been implemented as four separate tables in a normalized relational database are now implemented in the single Geopolitics definition table as four different subdimensions. The relationships among the different constructs will be handled as entries in the Geopolitics dimension itself rather than as foreign keys in the database among the normalized tables.

Combining subtypes into a single supertype table during database design is not new. Indeed, it is a traditional part of the database design process. It's why we refer to a *logical data model* as an engineering deliverable distinct from the *logical database model* deliverable. Dimensions and subdimensions are part of the logical *database* model for our warehouse, and we use those constructs to implement the subtype and supertype entity relationships that we've defined in our logical *data* models.

Geopolitical entities share many data commonalities, with scale being a key differentiator. In addition, a main use for the distinctions among the subtypes is precisely the hierachic aggregation that is supported so well by treating them as a single warehouse dimension that can be aggregated. If we want to store a census count in our warehouse, that fact will relate to the Geopolitics dimension in the same way whether the census is a local or national or intermediate value. The power of placing subdimensions together in dimensions is in our ability to connect facts to the dimension without regard to the distinctions among the subtypes defined in our logical data models. This requirement causes us to promote data normalization strategies differently than if we were designing a relational database model for a relational warehouse.

That power becomes more evident when we look at examples that are less obvious in their combinatorial nature. The Patient dimension in [Figure 2.9](#) includes subtypes that are less obviously related to a single supertype definition. The most obvious example of a Patient dimension entry would be the Person subdimension, what we typically call the Patient. Most of the facts in our warehouse will be facts

about a single Person as Patient, but our Patient dimension could include two other subdimensions that have larger and smaller grains.

At a larger grain, groups of people form cohorts, often for research studies or simply for analysis purposes. Any fact we might want to store about a person might have been stored, perhaps as a median value, for a *cohort* of patients. We want to be able to store our facts regardless of the grain of the patient, so we make both cohorts and people subdimensions of the same dimension. Likewise, at a smaller grain, those clinical results might have been captured about a specimen drawn from a person. Since we want our facts to be able to be defined against our generic dimensions regardless of that grain, we put Specimen into our Patient dimension also.

The combination of subtypes into supertype structures, of subdimensions into dimensions, has been part of database design for as long as there have been databases. While the choices are supported by technical considerations and theory, it is affinity analysis among the constructs that has always helped determine, or even driven, that discussion. Consolidation of data into common structures always results in the combined data having more in common with its counterparts than with the data it wasn't combined with. Combined elements share an affinity. The main thing that has changed more recently is the set of heuristics that we use to optimize that process, and evaluate the resulting designs as the needs of traditional relational databases have evolved toward the requirements for star schema data warehouses.

The affinity approach is, therefore, the technique most used in data warehouse design today. It tends to take an information technology bias because the systems and databases available for loading into the warehouse are often the best known artifacts for identifying and describing what is intended for loading into the warehouse upon completion. Within information technology, affinity analysis is analogous to data model normalization. Identifying dimensions and subdimensions is like trying to identify all of the entity types and subtypes that are needed to model a problem domain.

Reverse Engineering

Many data warehouse design efforts conduct this affinity analysis while looking directly at the data files and databases that constitute their source applications, because those applications lack any form of logical data model documentation. The entity types and subtypes are inferred from the implemented artifacts themselves. The result is that the design of data warehouse dimensionality comes to be based on the key structures and data dependencies designed into a few key source systems rather than on the semantics and business rules of the broader problem domain. This is adequate if the identified sources and models are of high quality, and if they truly represent the problem domain. If they don't, then the first generation data warehouse built from them will typically fail to stand the test of time.

The combination of dimensions works well for the facts associated with early warehouse use because they were derived by looking directly at the data sources that would be used to populate the earlier versions of the warehouse. With the passage of time—usually measured in *years*—the need to add, change, split, or combine dimensions will place burdens on the warehouse design, the support team, the governance group, and the users who want to query data from that design. Despite these weaknesses, this approach has dominated warehouse design because group consensus can typically be achieved in the affinity analysis steps, and an effective alternative has not been available.

The underlying problem is that warehouse dimensions are being discussed and designed only in the context of the facts expected to be stored. This is a backward approach because it is the dimensions that are meant to provide the various contexts in which the stored facts will make semantic sense. It is circular to define those contexts by looking at the facts. The relatively quick progress that is made by reverse engineering database artifacts or logical data models into lists of dimensions might seem satisfying, and will be more than adequate for the first round of data, but it is insufficient for designing and building a true enterprise-wide data warehouse for the long term. The challenge is to identify, model, and design dimensions without direct reference to the facts or to the source systems that will provide those facts. The dimensions should be designed independently, and then facts can be mapped into those dimensions as they arrive.

Chapter 3 presents an approach to understanding source data that is based on forward engineering from the real world rather than on reverse engineering from existing systems. Understanding what exists and how those things relate to each other is an *ontological* problem, and the definition of dimensions for a data warehouse is a specific instance of such an ontological problem domain. To an ontologist, an ontology is a *product* created while exploring a conceptual problem domain. It is a formalized list of concepts, and the relationships among those concepts, that clarifies and defines the conceptual space under study. An ontology is defined independently of the application domains in which it will be referenced and used, and thereby forms an effective paradigm for data warehouse dimension design. Ontologies can help define the contents of each dimension of the warehouse, and when done properly, each dimension can become an *applied* ontology to be used in annotating facts.

Chapter 3

Understanding Source Data

Having prepared ourselves to jump into the analytical process of defining the dimensions of a biomedical data warehouse in the last chapter, it's understandable that we would want to next jump into that process. If we jump directly from a discussion of dimensional modeling into the design of our data warehouse, we'll be treating the database itself as far too central to our effort. Instead, it's important to take a side trip into understanding our source data from which we'll eventually build our data warehouse. We don't want our warehouse to feel like an information technology system. Instead, we want it to have a biomedical, clinical, or healthcare feel to it. To create that feel, we need to understand our data in more and different ways than we typically do in traditional projects. We don't want to see the data in our systems; we want to see our world in our data.

Chapters 4 and 5 will provide a view into the general framework and standard design patterns that are available for the development of strong and vibrant biomedical data warehouses. Those models are intended to illustrate that building a data warehouse is an increasingly standardized activity. However, it is not my intention to claim that these design patterns make data warehousing easy (in the *verb* sense of data warehousing discussed at the opening of Chapter 1). While it is increasingly true that the information technology aspect of building a data warehouse can be done in very standard, productive, and effective ways; the complexity of the actual biomedical data to be stored in a data warehouse prevents us from having easy paths through the source data. We'll build the data warehouse fairly quickly, but the success of that implementation will rest upon how well we come to understand the source data we'll be loading into it.

On its surface, biomedical data are complex because there's a lot of it, it is typically collected in diverse ways, its meaning can vary across disciplines and specialties, and we don't always have access to all of the data we want. In that sense, biomedical data shares a lot in common with any complex industry or economic sector. I would argue that the biomedical sector is among the most complex, and is certainly the most complex that I have dealt with in my career. This chapter deals with understanding that complexity well enough to be able to

effectively load source biomedical data into our emerging data warehouse. These heuristics will apply to any highly complex domain, but here we're specifically interested in understanding source biomedical data.

In addition to the scale issues already implied for biomedicine, I see three problematic dimensions that we'll have to be able to address to effectively understand and load our source data in biomedicine:

1. *Implicitness of most data in IT systems*: Much of the data we want to source into our warehouse are actually metadata about the source systems rather than actual data in those systems, and even more data are tacitly embedded in the knowledge or awareness of the users of those systems. If we want the data to make sense outside of the context of those source systems, any implicit data must be made explicit in the warehouse.
2. *Differences in semantic levels represented in our data processes*: We talk as though our databases contain patient information, and that works well for our human brains that show a remarkable resiliency for simultaneously sorting out multiple contexts. In reality though, our medical records aren't even directly *about* patients. They are *about* the clinical contexts in which providers interacted with patients. Those contexts are *about* the patients, and they need to be kept semantically distinct if we want to be able to automate any formal reasoning against our biomedical data.
3. *Epistemological differences across our domain and data*: Not all biomedical data are true data in the same way. Some of our data are observational, representing the real world and how we'd like to think all of our data exists. Other data are conclusions based on the observational data, making it inaccurate over time as observations change, or the knowledge we apply to observations changes. Other data are hypotheses, yet to be explored or confirmed by other observations or conclusions. Differences in knowledge represented by these data types will affect our ability to draw conclusions correctly, particularly in queries that cross these types.

By addressing these issues, we'll design and load data into our warehouses that can be used by people in the biomedical sector to develop and answer real questions that solve real problems. Loading data is easy, but loading it correctly is a real challenge. Let's take a look at these three challenges in more detail.

Implicit versus Explicit Data

We think of the contents of our databases and files as the data that we eventually want in our data warehouse. While that's true, it's only the start. Data inside these different kinds of structures are explicit. We can see it, extract it, manipulate it, and load it into the warehouse. The problem with that perception is that it gives us a very incomplete picture of the work that actually has to be

done to get the data into the warehouse. A focus on the explicit data causes us to concentrate our efforts on the technical challenges of data connectivity and interoperability. It's good to get the technical aspects right, but the failure of a data warehouse initiative is very rarely as a result of doing those things poorly.

What we know about data in any application system database is typically a combination of explicit data and implicit information about that data, or metadata. Most source data in our environment are actually *implicit*. It's buried in the metadata that describes our data structures, and much of that metadata are never actually documented anywhere. A lot of the metadata we need are stored in the minds and actions of the people who create and use the data.

The actual application systems we want to get data from are a form of metadata. For example, where in the data stored in an Electronic Health Record (EHR) system are the data that tell us that the data are from an EHR, or from a particular EHR? Those data don't exist. It's metadata stored in our minds. If we were to compare some basic health transactions received from an EHR to functionally similar transactions received from a patient portal's Personal Health Record (PHR), we wouldn't necessarily be able to tell them apart. In our data warehouse, we certainly want to be able to tell them apart: The implicit notion that a given piece of data is somehow associated with a particular source system, or a particular source organization, must be made into explicit data as it is brought to the warehouse.

We might not need to do this if we were simply querying a bunch of source systems. I could write a query that joins data in our EMR and portal PHR systems to produce a desired result. I know to do this because the requisite metadata is in my head. I know I want to join the data across these two sources, but in my future data warehouse, I want all of that data to be in my warehouse. To conduct the same analysis in the warehouse that I could do on my own against the two sources, I'll need to be able to tell which data are from which source. The metadata that is the name or domain of the source system is relevant metadata for our warehouse, even though it is never instantiated otherwise in our database environment prior to building the data warehouse.

Within these different source systems, the actual names of our databases, tables, and columns are also metadata. The Vital Signs table might have a column called Systolic Blood Pressure. Another column might be called Tympanic Temperature. This metadata is implicit. We're told that these two particular measures are being stored in this table, and as users of the system we're also likely to be familiar with the vital signs clinical event that might record them. Since those names won't necessarily appear in our warehouse design, we need to make those values into explicit data available for our loads. The kind of data warehouse we're looking to build won't have column names that match every column of every source system. The point of our star schema dimensional model is to make our data much more generic than that.

Even the explicit data known to our database management systems aren't all of the metadata we actually need to load our data warehouse. Nowhere in

the explicit metadata of the system can we see that blood pressure is stored in millimeters of mercury (mmHg) and that temperature is being recorded in degrees Fahrenheit. That information is completely implicit metadata. The database manager knows the name of the column as explicit metadata, but it doesn't know the units of measure that users have in mind when they populate values in that column. In fact, another institution using the exact same system might be recording temperatures in degrees Celsius in the exact same system column. How will our warehouse know the values in a data column named Tympanic Temperature should be interpreted as being degrees Celsius? How will it know the text in a column called Observational Note is in English?

In a small percentage of cases, the unit of measure is explicit in the data structure (e.g., lab results); but in the vast majority of cases, the unit of measure is simply implicit. Our EHRs have been designed over the years to present data to the human eye in either reports or on display screens. No provider would be confused seeing an oral temperature display that contains both Celsius and Fahrenheit values, but the warehouse needs to know. That implicit data must be made explicit in order to be properly recorded and interpreted within the data warehouse. Done poorly, the data warehouse will fail.

The jobs we create to extract data from source systems will pull a lot of data stored in the data structures in those systems. My experience is that the data probably comprise no more than half of what a typical source extract needs to instantiate for a data warehouse load. In addition, we'll want to extract data that might be known to the database technology in those systems as metadata (e.g., column names, data types). Our extract routines will make some of that explicit metadata into actual data for our warehouse. Additionally, our extract routines will need to make additional implicit metadata (e.g., the stuff in our heads) into explicit data as well. Most of that data will be discerned during the analysis process and will be coded directly into our warehouse extract and load jobs. The challenge for warehouse analysts is to work to ensure that the instantiation of the metadata serves the purpose of semantic understanding. We don't want to load data into the warehouse according to syntax rules inferred from source databases. We want to load data, the metadata of which describes true semantic content, as it was intended to be used. We want what the data means, not just how they were stored.

Semantic Layers

Even after making all of our data explicit through definition and capture of implicit elements of the metadata associated with our desired sources, we'll still need to deal with the differences in meaning that different stakeholders will associate with the data we choose to load into the warehouse. To consistently understand the meaning of all of the myriad data that we will be loading into our data warehouse, both now and into the distant future, we need an

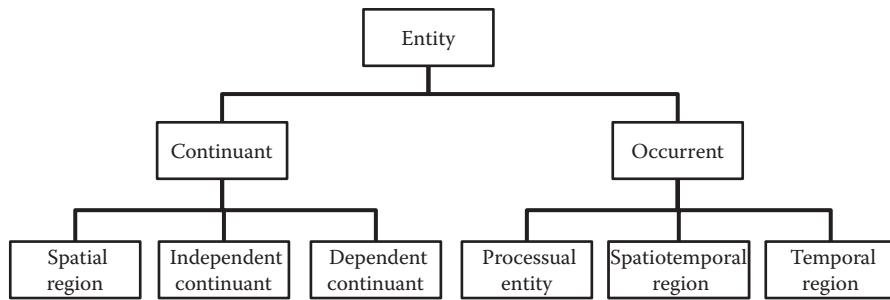


Figure 3.1 Basic Formal Ontology (BFO), high-level categorization.

anchor against which all of our data can be modeled and understood. For this, I look to the community of informatics professionals that define and build a variety of tools that as a class are referred to as semantic ontologies. Among these, the first to consider is the Basic Formal Ontology (BFO) (Figure 3.1) that will serve as that single common anchor for all of our semantic analysis and understanding.

The BFO supports the definition and classification of any data source we choose to analyze and load into the data warehouse. It treats every construct against which we want to define or store data as an entity, and begins with a central bifurcation of the things about which we want to store data into continuants and occurrents. BFO continuants are those things that exist and persist through time. BFO occurrents are those things that occur in time, having a start and end, and often passing through phases in time. I think of continuants as my data *nouns*, and occurrents as my data *verbs*. An ontological purist might cringe at such an over-generalization, but I find it useful as I try to apply this noun-vs.-verb distinction in my analysis work.

As we identify and analyze data elements for data sourcing, we'll be continually trying to recognize them as either describing continuants or occurrents. We'll use the presence or absence of each to help us identify missing data, or to seek additional sources to close gaps in any emerging source data. The BFO supports our thinking about our continuants (nouns) and occurrents (verbs) in very systematic ways. If we see data that seems to suggest that a continuant is changing, we'll look for the occurrent in which the continuant participates to result in that change. If we see data that seem to suggest an occurrent is happening, we'll look for the continuants that would participate or be impacted by the occurrent. This set of feedback loops will guide us toward more complete and robust datasets for our warehouse.

Beyond this high-level categorization, we will also be able to use further details from the BFO to understand the meaning of our source data. Things that are continuants and occurrents will not be so in the same ways. The BFO provides further detailed perspectives within which we can model our source data. Differences among these perspectives will be determinant over whether source data end up in our dimensions or fact tables.

BFO Continuants

The BFO breaks continuants into three significant categories: Spatial Regions, Independent Continuants, and Dependent Continuants (Figure 3.2). Independent and dependent continuants form the backbone of most of our dimensional data, but the spatial region constructs serve as the zero-, one-, two-, or three-dimensional canvas on which other data are painted. To the extent that any of our warehouse data needs to be anchored in space, we'll use data elements that map as subtypes of these spatial regions.

Spatial regions are everywhere, and everywhere is a spatial region; so all of our continuants and occurrents ultimately take place within spatial contexts even if our source data systems don't track data back that far. Examples of spatial regions include the two-dimensional surface of the Earth that provides the foundation for the geopolitics dimension of our data warehouse, or the three-dimensional volume of a hospital or medical center. Not all data sources will explicitly tie data back to geopolitical constructs, but our data analysis should attempt to find the connections in order to assure high levels of data consistency. It's a critical distinction in our data modeling, but one that isn't always at the forefront of our minds as we analyze source data.

As analysts, we'll spend much of our time looking for independent continuants. The continuants exist in the world whether we source data about them or not—they are *independent* of our knowledge of them. Most of the independent continuants we'll model in our data are BFO material entities like people, equipment, and facilities. Most material entities are objects at different scales. A patient's body is an object, and so is a patient's heart. The heart being part of the body is a relationship between these two objects that is not covered by the BFO. That relationship will be covered by another ontology (e.g., the Foundational Model of Anatomy or FMA) that itself is mapped back to the BFO we're using. The BFO also includes Fiat Object Parts—parts of objects that are defined by definition or mandate rather than some natural physical demarcation. The heart is an object, and not a fiat object part, because it is a physically distinct component that can be demarcated without ambiguity. No fiat agreement is needed. The tummy is a fiat object part precisely because children, parents, and pediatricians generally agree where this body part is by fiat, allowing a certain ambiguity over exactly what physical objects are included.

Material entities also include aggregates of other objects that lack physical associations that we would use to define simply more expansive or larger objects. Examples of aggregates might include the population of patients in a hospital, or the faculty of a medical school. These aggregates include physical objects but lack physicality beyond their largest component objects. The distinction between object aggregates and the objects of which they are composed will be important in analyzing data being sourced into the biomedical warehouses. For example, a patient is clearly an object. Parts of the patient, like a lung, heart, or arm, are clearly objects. The parts help make up the whole, in a physical

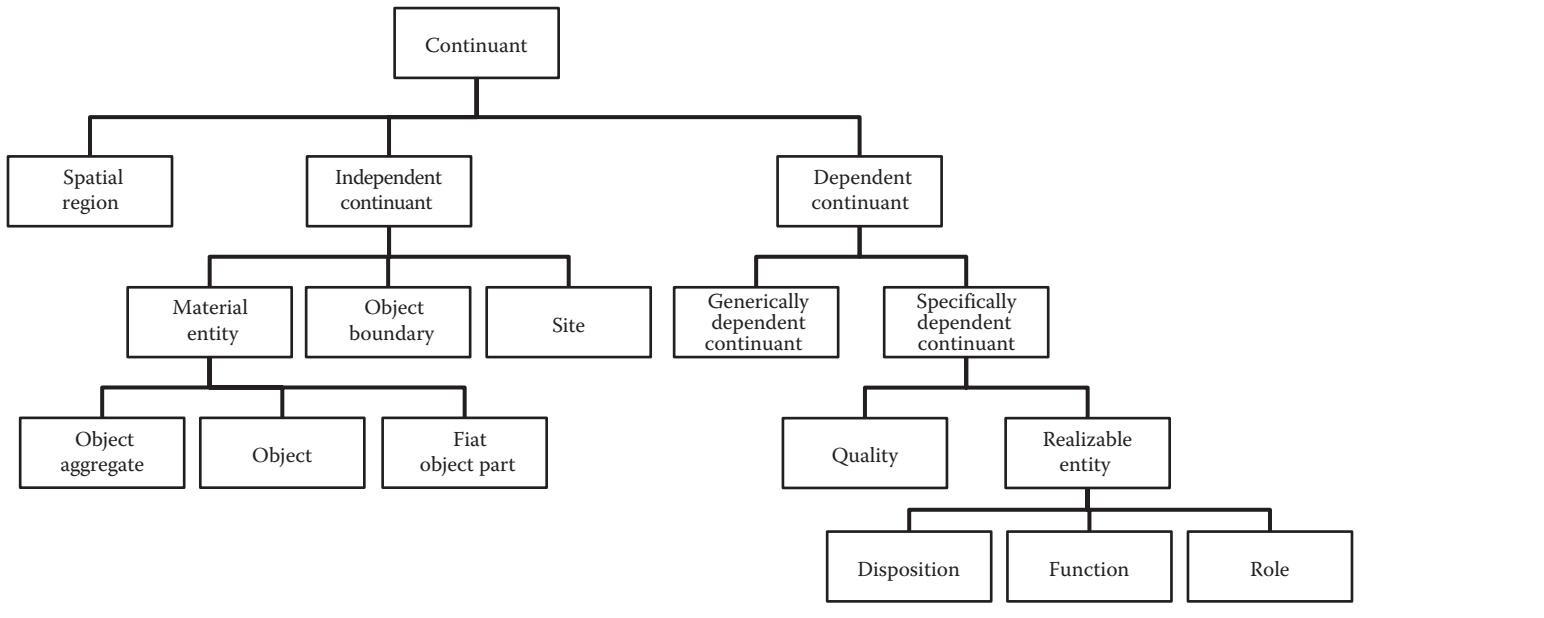


Figure 3.2 Basic Formal Ontology (BFO), continuant details.

sense that anyone would acknowledge, but what about the bacteria in a patient's gastrointestinal tract or the viruses throughout the patient's body? Even though they share certain physicality with the patient's body, that relationship is not the same as to the patient's heart or lungs. The BFO allows us to think of the patient as an object, and also as the dominant object in a BFO Object Aggregate that also includes bacteria, viruses, and additional foreign objects (e.g., a bullet in the shoulder, a swallowed coin). Most biomedical work involves the patient as an object aggregate, not just as an object.

Beyond material entities, independent continuants can include BFO Object Boundaries and BFO Sites. Object boundaries include surfaces like the surface of the skin (as opposed to the skin itself which, being composed of layers of tissue, would be an object) or the surface of the Earth. A BFO site is an independent continuant that focuses on some other continuant in some way. For example, a patient's mailing address is a site focusing on some particular region of the surface of the earth, in itself a two-dimensional spatial region. A laceration is a site that is located at some particular location on the surface of a patient's body.

It'll be quite a challenge while analyzing data sources to correctly map the data we want to source to the various forms of independent continuants defined within the BFO. Our data can be sourced without these mappings, but the consistency and accuracy of our data sourcing can be improved by systematically applying these BFO concepts to our source data analysis. The analysis conducted on these independent continuants will largely determine the definition of the dimensions and subdimensions of our star schema warehouse. The rows of our dimensions can all be mapped back to independent continuants in the BFO. The properties that describe those dimensional constructs will end up as dimensional attributes or facts depending upon their relationships to the independent continuants, making those properties *dependent* continuants in the BFO.

For modeling data in our warehouse, we are most interested in dependent continuants that are *specifically* dependent upon the independent continuants we've already been modeling. Among those dependencies are the BFO qualities of those continuants. A patient (an independent continuant) has a variety of qualities, including name, height, weight, and blood type that are dependent upon the specific patient being described. Those qualities exist only to the extent that the specific patient exists, so they are described by the BFO as Specifically Dependent Continuants. Whether the continuants will be placed in the warehouse as columns of the dimension that describes patient or as facts in our fact tables will be based on factors of volatility and usage that are not considered factors in modeling data against the BFO. In theory, every specifically dependent continuant could be seen as a column in our warehouse row associated with the independent continuant being described.

In addition to qualities, the BFO describes other specifically dependent continuants that serve to place the continuant being described into a broader information context. These are known as Realizable Entities. One such realizable

entity is a BFO Role. A role is something that manifests through the independent continuant participating in some natural, social, or institutional process. For example, I've been using the idea of patient in my discussions of independent continuants. In fact, that is incorrect. Patient is a role that is realized in a person when she or he interact with the healthcare system in certain ways. Without that interaction, the person would exist (an independent continuant), but would not be a patient (a realized entity). Independent continuants participating in healthcare processes (defined as Occurrents below) will typically be in roles that will be increasingly specified and refined by our analysis. For example, two people in patient roles will be seen differently in a transplant process; one in a donor role, and the other in the recipient role. Roles help refine the contexts within which our continuants are seen.

Two other forms of realizable entities include BFO Functions and BFO Dispositions. The BFO defines a function as an essential end-directed activity of a continuant. A heart's function is to pump blood. A hammer's function is to drive nails. A screwdriver's function is to insert and remove screws. While most of us would have to admit to having used a hammer to drive in a screw once in a while, that doesn't alter the essential function of the hammer. In fact, we might describe this usage as dysfunctional in a way that is very appropriate to our data analysis in healthcare.

A disposition can cause a continuant to participate in a process or transformation under certain conditions. Of particular interest in healthcare is the idea of disease. A disease is a disposition toward some pathology under certain conditions or circumstances. That disposition can be latent, chronic, or acute. Our disposition to hammer a screw once in a while is typically latent, but results in the undesired pathology under the right set of conditions (e.g., a frustrated effort to insert a screw into excessively hard material or the lack of a screwdriver). A disposition toward heart disease will manifest in changes in the heart's function to pump blood. That manifestation will eventually lead to signs or symptoms that will bring the person to the healthcare system.

From our data perspective using the BFO, healthcare is largely about using signs and symptoms (qualities of continuants) to understand pathologies (abnormal functions, or dysfunctions) in order to treat or refocus the underlying diseases (dispositions). The connections between patients and diseases in our data warehouse will be through the vital signs, lab results, imaging, and other observations that lead to diagnosis and appropriate interventions.

BFO Occurrents

To the extent that our data about, and our knowledge of, continuants will change over time it is because continuants participate in, and are impacted by, BFO occurrents. Our knowledge of continuants would never change if they didn't participate in occurrents. As with continuants, the BFO provides guidance on how we should be analyzing occurrents. While the BFO model for continuants

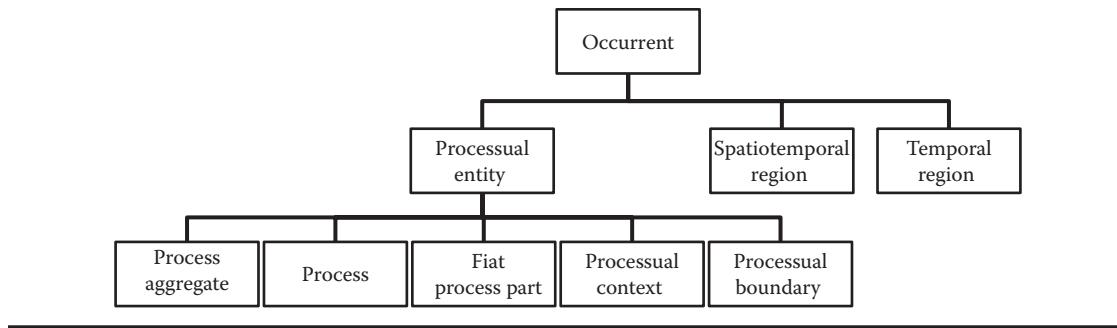


Figure 3.3 Basic Formal Ontology (BFO), occurrent details.

started with foundational spatial regions, the BFO occurrent model starts with foundational temporal regions (Figure 3.3).

BFO Temporal Regions

Temporal regions are the concepts we need in order to understand and model the passage of time. Occurrents occur in time, so the ways we analyze our source data to understand the role that time plays will be very important to the correct sourcing of our data into the warehouse. The BFO draws a distinction between regions that are connected and those that are scattered. A connected temporal region is one in which all of the temporal points form a continuous flow from the start to the end of the region. The time a patient spends in a clinical setting from admission to discharge would be a connected temporal region. We might not have data to represent every moment in that continuum, but we know the region is continuous.

Connected temporal regions can be of arbitrary duration, from the decades of a human life down to the exact moment of birth. The BFO defines a temporal region of zero duration as a temporal instant. Temporal regions of any nonzero duration are identified as temporal intervals. The distinction is important in the data warehouse, and analysis of this distinction in our source data will be critical to correct data sourcing. The moment of a patient's birth (a temporal instant) occurs during the time required for labor and delivery (a temporal interval). It need not correspond to the exact beginning or ending of that interval; but a temporal instant does serve as a form of temporal boundary, just as the object boundary serves as a delineation of objects. When we see a temporal instant in our data, we know there is some component occurrent (discussed below) that begins or ends at that instant.

Our source data systems will typically be unable to report data at the level of a temporal instant, so it is important that we properly analyze all temporal aspects of our source data to detect data that need to be treated as temporal instants. Computer timestamps aren't instants. They represent explicit temporal intervals that are limited by the computer technologies that provide the timestamps. At a macro level, a date is actually a 23, 24, or 25 hour temporal interval (depending upon whether the date is at the start or end of daylight

savings time in locations that include those requirements). At lower levels of scale, even a full timestamp represents an interval of arbitrary length. A timestamp accurate to the second is still a full 1 second interval.

As we analyze our data sources, we'll identify timestamps that are intended to represent temporal instants. We need to exercise care in consistently modeling and sourcing data so the intervals represented by the system timestamps are correctly used to represent the intended temporal instants. Examples include dates of birth, admission dates, and discharge dates. The weakness of using full dates to represent temporal instants becomes more obvious when the use of the data requires more precision. The precision of an instant of birth is far less important when calculating the age of an adult patient versus the age of a neonatal patient. The precision of the instant of admission is less critical when calculating length-of-stay for an extended encounter than when trying to calculate the time from arrival to admission. Since we can't know in advance the precision that will be required by a user of our warehouse, we will attempt to maximize our precision when analyzing sources.

Scattered temporal regions share features of connected temporal regions without the requirement that all of the temporal points within the region be continuous. The distinction will be important in our modeling of source data, and the ways we eventually use that data. Within a connected temporal region we will have a collection of sourced data, and we'll acknowledge that there are extensive gaps in that data that would need to be filled by additional sources. Within a scattered temporal region, we will have a collection of sourced data within the included temporal intervals and instants, but there typically will be no expectation that we have data sources within any included temporal gaps. This distinction is important. The larger the scale of time in our data, the more likely that we are dealing with a scattered temporal region than a connected temporal region.

The lifespan of a patient is a connected temporal region. Everything that happens to a patient takes place within that connected temporal region. Our knowledge of a patient is much more limited though. Our knowledge of a patient is limited by the scattered temporal region during which our healthcare systems has interacted with that patient. The more often the system sees a patient, the more the scattered temporal region in which our data falls will cover and align with the connected temporal region of the patient's lifespan. We collect histories on our patients in order to extend the range of those scattered temporal regions. Sourcing additional patient data from their personal health record (PHR) systems will similarly expand the scatter temporal region over which we have data for patients.

This distinction between connected and scattered temporal regions will constantly challenge data analysis for the warehouse. As the scale of activities varies, we'll usually find ourselves in situations where our data are temporally scattered, while our interests are often temporally connected. Much of what we do in analyzing data for the warehouse is an attempt to increase the coverage of the scattered region. We're always trying to find and source data that will fill in some of the gaps in our

scattered temporal region. Using that approach we can view the connected temporal region as the upper limit on any corresponding scattered temporal region.

BFO Spatiotemporal Regions

While all occurrents take place within time, they also take place within space. To support this, the BFO incorporates the intersection of the continuant spatial regions defined previously with the occurrent temporal regions to define the BFO Spatiotemporal Regions. Inheriting from both the spatial and temporal sets of concepts, spatiotemporal regions exhibit the same analytical challenges and opportunities discussed above for temporal regions.

A patient encounter in a hospital is an example of a connected spatiotemporal region. From the point of admission to the point of discharge, the patient was somewhere within the facility at all times. However, the data we have access to about that encounter for loading into our warehouse represent only a scattered spatiotemporal region. In this case, the scattering arises from both temporal and spatial ambiguities. There are times during the encounter where we have no data, but can infer from data that we do have where the patient was during some of the gaps in the data. For example, if we have data that indicate the patient was served lunch in his or her room at noon, and more data that indicate they were given a medication in the same room at 12:30 p.m., we're typically safe in presuming that the patient remained in his or her room from noon until 12:30 p.m.

In other situations, the data are sufficient for us to realize that we don't know where the patient was at certain times. For example, if the data record a bedside vital signs observation at 1:00 p.m., we know the patient was in her or his room at 1:00 p.m. If the next known fact about the encounter involves arriving in radiology at 2:00 p.m. for an ordered imaging, then we know the patient was in radiology at 2:00 p.m. What we wouldn't know is the timing or route of the patient's movement in getting from his or her room down to radiology. If, as source data analysts, we felt that this spatial and temporal ambiguity would result in problems in using the data warehouse for certain types of queries, we might seek out a data source for the hospital's transport data in order to find one or more data points between the two known points.

The level of spatial ambiguity in the data is a function of the scale at which the data are being analyzed. The in-patient encounter can be thought of as spatially connected at the scale of the hospital facility since the patient doesn't leave the hospital during the encounter. At the scale of the nursing unit, the encounter data became spatially scattered: At one time the data showed the patient on the nursing unit and later in radiology. At the nursing unit level of detail, the encounter data became scattered spatially. As with temporal regions, the more data we source into the warehouse, the more the scattered spatiotemporal region illuminated by that data will approach the upper limit of the entire connected spatiotemporal region being represented. As data warehouse analysts, we're always looking to fill in the picture to make it more comprehensive and valuable.

BFO Processual Entities

The source data we analyze for our data warehouse aren't ultimately focused on space and time. The spatiotemporal concepts in the BFO provide the space-time canvas against which BFO Processual Entities occur. It's these process entities that result in the data we source into the data warehouse, whether that data represent changes to the qualities of continuants, or outcomes associated with the occurrents themselves. We're finally at the point in the BFO where most of our source data will be analyzed. Space and time form the stage, continuants provide the actors, and processual entities are the actions against which we want to capture data in our warehouse.

The BFO offers a breakdown of processual entities that includes considerable analogies to the breakdown of continuants discussed above. Just as the BFO continuant model seemed anchored around objects, the BFO occurrent model is anchored around the BFO process. Processes have distinct beginnings and ends, and can include other processes, just as continuant objects can include other objects. The narrower a process, the more likely it will occur within a connected spatiotemporal region; the broader a process, the more likely it will occur within a scattered spatiotemporal region. For example, a medication process takes place in a scattered spatiotemporal region, while its component processes of ordering, dispensing, and administering the medication each occur individually as connected spatiotemporal regions.

Most process data in our data sources will describe processes like these. The BFO supports a few alternatives that handle exceptions to the general discrete process paradigm. BFO Fiat Process Parts are processual entities where the beginning and end are not directly discernible, requiring agreement by fiat as to what constitutes the boundaries. They are analogs to the fiat object parts that allow for ambiguous components of continuant objects.

Because of their amorphous definitions, fiat process parts can be very difficult to capture in datasets, making their correct analysis more important. In many cases, an inability to lock down the boundaries of a process start won't create major problems for us. That part of a hospital stay where the patient is becoming likely to be discharged soon is a commonly discussed period with patients, but an inability to exactly define and capture its start and end isn't a serious analytical problem. We devote more time to defining fiat process parts that seem like they should be able to be more precisely defined because we sometimes don't want to admit that we don't all agree with the fiat definitions we capture in analysis.

Hospital encounters are a great example of common disagreements. I have often been surprised at just how difficult it is for some hospitals to agree on their definitions for when an encounter begins and ends. The core time period between admission and discharge is easy enough, making that core period a process part, but what about the period that begins earlier than that? If, in the morning, a physician makes an appointment to admit a patient later in the day, hasn't the encounter started? If the appointment transaction is the boundary,

then the pre-admission part of the process can be considered a process part, and knowing the transaction boundary makes it a bona fide process part; but what about the period during the office visit where the physician begins to think about admitting the patient to the hospital? Has the encounter started at that point? Is there a flat process part that starts the encounter when the physician begins her or his consideration? What about the encounter's post-discharge period? When does an encounter actually end? I've seen financial adjustments made to encounters years after discharge. When do those encounters end? It might not be critical to always be able to answer these questions, but the flat process part gives us a vocabulary for analyzing the data associated with these examples.

BFO Process Aggregate

Unlike an encounter process where everything that happens to the patient can be considered a component of the overall encounter process, a BFO Process Aggregate allows data to be captured about sets of processes that are not connected in ways that allow them to be treated as components of each other. Analogous to an object aggregate such as a cohort of patients, the process aggregate allows for combining processes that might not otherwise be grouped based on the formal definitions of processes and their components. The collection of behaviors of an entire surgical team during an operation would be an example of a process aggregate. Each individual's behaviors might constitute a formal process, but the combined set of interactions would be difficult to describe as a single process. That distinction, and the decision as to whether to treat the set as an aggregate or single large process, requires analysis during the data sourcing effort.

BFO Process Boundary

A BFO Process Boundary is a process component that takes place in a temporal instant, starting or ending a process, or else separating two connected process components. Data about process boundaries will inherit the temporal ambiguity associated with all temporal instants that need to be represented by systems timestamps that always represent temporal intervals, however small the interval. These boundaries include elements of the patient's lifespan (e.g., birth, death) as well as components of our clinical processes (e.g., initial incision during surgery).

BFO Processual Context

The final BFO element, the BFO Processual Context is analogous to the continuant site discussed above. Just as continuants are found at sites, many processes occur within a context that isn't technically itself a process. A hospital encounter is an example of a processual context. Each encounter has a start and end like a process, but otherwise isn't defined by any particular process

definition. Instead, the encounter is the context within which a lot of other clinical and administrative processes happen.

I've devoted a lot of time and space to discussing the Basic Formal Ontology even though this discussion hasn't yet actually led to any particular discussion of the analysis of source data that this chapter is nominally about. That will change now. By using the BFO as the foundation for all of our data discussions and analysis, we'll be in a safe position to know our analysis of potentially hundreds of sources will tend to move according to the same analytical process and in the same direction. Using this and the next few ontologies consistently is what will allow our data warehouse to seem like it is telling the story of our patients and their health rather than the story of how our HIT systems were used to impact those patients and their health. We want our warehouse to have a healthcare feel, not an information technology feel. These ontologies will help us get there.

Information Artifacts

Building on the BFO foundation, we now want to concern ourselves with our data sources for the emerging warehouse. These data sources, in a variety of forms and technologies, are all information artifacts. We'll use the Information Artifact Ontology (IAO, Figure 3.4) to understand and model them. We don't want to simply load the sources directly into the warehouse. We want to load the information that these sources are *about* into the warehouse. Sometimes that will look the same, and many analysts skip this level of analysis and jump right

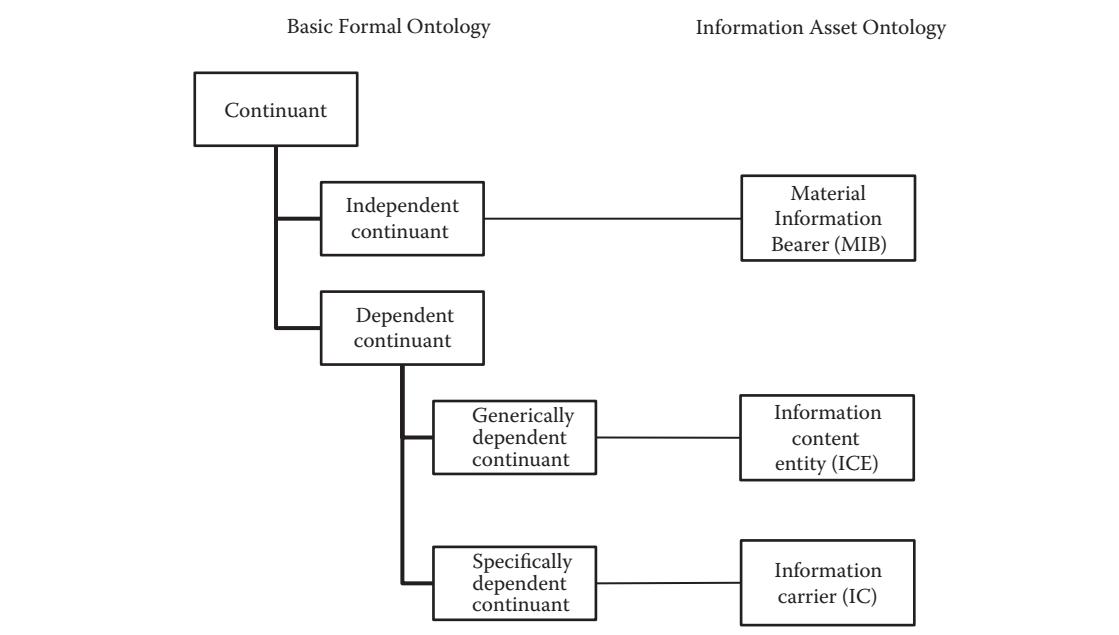


Figure 3.4 Information Artifact Ontology (IAO), high-level categorization.

to loading the source data; but many times the clinical picture we want to load is slightly different than the systems picture painted in our information artifacts. We ignore the distinction at our peril.

The IAO employs three central concepts for understanding information artifacts, each mapped back to an appropriate spot in the foundational BFO: the Material Information Bearer (MIB), the Information Content Entity (ICE), and the Information Carrier (IC).

IAO Material Information Bearer

At the level of the independent continuant is the Material Information Bearer (MIB). MIBs are the information artifacts that actually bear the information we're looking for. The copy of this book is a material information bearer, whether it is electronic, printed, or recorded as an audio disk. Another copy of this book is a different material information bearer. Note that the MIB is an instance of something, allowing it to qualify as an object in the BFO: an independent continuant. Our warehouse sources won't include instances of books, though. Our sources will arrive as database tables, extracts from databases, other files types (i.e., XML, HL7, CSV), or telecommunication messages. Each instance we receive is an MIB. Whether a database table or an XML file, the IAO provides a structure for consistent and complete analysis. As analysts, we must think about what's in each of the MIBs that arrives, and that's where the IAO provides specific concepts that are not offered in the higher-level BFO.

Before we continue, we have to back up and discuss one more aspect of the BFO: the *Generically* Dependent Continuant. Previously, we encountered the BFO *Specifically* Dependent Continuant as something that inhered in a specific BFO Independent Continuant. Each patient can have a name, a weight, and a blood type; and those specific properties inhere in the respective patients. How do we note that name, weight, and blood type are actual properties that can be thought about as inhering in independent continuants? The BFO Generically Dependent Continuant provides that ontological function. An object (independent) can have a name (specifically dependent), and what that name provides is "name-iness" (generically dependent) for the object. I tend to think of the generically dependent continuant as my ontological metadata, answering questions regarding what I can know about the instances of the independent continuants that I eventually look at.

The IAO offers a conceptual pathway for analyzing or decomposing a message or dataset. As a BFO Object, the MIB can be broken down into its constituent parts. The resulting parts can still be considered Material Information Bearers, except as decomposed components they would be mapped to the BFO as Fiat Object Parts. Whether what you are analyzing is an Object or a Fiat Object Part can be an arbitrary distinction. The important thing for analysis is to be able to break datasets down to their smallest components that can be sourced and processed by your Extract-Transform-Load (ETL) subsystem. A sourced database

MIB might include many table MIBs that, in turn, contain many column MIBs. A sourced XML message MIB might include many layers of structured tags, each an MIB, down to a lowest level at which data values occur. While the database example might only have data values at the lowest, or leaf, level of the structure, the XML message example might include data values anywhere in the complex structure. In some cases, the presence of the tag *is* the intended value. The key analytical element to keep in mind when decomposing MIBs is that the analysis is always about the data source or message *as an information artifact* and not the content or intended meaning of the data or message.

This book (again, your copy of which is an MIB) has many structural components that can be described by component MIBs. It has its title page, a table of contents, an index, and many chapters. Each chapter is broken into smaller cascading sections, and sometimes includes tables or figures. A well-structured book is easier to read than a poorly structured one, regardless of what the book is about. The decomposition of a MIB into its component MIB structures could go on indefinitely; so an analyst needs to know how to stop, as well as how to continue. If the intent is to analyze what a book is about (perhaps as a book reviewer), then structural considerations might stop at the chapter or major section level. If the intent is to analyze the grammatical correctness of a book (perhaps as a copy editor), structural considerations might continue down through paragraph, sentence, clause, word, and punctuation levels. The IAO supports this detailed analysis, but I don't expect to go that deep for most data warehouse sources for now, although these considerations might become more important in a future when natural language processing starts playing a greater role in how we analyze and source data into the warehouse.

As I move past the Gamma phase, and start looking at natural language processing opportunities against larger clinical report data, I'll eventually get to that level. For now, though, analyzing my book at the paragraph level would probably be extreme for our purposes. Just always keep in mind that we could be modeling at those deeper levels if situations arose where the data of interest warranted this analysis.

IAO Information Content Entity

Once I've analyzed the structural composition of my data sources, I start looking for content. Within the decomposed MIB structure, I evaluate each MIB that I can associate with a value (or that constitutes its own value when present) that is of interest within the biomedical domain of my data warehouse. Each of these MIBs is an IAO Information Content Entity (ICE). An ICE is a BFO Generically Dependent Continuant that is actually *about* the data content rather than simply providing the structure for the content. The relationship between the structural MIB and the content-oriented ICE is typically straightforward, often one-to-one.

For example, in the message fragment <LAST NAME>BIEHL</LAST NAME>, Last Name is an MIB, the value of which is an alphanumeric entry, with an apparent

proper name syntax. Last Name is also an ICE, the value of which describes me, a person with the name designated by the value received. The MIB exists as a BFO Fiat Object Part, an independent continuant that exists as a component of an information artifact. The ICE is a BFO Generically Dependent Continuant, a quality that can be instantiated against any BFO Independent Continuant. Later on in the analysis, I will be the independent continuant of interest in this fragment, but I am not described by the IAO.

MIBs, as independent continuants, are the datasets and messages that we use to source data for the warehouse. ICEs, being generically dependent continuants, are the metadata we're after as analysts looking for how the sourced data should be loaded into the warehouse. The distinctions drawn between the two concepts can be confusing. The way I keep them straight in my head (with apologies to the ontological purists that I am undoubtedly offending) is to look at ICEs as things that can have values (i.e., they are always *about* something). MIBs focus on structure, and lead me to ICEs. If I look at MIBs as a structural decomposition of my source data, the lowest level leaves of that decomposition will typically be ICEs. There might have been other ICEs throughout the decomposition (e.g., the tag for <FULL NAME> might not have its own value, but will aggregate the tags for <FIRST NAME> and <LAST NAME>), but I mostly expect to see ICEs at the bottom of this analysis.

IAO Information Carrier

The third IAO concept of interest here, the IAO Information Carrier (IC), finally brings us analytically into contact with the data we're trying to source. The ICs will serve, with some qualifications below, as the source data we are analyzing and want to load into our warehouse. ICs are BFO Specifically Dependent Continuants; they specifically inhere in the actual MIB that is being sourced. The book you are reading is a type of MIB that typically has a title page (MIB) with a title (ICE). This book actually and specifically has the title (an IC) printed on the cover page. Another copy of this book is a different MIB that also has a title on its cover page, perhaps in a different font or language (a different IC). The IC isn't about the abstract title that I had in my mind as an author; the IC is about the ink on the page, or the pixels on the device screen, or the sound intonations on the recording, or the ASCII characters in a message.

In the <LAST NAME>BIEHL</LAST NAME> message fragment, the five-character alphanumeric string “Biehl” is an IAO Information Carrier. As an information artifact, it conforms to the constraints we might have specified for the Last Name ICE, namely that it consists of alphanumeric characters with apparent proper name syntax. Resolving syntax issues that arise between IC contents and ICE constraints is a data quality role within our ETL that is directly supported by mapping our data sources against the IAO.

I know this can seem overwhelming at times, and there can be cases where modeling data sources with this level of rigor seems quite tedious and

time-wasting. The intent is to focus on using these concepts where they add the most value, and passing over them when they don't (again, with my apologies to the ontology purists). The most important point is to recognize that there's a difference between what we see in a data source (the ICs) and what those values tell us about the world (what the ICEs are *about*). If the source data are well defined and data quality is high, the distinction we're discussing might be minimal and we can analyze our sources fairly quickly.

However, if the source data are less controlled (i.e., comes from outside the institution, or from an application we don't know much about), of poorer design (i.e., no data normalization, poor naming standards, or inconsistent formats), or includes data quality problems (i.e., range check failures, misspellings, broken dependencies, omissions or gaps, or variable data formats); we'll want to move through the IAO analysis layer much more slowly and carefully. Even when an IC is erroneous (i.e., misspelled), it still represents what the ICE is about.

We need to analyze these issues in our data sources so data we load are what the sources are *about*, not just what they *contain*. The warehouse we load is also an artifact, so what we actually load are also ICs for our intended target ICEs. The ICs we load won't be the ICs we received, and our analysis of these sources provides for the transformation between the two. The Information Artifact Ontology (IAO) allows us to compartmentalize various portions of our data sources for analysis, differentiating structural versus content components. The content components are *about* the things we want to know, but not all source data are equivalent analytically.

Biomedical Context

The context of interest in this book, and in our emerging data warehouse, is biomedicine. Some of our source data, like the title of a book or the name of a chapter, are going to be about our practice of biomedicine rather than about the populations, patients, and specimens of interest in that practice. As we analyze our data sources for the warehouse, we need to anticipate and capture this distinction in our source data. Data about our biomedical context often tell us more about ourselves than about our patients, although we also recognize that these data are useful for inferring additional properties of our patients.

Our analysis of our source data will be further aided by the addition of another ontology to our analytical toolkit: the Ontology for Biomedical Investigations (OBI, [Figure 3.5](#)). The concepts in the OBI, all grounded in our foundational BFO, are about biomedicine and its practice rather than being directly about the populations, patients, and specimens on which we practice that biomedicine. In the same way that the IAO replaced the abstract concepts of the BFO with more specific concepts for discussing information artifacts, the OBI provides a means to analyze data using the concepts of biomedicine, a critical step in analyzing data sources for a biomedical data warehouse. The source ICEs analyzed using the IAO

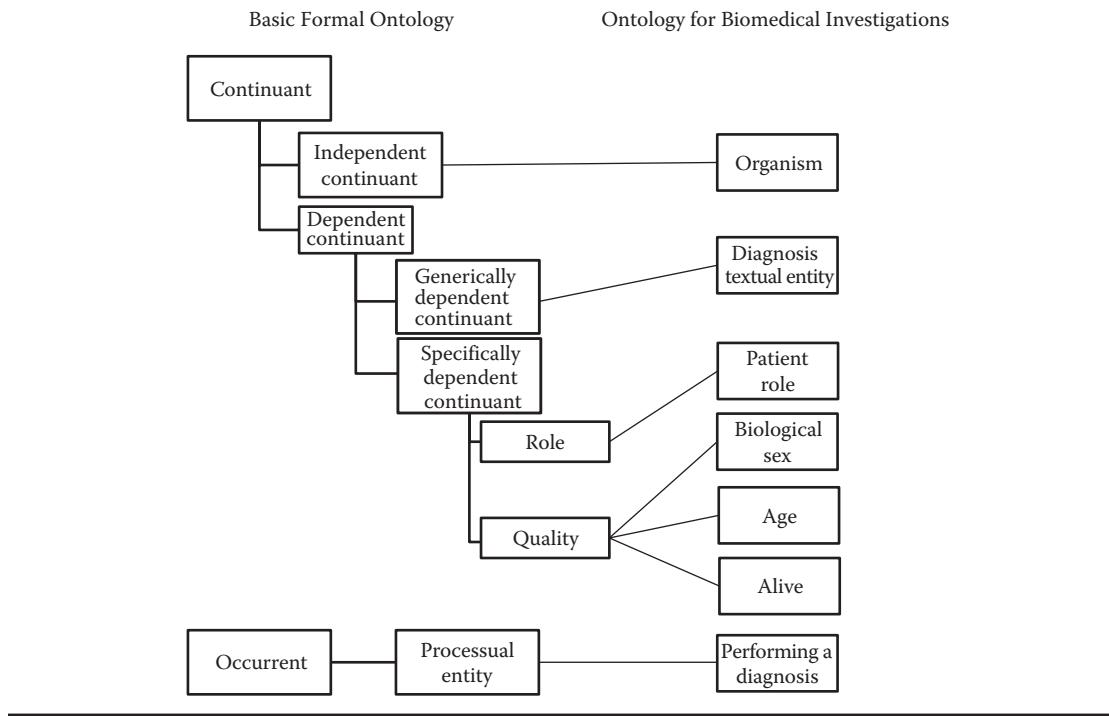


Figure 3.5 Ontology for Biomedical Investigations (OBI), representative concepts.

end up being mapped to concepts in the OBI, eventually serving as the mapping criteria for loading the sourced data into the warehouse. The IAO ICE concepts are said to be *about* the things represented by OBI concepts.

OBI Processes

Many of the processes that are represented in the data we analyze as source data are modeled at the biomedical level represented in the OBI. The BFO includes concepts for independent continuants to participate in processual entities, such as when a provider and patient both participate in a diagnostic process. The OBI provides the concepts needed to conduct data analysis at that level of conceptualization rather than having all of our data analyzed using only the broader concepts of the BFO (Figure 3.6). When analyzing data sourced

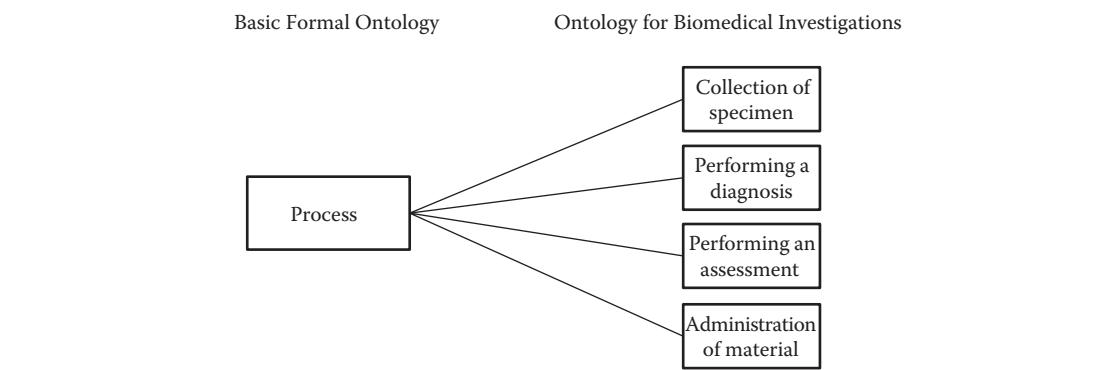


Figure 3.6 OBI Process, example of broad concepts.

from processes, our analysis will typically discover combinations of biomedical data about those process instances as well as actual clinical data about our populations, patients, and specimens. Most of the instance, data in our sources will require analysis and loading of the definitional concepts to support those instances in our warehouse dimensions.

Our grounding in the upper-level BFO ontology gives us an overarching set of heuristics that we need to apply when analyzing source data at the OBI level. The BFO showed us that our world is made up of, and so our source data describes, continuants and occurrents. When we're looking at data about continuants, we should be asking about the occurrents that create, alter, or consume the continuants. When we're looking at data about occurrents, we should be asking questions about the continuants that participate in, or are impacted by, those occurrents. Appropriately integrating these two perspectives is a core analytical requirement if we want users to be able to query our data warehouse without having to know the structure or content of the systems from which data were sourced.

Another important analytical heuristic comes from the BFO: The distinction between continuants and the roles those continuants play. A person is a continuant in the BFO, but being a patient is a role in the OBI. We've also got people in our warehouse who exist in physician, nurse, and other OBI roles. The roles apply to our data in action, but don't absolutely define the continuants in those roles. In fact, if we generalize the role of clinical observer in the OBI, we see that the continuants in that role are often people but could also be certain medical devices. This is why our source analytics needs to be informed by the OBI ontology, so we don't make some common mistakes in data sourcing that will put data in places that don't correspond to the semantics against which users will be looking for that data. The data should be a description of our biomedical setting, not the content of our databases.

An example of all this might be a lab test to assay levels of acetaminophen in a patient, an OBI process that is a BFO occurrent. We might be sourcing the results of a test under its procedure code in a laboratory system, typically seen as CPT 82003. A problem with simply sourcing the data against its procedure code is that most of the other data we have about acetaminophen treats it as continuant data, not occurrent data. Most users of the warehouse will think of acetaminophen as a noun, and will look in the place where drugs are stored, not lab tests. Additionally, how we store those drugs needs be analyzed, as well. The OBI guides us to a view of treating the BFO continuant of interest as a molecular entity, an OBI continuant. We then define the different roles that this entity can play in our source data. Acetaminophen will often be in a drug role in the OBI, but it could have been in an allergen role instead. The role will vary by the situation in which data are being sourced.

The important thing is that, as source data analysts, we want to recognize all of these distinctions and do our best to source the data as completely and correctly as possible. We want to define data by its semantic meaning, not its

structure in any single source system. If all data are sourced semantically, it increases the chances that the data will be right where users expect it, and will be able to be queried consistently across lots of data that were sourced from different systems with different data structures and organizations. Our job as analysts is to get past these system differences, and capture source data that provides a complete biomedical context.

What this means in terms of our assay of acetaminophen is that we need to do the necessary analysis to ensure we capture the entire context for our source data, not just the data explicitly stored in the source. The specifics of how we ultimately implement that fuller context will unfold in the chapters that follow. If possible, we'll simply extract more data as we pull in the particular data sources containing this information in order to provide richer dimensionality for our loaded facts. If we can't directly source the data, we might end up doing some form of automated curation of the data after they are in the warehouse, to further enrich the data.

To connect our lab test to our drug, we might use some simple text processing to extract “acetaminophen” from the name of our lab test, find that name in our drug list, and extract the appropriate drug code along with the source procedure code. Text processing can get very cumbersome, so it is only useful in the simplest of situations, but it can be very effective when needed. Alternatively, if we've sourced the Current Procedural Terminology (CPT) Ontology into our data warehouse, we'll already have the mapping connections that the curators of that ontology have provided, including the match between our Assay of Acetaminophen (CPT 82003) and Acetaminophen (RxNorm 161).

If we can make the connection during initial sourcing, then the data we load are already enriched. If we can't make that connection at extraction time, we can do the cross-matching after the data are in the warehouse; and add the drug connection after the fact. Either way, we end up with enriched facts that include dimensionality that was unknown, or only implicit, in the source systems from which the data were gathered. The norm will typically be to add the dimensionality to enrich facts in the warehouse after they've been loaded, for two reasons:

(1) we don't want to overcomplicate our source extraction jobs with a lot of extra logic, and (2) the ontologies needed for many of the connections aren't loaded as early as some of the data.

We'll always need to be able to auto-curate existing data as we add new semantic ontology capabilities. For these reasons, we want to do all of the analysis up front while defining our source data, even though some of that extra implicit data won't be loaded into the warehouse until sometime later. Implementing that analysis might require waiting until the appropriate ontologies have been loaded to the warehouse. Whenever we add new ontologies to the warehouse, we'll always have an opportunity to apply some form of curation logic to any data already loaded into the warehouse prior to their loading. Semantic ontologies will come to be seen as very important source datasets for the warehouse because of the enrichment they enable. The governance process in Chapter 20 will prioritize those sources accordingly.

OBI Process Aggregates

The BFO provides for the aggregation of related processes when those processes together constitute something of interest. The aggregate process might be considered a process itself, but is likely to be only loosely considered so.

Figure 3.7 illustrates some aggregate processes that might be of interest in our data warehouse. A common characteristic among these is that the processes in question ultimately have both start and end times as any process would, but are unlikely to be considered as a single stream of action from start to finish. They typically cover BFO scattered spatiotemporal regions. As source data analysts, we want to be on the lookout for these aggregations. We want to be sure that the way we analyze the data will allow all of the components of the aggregates to be easily seen by users, and we want to be careful to ensure that we track correct dimensionality throughout any related facts. The continuants that participate in these processual aggregates tend to be less stable in their definition than when looking at narrower independent processes.

An example might be a course of medication for a patient. The aggregate process might include the ordering of the medication and multiple administrations of the medication to the patient. This aggregate process might take place over the course of a few minutes, or it might run for several hours or days. The source data associated with the medication order might include only a high-level indication of the intended medication; perhaps specifying simply Analgesic, or Acetaminophen. The actual dispensed medications will be much more specifically defined medications, like Acetaminophen 250 mg TAB. The timing of the administrations of these medications could vary depending upon whether the order called for specific administration times, or was on an as-needed basis. Either way, some administrations will fail because the patient might be sleeping or off the nursing unit at the expected administration times.

A result of this complexity is that the source data representing the aggregate process—as opposed to simply any source data representing individual process components of the aggregate—needs to be analyzed for nuances that might create problems when users try to query and use the data at an aggregate level. Making sure that warehouse users can see continuity in the aggregate data is important, even if medication identifiers change over the span of the medication course. Additionally, analysis might show that the dispensing of the

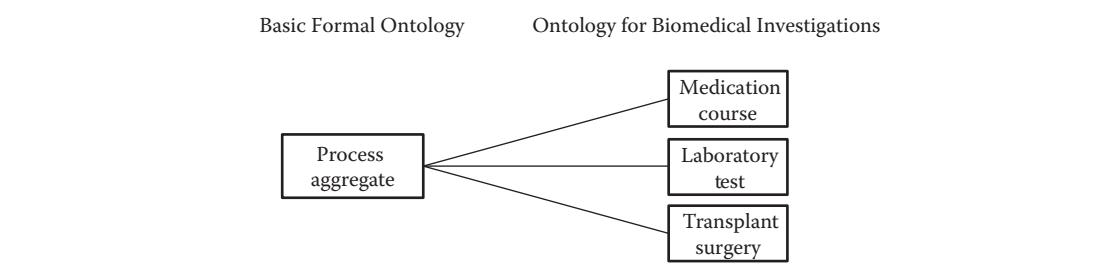


Figure 3.7 OBI Process Aggregate, example concepts.

medication needs to be included in the aggregate. This is particularly likely if the medication is dispensed in the nursing unit so the final administered drug code might not be correctly known until the medication is actually dispensed. Some users might expect that an allergy assessment or reconfirmation event be included in the aggregate preceding the first administration of the medication. Since aggregate processes are fairly arbitrary, substantial analysis is needed to ensure that all the necessary pieces can be properly aligned and reassembled in the data once loaded into the warehouse.

Lab tests are more complicated than medication courses. Like a course of medication, a lab test can be very complicated, and can extend over numerous individual processes. The aggregate process has facts and features that users will look for in the warehouse, and the analyst has to ensure that the aggregate view makes sense in the data. One complication that must be routinely handled is that the laboratory panel ordered is itself an aggregate set of individual laboratory tests. That means that the procedure order for the test won't be the same as the procedure for which results are received because the resultant procedure is only one of many procedures in the aggregate set that was ordered. Also, where a medication is dispensed and brought to the patient, a lab test typically involves first drawing a sample from the patient and taking it to the lab. The number of distinct processes needed in this overall aggregate can be quite cumbersome to properly represent in the sourced data.

Generally, the source data for a single lab test will involve the initial order for the lab panel that included the test, followed by a specimen collection from the patient, the receipt of that specimen in the lab, the performing of the test in the lab, the return of preliminary results, the quality control of those results, and the return of final results. Some of these processes take place in the Electronic Medical Record (EMR) System, and some in the Laboratory System, so it will be uncommon to be able to extract all of the needed data from a single source. Additionally, each source might identify the data differently, with a Placer Order Number on the EMR side, and a Filler Order Number on the lab system side. This necessitates cross-source analysis in order to define proper metadata for sourcing and loading the data. Lab tests where erroneous results are produced, or samples are inadequate for processing, or additional new tests are triggered by the tests in question only make the aggregate more complicated.

Process aggregates can be arbitrarily complex, and the analyst might not always have sources of data available for every component. For example, a surgical transplant aggregate might involve two surgeries in side-by-side theaters, or might involve a donor surgery that took place hundreds of miles away, and hours ago, in another institution. Circumstances for data sourcing will vary based on these types of circumstances and the availability of automated systems to support data collection.

OBI Processual Contexts

In addition to aggregate processes, the BFO also offers an alternative construct for processual contexts that is relevant in analyzing OBI-level source data. A context is not so much a process as a setting in which processes can take place. The OBI includes several constructs that represent the conceptualization of these processual contexts in which the various processes and process aggregates take place (Figure 3.8). Most of the data we'll place in our data warehouse will somehow be anchored (i.e., dimensionalized) against the encounters and visits within which everything else happens.

Process contexts and process aggregates can be difficult to differentiate during analysis, but that makes them very useful in analyzing source data, precisely because resolving any nuances increases the value of our analysis. Generally I use the context construct when I'm looking to differentiate what is inside the process from what is outside the process. I use the aggregate construct when I'm looking inside the process to understand its components. Likewise, I see the aggregate as interconnected components, while I see the context as an overall container. The distinction is somewhat subjective depending upon what you know about your data and the actual biomedical activity represented by that data.

While we work out these distinctions in our source data analysis, we improve the semantic richness of the data we'll eventually load into the warehouse. Each level of process, process aggregate, and processual context offers a framework for analyzing data and working to ensure we source and properly load them into the warehouse. Eventually, we'll be sourcing data across a variety of contexts, involving data from many aggregating processes, with process and process component details furnishing the source data that we analyze and source into our data warehouse.

The one main thing that these constructs have in common as source data constructs is what they actually don't give us: They don't actually tell us anything about our patients and research subjects. The OBI ontology is about the biomedical investigation itself. That investigation will collect and identify information about the biomedical circumstances of our subjects, but the OBI doesn't describe that set of outcomes of those circumstances. In fact, most data

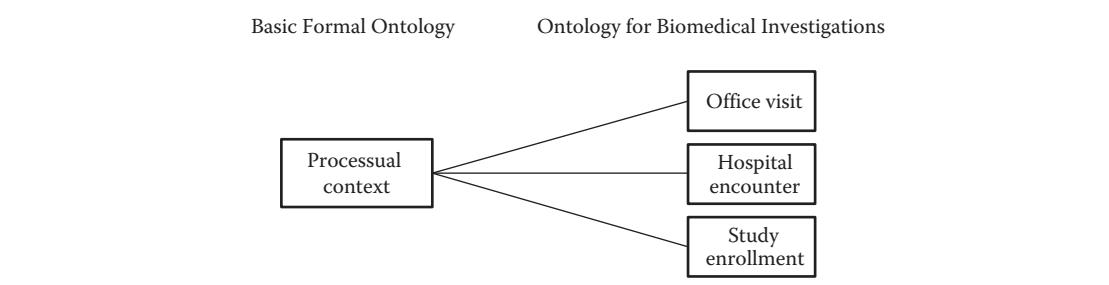


Figure 3.8 OBI Processual Context, example concepts.

in a healthcare institution aren't about the health of patients directly. It's mostly about the various investigations that take place in our biomedical institutions or settings. The actual health of patients usually ends up as implicit data behind all of the processes about which we record data. The data we collect need to be turned into the clinical picture of our patients.

Clinical Picture

As a result of carrying out processes in our biomedical context, the data that describe a clinical picture of our subjects finally appear in the data available as data sources. It is at this level that our ontological journey culminates in actually knowing something about our populations, patients, and specimens. The IAO allowed us to understand information artifacts that were about our biomedical contexts. The OBI allowed us to understand those biomedical contexts that were about practicing biomedicine. Finally, the Ontology for General Medical Science (OGMS, Figure 3.9) will provide us a set of concepts for understanding the data arising from practice.

As source analysts, we want to extract data from our source systems that will paint as clear a picture of the biomedical condition of our subjects as possible. The framework within which we do our analysis needs to be grounded in data about our patients, not data about our clinical practices in our clinical settings. One way to do this is to analyze our sources, looking for data that occur at the same levels of process definition as we saw in the OBI model for our clinical practice: processes, aggregates, and contexts. In the OGMS ontology, we find concepts for doing this (Figure 3.10); although finding those concepts in our source data will present many challenges.

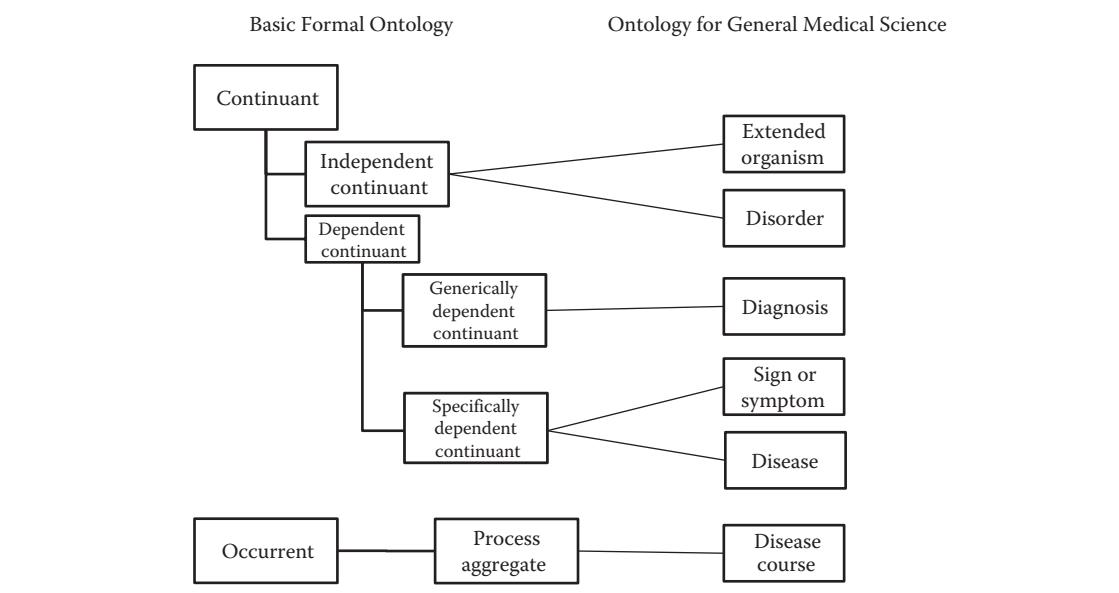


Figure 3.9 Ontology for General Medical Science (OGMS), representative concepts.

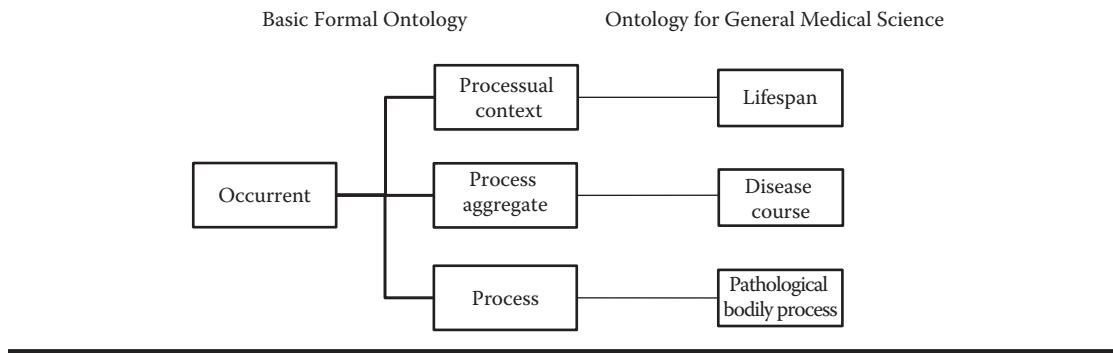


Figure 3.10 Ontology for General Medical Science (OGMS), representative occurrent concepts.

The OGMS lifespan of a patient is the processual context in which everything about a patient plays out. Over that lifespan, patients will experience numerous courses of disease, each a process aggregate in the OGMS ontology. Those disease courses will result from OGMS pathological bodily processes, the severity of which will bring the patient into contact with our healthcare system. As usual, I'm using the term "patient" very loosely. Our biomedical warehouse can be about so much more than just individual patients in healthcare. The OGMS gives us concepts for identifying and discussing various BFO continuants that can then be the subject of our data and analysis (Figure 3.11).

If we're dealing with population health or clinical research contexts, we'll often be analyzing data about OGMS organism populations. These aggregates of organisms (typically people, but sometimes nonhuman animals) can be the subject of data in the warehouse as aggregates, as well as being made up of the individual OGMS organisms that make up those aggregates. A critical OGMS concept that mediates between these two levels in our data is the OGMS extended organism: an aggregate that includes everything about some primary organism of interest as well as everything else that might be incorporated into the organism without being considered part of the organism.

You and I are human OGMS organisms, but we're also OGMS extended organisms that include all of the biome in our gastrointestinal track, the fillings in our teeth, any surgically implanted devices we might contain, and the foods and

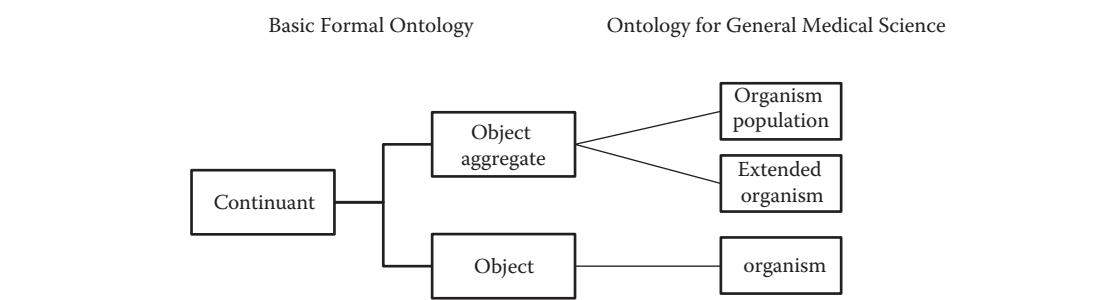


Figure 3.11 Ontology for General Medical Science (OGMS), representative continuant concepts.

medicines we've recently consumed, as well as any latent or active viruses that our immune system hasn't yet eliminated. This variety of things, some of which are organisms in their own right, are an important part of what it means to be the subject of a biomedical investigation or treatment. At any given time, our extended organisms might have additional inclusions that wouldn't be considered normal parts of a human organism, including foreign objects like a swallowed marble, or a bullet lodged in a shoulder. The extended organism is the complete subject of biomedicine, and it is actually what we mean when we talk about collecting data about patients.

Other concepts within the OGMS ontology can be used to analyze source data, emphasizing those aspects of the data that describe a patient's clinical picture independently of the clinical setting in which all of that data are generated (Figure 3.12). These concepts allow us to identify and extract data about the actual clinical picture of our patients without the complication of sifting through all of the clinical setting and investigation details. We'll have all of that data, and we'll continue to need to infer much of the clinical picture from the clinical setting for years to come, but by analyzing source data in such a way that we try to separate these two perspectives, we'll end up with data in our warehouse that support a broader base of user query and analytic requirements.

In the years that I've been working in healthcare to build data warehouses, the emphasis has been on pathology in patients and the treatment of disease.

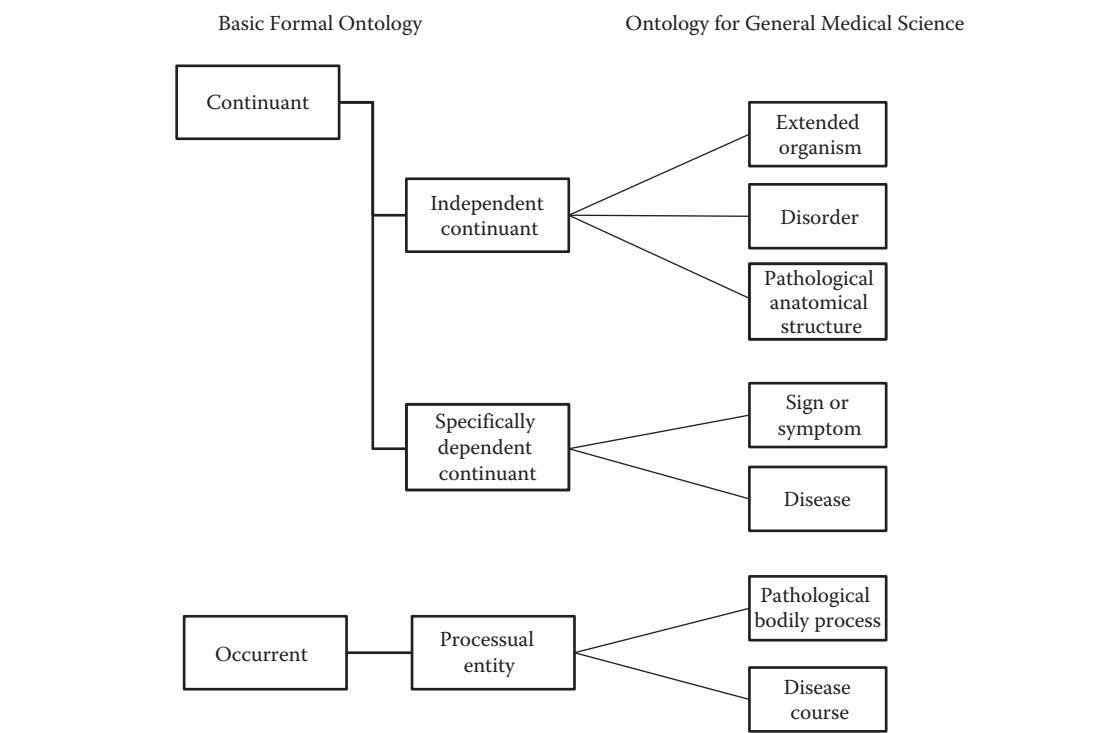


Figure 3.12 OGMS representative conceptual scenario: patient has a disease resulting from a disorder with disposition toward pathological bodily process which produces pathological anatomical structure recognized as sign or symptom.

Wellness has been the desired condition, but most of the data I've ever loaded into a biomedical data warehouse have been about clinical dysfunction.

The examples I've been using to illustrate analysis of source data with these ontologies largely reflects that historical bias, but healthcare is changing. We're increasingly emphasizing broader management and fuller accountability for the wellness of the patients in our charge. These ontologies will support the concepts needed to source and integrate wellness data into our warehouses as that data become increasingly available from our biomedical settings.

Ontological Levels

We've now encountered the four main ontologies that a source data analyst should have in mind when identifying and analyzing data sources for the biomedical data warehouse (Figure 3.13). They correspond roughly to the three types of data that the analyst will be eliciting and defining from each source: (1) IAO data about the data or metadata, (2) OBI data about clinical practice, and (3) OGMS data about the populations, patients, and specimen of interest.

Every candidate data element for the warehouse will be evaluated to identify the ontological level that the data supports. Some elements will be distinct to one level (e.g., Provider Clinical Specialty exists only in the OBI), while others will map to multiple levels (e.g., patient age is used for age at admission in the OBI and age at onset in the OGMS). Data mapped to the IAO concepts will eventually be used to define information in the Metadata dimension of the data warehouse, and will drive the generic ETL subsystem execution in the Beta version of the warehouse. Data mapped to both the OBI and the OGMS will eventually be loaded into the warehouse as the contents of our dimensions and fact tables.

There will be other semantic ontologies that we'll add to our analytical toolkit as we proceed, particularly as we start sourcing data in the Beta and Gamma phases of the warehouse development. These ontologies will be *domain-specific* ontologies. They'll add richer detail and refinement under these high-level ontologies. Once those domain-specific ontologies begin to be loaded, we'll be referring to the IAO, OBI, and OGMS ontologies as *mid-level* ontologies, and the BFO as our *upper-level* ontology. Some of the domain-specific ontologies will already seem familiar (e.g., SNOMED, LOINC, RxNorm, MeSH), while others will have entered the domain

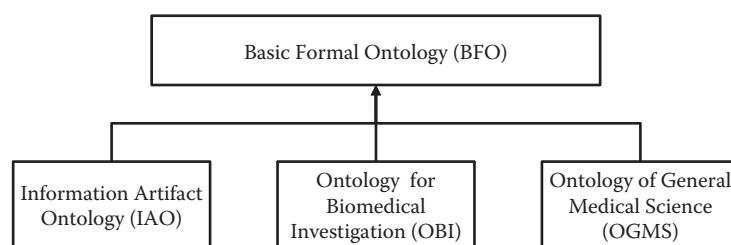


Figure 3.13 Core conceptual ontologies to support data source analytics.

even after this book was written. Many of them will end up being treated as data sources that need to be loaded into the warehouse because, during source analysis, we'll have decided that ontological references would be an effective way to dimensionalize and harmonize some of our source data.

As users access the warehouse in the future, the distinctions among these models become increasingly important, and the data derived from each will be used at different times. A challenge will be to integrate data across domains in ways that are useful and correct. The better we analyze our source data for loading, the simpler and more powerful the warehouse will be when we're done. Analysis is our investment in future completeness, correctness, and ease-of-use. Warehouse users studying clinical practice effectiveness will be looking to correlate the processes in the OBI data to the outcomes in the OGMS data in order to evaluate practices and determine best practices. This is an area of research that will benefit from clear delineation between these two types of warehoused data. The distinction between the domains is an advantageous strength if properly modeled by the data analyst.

Other times, the distinction will point to a dangerous weakness. Clinical researchers will want patient phenotypes, and we won't necessarily have them. Our OGMS clinical picture looks a little like a phenotype, but we see, through the OBI, that there might have been process variances and practitioner biases that influenced what ended up in our clinical picture. The stronger and more complete our OBI model, the more likely that the OGMS clinical picture will approximate a correct patient phenotype. To many users of the warehouse, our ability to source and deliver clinical data that is a believable approximation of the full phenotype of each subject will be the most important success factor in their evaluation of the data warehouse.

Epistemological Levels

As we analyze data sources for the data warehouse, the ontologies provide concepts for understanding what is being described in our data. Another independent perspective we also want to include in our analysis is epistemology, or the study of knowledge and justified belief. No matter how we classify our source data against the semantic ontologies, we'll still be left with a need to analyze what that data actually allow us to claim as knowledge or belief about the real world.

Observations

Observational data sources exhibit the strongest epistemology with regard to describing the clinical state of a patient. Vital signs are a quintessential example of observational data that users can typically trust as representative of the real

clinical world. Allowing for some measurement error or differences in desired precision, data sources that provide a patient's weight, height, temperature, and blood pressure are considered highly reliable in use. The differences in precision might matter a great deal in the way these data can be interpreted and used, but they are equivalent in the sense that they are about some aspect of the real world.

Both a patient entering their weight into a personal health record and a licensed practitioner measuring that patient's weight during a routine physical examination are measuring the same thing in the real world. The differences in the value of their observation might be attributable to device differences (i.e., precision of scale) or personal bias (i.e., wanting to weigh less), but both are real-world observations in terms of epistemology. Because these observations will potentially have many distinct values at the same time from different sources, these kinds of observations are modeled as facts into the data warehouse. Recognizing the differences, though, we want to analyze our data sources to provide as much provenance as we can in our dimensionalization of these facts. If all weight observations are dimensionalized to the provider, device, and clinical setting; then a user of the data can be as selective as required in choosing which observations to pay attention to. Observations from patients or their families might be given less query emphasis than observations recorded by staff providers in our own institution. Professional observations received from health information exchanges (HIEs), typically of less reliable or trusted provenance, might fall between these two extremes of trust.

These vital signs examples represent observational data that map into our OGMS medical ontology semantic layer. The data analyst always needs to epistemologically keep in mind that these observations aren't the real world; they are observations of that real world. As such, we don't know the patient's weight in an epistemological sense. What we have are observations of that weight that exhibit a provenance sufficient to justify a belief that the patient's real-world weight is what we have observed it to be. This distinction can be trivial when discussing an observation as simple as a patient's weight, so let's look at a more subtle example: radiology.

The warehouse load strategy will very likely include a data source that provides radiology results to the warehouse. The data analyst will analyze these data and most likely place the actual result findings in one of the fact tables. Very often, radiology results are sourced as textual reports that can be difficult to load; but that distinction isn't epistemological. Of interest here is that the result might include confirmation of a fractured right radius. If the provenance of the result includes a licensed staff radiologist, we'll likely consider that finding to be an observation. No further steps are likely to be taken to confirm the finding. We trust that the radiologist *observed* the fractured radius in an appropriate image of the patient's right arm. Observational data are taken to be data about the real world,

and we accept it in the warehouse trusting its face value. We always record as much provenance as we can in order to support future users of the data who might treat provenance as part of their query or usage criteria.

Hypotheses

Many data sources for the warehouse will contain data that doesn't conform to our expectations for observations of the real world, at least not necessarily in the ways that data are initially presented to us. For example, take the broken right radius example above. If the patient had presented stating that she or he had a broken arm, few would immediately interpret that statement as an actual observation of any real-world physiological state. Instead, the clinical process would investigate the statement, presenting both semantic and epistemological opportunities in our analysis.

A patient's *principal complaint* is a common data element that is analyzed very early in the data warehouse sourcing process because it is typically present in the early hospital encounter data that are often sourced very early in the project. It is a hypothesis regarding an appropriate diagnosis for the patient. As data analysts, we always need to conduct research on where the data value came from to ensure we are interpreting it correctly in our data load. I've seen principal complaint values that were clearly the words of the presenting patient, and I've seen principal complaints that were clearly the words of the person taking the patient's registration on arrival. The analyst should always research the procedures under which the data are collected to verify the likely provenance that should be recorded for the complaint fact. If the words come from the patient or family members, the fact is a principal complaint. If the words come from the registration staff, then they are a working arrival diagnosis. The two are not the same fact.

As a source data analyst, I typically operate on the working assumption that diagnoses don't come from patients or their families. Therefore, I start with an assumption that the principal complaint in my data source, if originating with the patient or family, is pathology and not diagnosis. If I have more descriptive data available in my source, I'll look for words and phrases that might indicate the signs or symptoms that were being expressed by the patient at the time. I might find data that mention potential causes of the injury, such as a workplace or sporting accident. While I'll likely treat the principal complaint as a warehouse fact, I'll model it as though it were an epistemological hypothesis.

For any fact representing hypothetical data, I'm interested in supplementing the fact as much as I can with additional details that will be helpful later in the clinical process as a caregiver attempts to confirm or refute the hypothesis. Note that, as analyst, I'm trying to understand exactly how the data are used in the clinical setting. I'm not suggesting that the warehouse be used in the clinical setting. I'm well aware that the entire process I'm trying to see in the data is over (i.e., it is *historical*) by the time our warehouse gets the data. The reason I always

conduct my analysis in terms of the clinical process is that I want the data to be used by people who have that clinical perspective. I'm less interested in how the data are actually stored in an HIT system because the clinicians aren't interested in that perspective, and sometimes that IT perspective is part of the problem that justified building the data warehouse in the first place. My perspective doesn't necessarily result in more or fewer facts in the warehouse, but it does typically expand the dimensional richness of those facts.

In the case of diagnostic hypotheses from patients, a lot of that data will end up dimensionalizing to the Pathology (e.g., pain, immobility) and Environment (e.g., workplace fall) dimensions. As always, I want provenance data to be part of that dimensionality, as well. With more serious presentations that might involve unconsciousness, the principal complaint can't be received from the patient. It is likely to be received from a family member, coworker, or EMT. Knowing the exact provenance of the hypothesis will help in its proper interpretation.

Eventually, the provenance of our data shifts toward professionals working within the clinical setting. A principle complaint becomes a working diagnosis, because it is indicative of the OGMS ontology that the clinical setting results in signs and symptoms turning into diagnosis of disorder. The provider's working diagnosis of broken right radius is a hypothesis that is justified by the signs, symptoms, and causative environmental factors already observed. Initiating a radiology order to image the patient's right arm is a correct action because there's a justifiable belief that it will confirm the hypothesis. At no point has the provider actually observed the fractured radius, so the diagnosis remains hypothetical until the radiology result actually confirms an observation.

Conclusions

Not all biomedical hypotheses can be confirmed by actual observation. Sometimes the provider must simply draw conclusions based on the available observations. The reasonableness of these conclusions, and therefore the extent to which they can be taken to be facts about the real world, will be dependent on the professional qualifications of the provider drawing the conclusions, and the availability of observational data to support those conclusions.

Most clinical diagnoses are ultimately conclusions reached by providers who are working through diagnostic hypotheses to rule out all but one definitive diagnosis. They are considered valid to the extent that we trust the expertise of the provider, or have access to the observational data that justified the conclusion. If we ensure that observational data are available in the data warehouse along with our diagnostic fact, then diagnostic conclusions drawn historically can be reevaluated against emerging practice.

This leads to an important epistemological principle for analyzing sources in a data warehouse: Presume that every fact needs to be traceable back to its

supporting observations. If our warehouse only stored actual observations then, given sufficient scope of data, we could recreate all warranted hypotheses and conclusions in our queries of those observations. In practice we won't reach that capability, but it's important for the data analyst to keep that objective in mind when analyzing the numerous conclusion data values throughout our source data.

A narrower example of a conclusion in clinical data would be the abnormalcy setting associated with a laboratory result fact. Whether a lab result is normal or abnormal is a conclusion drawn by the lab. It is not itself the result, or even part of the result. It is simply a quality of the result fact. No researcher would settle for receiving lab results for a population that indicated only whether or not the results were normal. They would want the *results!* It's fine to include the abnormalcy quality as part of the dimensionality of the lab result observation as long as we can get to the underlying observation. It can be interesting to query a data warehouse to see how many historically normal lab results would be considered normal by today's accepted reference range, or vice versa.

The lab result example is a simple case where it's actually very easy to store both the observation and the conclusion. A slightly more difficult example involves pain. Suppose we're asked to analyze a data source for the warehouse that includes Face, Legs, Activity, Cry, Consolability (FLACC) pain assessment scores for patients. They are clearly facts we want in the data warehouse, but they aren't observations. Each score is a conclusion by a provider after observing the patient along the five criteria measured by the score. We'll store them on that basis as facts, but we'll also try to analyze the source for observational data that might support the conclusion. Many times we won't be able to find any, and that's ultimately okay. If the score was entered by a competent professional properly trained in the use of the FLACC Scale, most researchers using the warehouse will accept the FLACC value at face value. We might later be able to correlate the fact against other facts, such as nursing notes or the patient's Twitter feed, to find phrases (e.g., squirming, uneasy, quivering) that support the FLACC conclusion.

The FLACC score example points to a general theme that the data analyst must keep in mind when analyzing data sources. All data values, the domain of which is any kind of Likert scale, are conclusions and not observations. Whenever a Likert-scaled value is encountered in a data source, we need to be on the analytical lookout for additional observational data that would support the conclusion represented by the scored value.

As we progress throughout analysis and modeling of data sources for the warehouse, the semantic and epistemology levels intersect. Observational data represent the real world and are typically mapped to OGMS concepts. Data about conclusions or hypotheses represent an interaction between patient and provider in the care setting, so map to concepts in the OBI ontology. These distinctions won't appear important to our warehouse users, but maintenance of

proper alignment among these levels will deliver data that are more consistent, complete, and trusted among that user community.

With an understanding of the various analysis heuristics that need to be included in the process of analyzing source data for the warehouse, including making implicit data explicit, properly understanding the semantic meaning of our data, and assessing the epistemological validity of our data; we're now ready to turn to the warehouse itself. What will the warehouse look like that we'll be sourcing all of these data into, and how will it conform to the model implied by these ontologies? The answer to that question is the generalized biomedical data warehouse design that will unfold in the chapters that follow.

Chapter 4

Biomedical Warehouse

The dimensional modeling paradigm described in Chapter 2 provides a mechanism for developing a flexible and robust data warehouse for any knowledge domain or industry sector. The source data analysis described in Chapter 3 provides an ontological basis for defining the requirements for a data warehouse in the biomedical arena. This chapter integrates those two perspectives to logically describe the actual dimensional data warehouse that my client healthcare and bioscience organizations implement. Chapter 5 will describe the physical implementation of this logical model.

I started building star schema data warehouses in 1979. We didn't call them star schemas yet, nor did we call them data warehouses, but they clearly fell within the dimension-fact paradigm that defines star schema warehouses today. My first data warehouse was a seven-dimensional star schema in the energy sector. Since 2006, I've worked primarily in healthcare and bioscience. The warehouses I build today have up to 26 dimensions, but bear remarkable similarities to my earliest implementations. The need to connect factual data to the dimensional contexts within which those facts occurred is older than all of the technologies on which we implement databases and systems today. The technologies are maturing, allowing us to meet more requirements than we were able to do in the past, but those requirements pre-date our efforts to meet them.

This chapter presents the data warehouse model that I currently implement in biomedicine: 1 fact table and 26 dimensions. The sections that follow provide descriptions of those various components. As you go through your design process, you might vary this model somewhat with slightly different dimensions or a variety of different or wider fact tables. I don't recommend such changes, but I see them all the time. Most of my clients who change this model end up changing it back. I urge you to leave the model alone for at least your Alpha-version development. The Alpha version time frame is measured in weeks, and that time frame won't be achieved if you start tinkering with this logical model. Build the Alpha and Beta versions using this model, and base any changes in your design on the experience you gain in the process. The Gamma version

offers opportunities to change the logical design, if needed. We'll discuss how to do so in Chapters 14 and 15, before building out the full data implementation in Chapter 16. My experience is that by then you will have effectively used features of this model that you might consider omitting or altering today. Keep the Alpha version simple, and you'll see that the generic nature of the Beta version will make many of the changes you'd consider today unnecessary.

Biomedical Star

Building a generic dimensional warehouse allows many design characteristics of the implementation to be determined by the dimensional model, without a lot of effort required by the development team. The main requirement considerations that need to be determined involve the choice of what dimensions to include, and how many different fact tables will be included. The single-fact-table 26-dimension model presented here provides an “out of the box” solution that allows your Alpha version to be built immediately without having to spend weeks or months determining your logical design. I always include the same dimensions in any biomedical star schema. They have emerged as my standard list after several implementations in which they worked well for mapping source data into the warehouse against the ontological considerations explored in the last chapter. My first healthcare data warehouse had 14 dimensions, and those dimensions worked well for covering the healthcare clinical data at that particular institution. As I expanded my work to include more bioinformatics or research settings, those 14 dimensions grew to the 26 dimensions presented here. In recent years, the disciplines of personalized and translational medicine have blurred the line between clinical care and research, making it much more difficult to differentiate care-oriented healthcare data from the research-oriented bioscience data. Knowing that data from either domain eventually integrate with data from the other domain, I now view this 26-dimensional biomedical model as my integrated baseline.

The 26 dimensions of the star design cover five different domains of interest ([Figure 4.1](#)). The database structure of each dimension is the same in terms of physical design, but the five domains serve different data administration purposes in how users tend to engage them in queries:

1. *Master*: The master dimensions represent the healthcare and bioscience domains about which data are recorded. Biomedical users recognize these dimensions as their primary areas of interest.
2. *Reference*: The reference dimensions represent the healthcare and bioscience domains in which facts are recorded. Those facts are typically about the entities defined in the master dimensions, while these reference dimensions define the biomedical meaning of those facts.

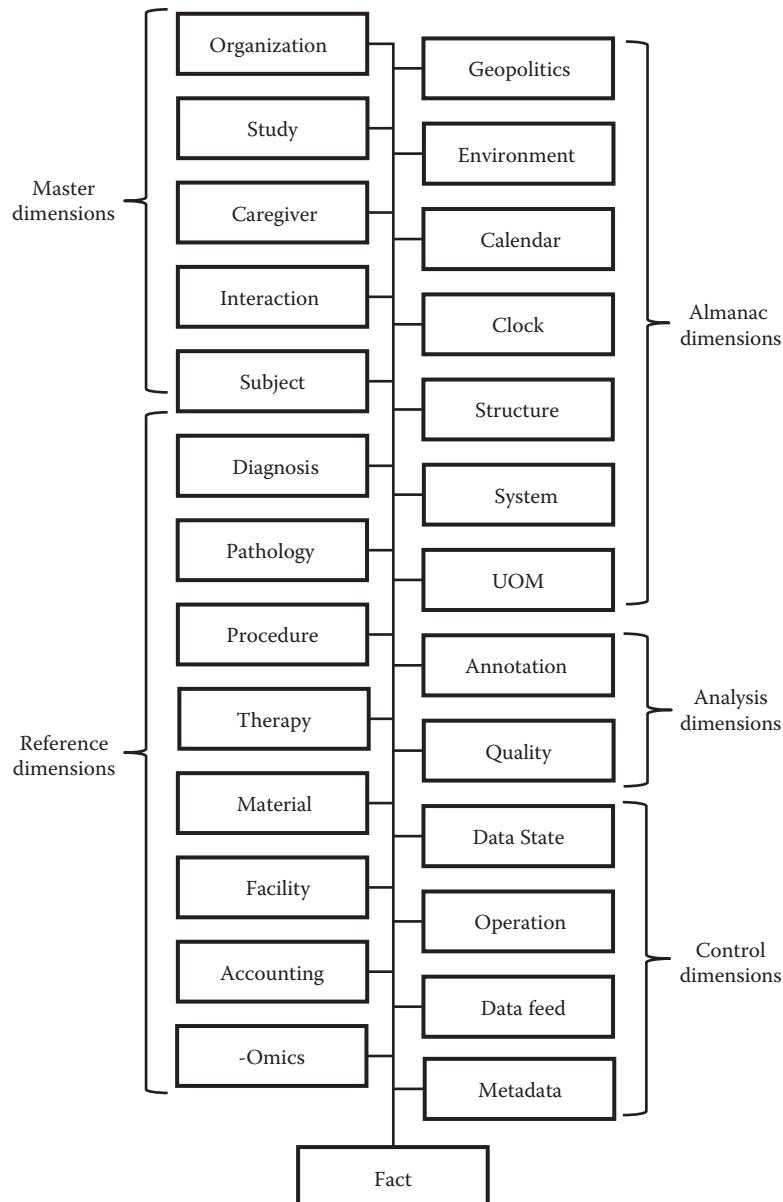


Figure 4.1 Data warehouse star model with 26 biomedical dimensions categorized into five domains.

3. *Almanac*: The almanac dimensions represent the data perspectives within which users often want to look at data, but that aren't typically defined by the institution itself. I think of it as data that I would look up in a reference source.
4. *Analysis*: The analysis dimensions get to the core of the data and their intended use, including qualitative aspects of the facts as well as curated annotations made against those facts.
5. *Control*: The control dimensions represent the data that drive the processing and inner control of the data warehouse itself.

All of these dimensions are equally available in the data warehouse for user queries. The distinctions among the different domains aren't about keeping any data within exclusive contexts. The way data can be viewed by a data administrator as master data or reference data was discussed in Chapter 2. The categories described here are consistent with that view. I'm not looking for warehouse users to become data administrators, but I do find that it helps to train users of the warehouse to differentiate these categories of dimensions in order to help them see what data are available and how they might use it. The distinctions are meant to offer guidance on usage, not to create formal boundaries.

Biomedical Facts

The Fact table in this biomedical warehouse has a single value column in it. Let me say that again: There's only one column in the entire warehouse that will actually contain the facts stored in the warehouse; it's the value column in the one and only Fact table. There will eventually be billions of rows in that table, and the vast majority of user queries of the warehouse will involve the fact table in some way. If you've worked with relational databases in healthcare, this notion will probably seem very strange. It takes time to get used to the star model. I emphasize the point because if you let relational thinking back into your design logic, you'll end up with a data warehouse that isn't a star, and isn't relational either. I've seen some very bad data warehouses end up somewhere in the middle. The two models are algorithmically interchangeable, as you'll see later, but for now let's commit to building a fairly pure star for your quick Alpha version.

Facts in the fact table will be the measures you collect in your data sourcing. Those measures might be quantitative (e.g., 100), or qualitative (e.g., Yellow), or simply logical (e.g., *exists*). Most of the data warehousing literature discusses facts as though they are always quantitative. Many of the tools for data warehousing are focused on manipulating and reporting those quantitative values in a variety of ways by numerically aggregating them across a number of analytical axes represented by the dimensions of the warehouse. You'll be doing a lot of that, but not as much as other industries or sectors might. For example, you won't be interested in the average systolic blood pressure the way a commercial manager might want to know the average order value. Your quantitative data are often more descriptive than transactional, and descriptive data don't fit the paradigm of cubes and reporting that some quantitative data fit.

The bigger difference in biomedicine is the preponderance of qualitative data, particularly in the form of unstructured textual information. Much of the data in healthcare are unstructured text, and unstructured text can be problematic when discussing data warehouses that are often intended to aggregate and report quantitative measures. You'll load textual data into your fact table in the same way you load any data, but you'll also end up using

additional tools to enhance those data and make them more useful to your warehouse users. You'll add curation processes and tools that help quantify the information contained in that qualitative data, making them available for more traditional query processing.

You'll also see many logical facts: facts that simply denote the existence of some relationship among the entries in our various dimensions. We sometimes refer to these facts as "factless" facts because there is nothing that needs to be placed in the value column of the Fact table. These facts exist only to connect the entries in the dimensions. In my experience, these logical facts end up outnumbering the quantitative and qualitative facts in the warehouse, sometimes by a very large margin.

The growth in logical facts is a by-product of our efforts to use semantic ontologies in order to better understand the contents of your data. Over time, patterns emerge in facts, particularly qualitative facts, causing you to recognize data being stored in facts that could actually be reconsidered as dimensional data. You might invent new subdimensions to support these semantics, moving the fact data out of the fact table and into that new subdimension. If you remove the last piece of qualitative data from a fact, you end up with a new logical fact where a qualitative fact used to be. This is part of the natural design progression of your data warehouse: Dimensionality increases as data in the Fact table decreases.

Many of the 26 dimensions described here have developed and evolved through this process of analyzing data in warehouses, looking for patterns that would indicate that one or more dimensions or subdimensions had been missed in a design. The process has stabilized fairly well for me in recent years in healthcare. I haven't added a new dimension in several years, yet I identify new subdimensions all the time. New subdimensions involve extending the *semantics* of the warehouse. I don't expect that process to ever end.

Master Dimensions

The five master dimensions in the warehouse—Organization, Study, Caregiver, Interaction, and Subject—are the dimensions of principal interest in the healthcare and biomedical research settings ([Figure 4.2](#)). These dimensions form the backbone of most strategies for querying data since all biomedical facts involve, or are about, entries in one or more of these dimensions. They are not all clinical since clinical data are only a subset of the data that might appear in a biomedical warehouse, but the core clinical dimensions are here. Most warehouse users will focus their queries here in the master dimensions, using the subsequent reference, almanac, analysis, and control dimensions to clarify and refine their queries. This category of dimensions is what sets the biomedical data warehouse apart from counterparts in other industrial or commercial sectors.

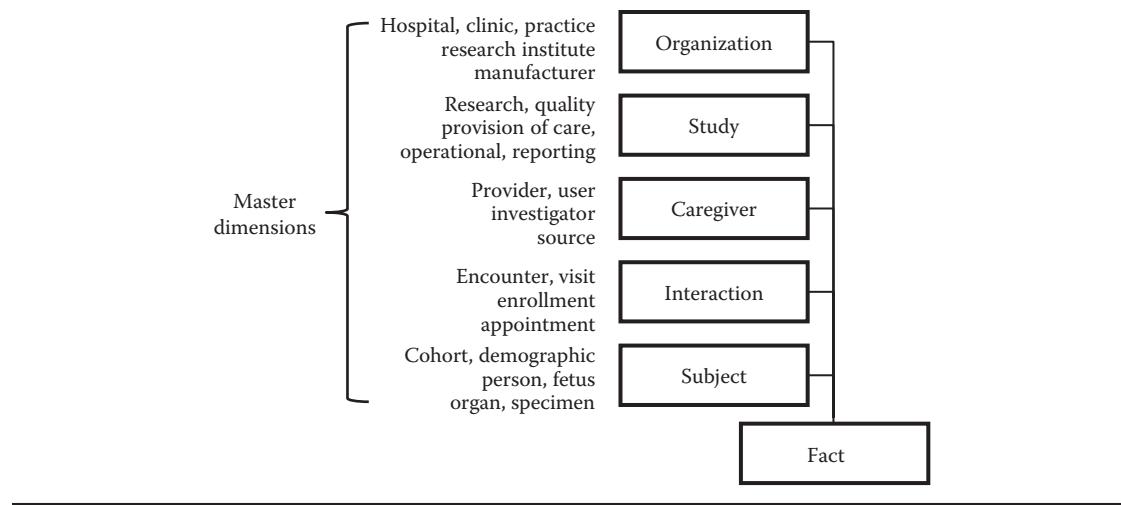


Figure 4.2 Master dimensions.

Organization Dimension

The Organization dimension defines the organizational settings in which relevant facts should be interpreted as having context in the warehouse. Organizations are typically areas of managed responsibility—as opposed to physical places—and can involve internal or external entities. Because many physical places are also areas of managed responsibility, there is a potential to confound elements of this dimension with related elements in the Facility dimension for internal organizations and facilities.

Because this dimension represents organizational entities that are both inside and outside of the enterprise, it tends to be conceptually very *wide* for the broad range of organization types that are relevant in the warehouse, and then tactically very *deep* for components of the internal enterprise organizations. Examples of the breadth of organization types and components populating this dimension include

- Hospitals, from small community to academic medical centers
- Nursing home and long-term care services
- Hospice/home care services
- Physician group practices
- Outside laboratories and service providers
- Ambulance and other transportation services
- Manufacturers and suppliers
- Operational and clinical departments
- Regulatory and professional organizations

Organizations and their components are connected through the dimension hierarchy, and can represent different levels of detail. Your organization might be a hospital with many dozens of departments, while a nearby hospital might be

defined with no departments at all, or perhaps only an Emergency Department. The permutations of these types of components provide the organizational context against which facts are stored. Examples might include:

- An admission to St. Mary's Hospital Cardiology Service from St. Luke's Emergency via transport by City Ambulance
- A discharge from Mt. Everest Oncology Department to Countryside Hospice
- A transfer from City Hospital Cardiology Service to Cincinnati Hospital

The Organization dimension, because of the management context that defines organizations, often serves as a data-ownership anchor point for data, with access security sometimes organized around this dimension when multi-organizational data will exist within the same warehouse.

Study Dimension

The Study dimension defines any investigatory or research settings against which data are being collected or interpreted. Healthcare enterprises that conduct research projects readily see a need for this dimension, but less research-oriented institutions are sometimes tempted to skip this one. I wouldn't. Even organizations that don't conduct research in the traditional sense still use data in ways that would constitute "studies" in the sense intended here.

For traditional research, every defined study can be placed in this dimension. Within each study, the dimension can also define the study's protocol variations over time. In this way, data generated within a study protocol can be dimensionalized to the appropriate entries. For data collected from research outside the institution, we can place a study entry for any particular source of data points being loaded. For example, if we source some quantitative data from a published appendix from an article we find in PubMed, we can use the Grant Identifier in the source data as a study entry in this dimension. In this way, our facts are always organized according to the study setting in which they were obtained.

It's true that the healthcare or clinical data typically associated with the data warehouse, particularly in the first year, are not considered study or research data. In those cases, the clinical data need not be dimensionalized against this dimension at all. However, there is an opportunity here to differentiate some data in ways that can prove helpful, but that might not fit the expected traditional mode of defining studies. It all depends on the full range of data that you anticipate loading into your warehouse.

If we slightly broaden our expectations of what constitutes research, we'll have entries that we want to include in this dimension. For example, I sometimes load a "Provision of Care" entry into the Study dimension so I can dimensionalize all of my clinical healthcare facts against that entry. This allows the source analyst to think about the data being sourced from healthcare systems in order to identify whether or not new facts actually represent the provision

of care in a clinical sense. A vital sign fact might get dimensionalized, while a room charge might not. Differentiation of these additional dimension entries doesn't diminish our use of these facts since the dimension is always there and available to every fact. It's common for data analysts to look for ways to add value to new facts by dimensionalizing them to additional dimensions.

In addition to enriching facts with some form of clinical versus research distinction, the Study dimension should also have a specific entry for every request that has been, or will be, submitted to the organization's Institutional Review Board (IRB) for approval to access data. In this sense, the dimension becomes important to the enterprise even if no traditional research is anticipated in the environment. Technically, anyone who wants to access identified or protected data should request IRB approval. That approval should be defined in this Study dimension. The entry is superfluous if the study generates no new facts, since the facts the study uses will have been dimensionalized in other ways. Depending upon the options selected for implementation in the Gamma phase of the workshop, entries in the Study dimension might still be needed for users who are not knowingly generating their own new facts. If the various audit capabilities of the warehouse are turned on during the Gamma version implementation of the warehouse, many queries that are nominally only reading data will actually be generating new audit facts in the background. Entries in this Study dimension will support the correct dimensionalization of those audit facts.

Just as the Organization dimension is typically part of the system access security for the warehouse, the Study dimension becomes part of that security and access control model, as well. The Organization dimension tends to identify the owner of the data, while the Study dimension tends to identify why those data are being accessed. Why data are being accessed is an important element in terms of whether certain protected or identifying information is made available, so this dimension is a key element of instituting data controls. You won't do anything special with it in the Alpha or Beta versions in terms of automated controls, but you'll begin mapping source data to studies, when known, in anticipation of using the data more in the Gamma version of the warehouse. In the meantime, the study linkages will be available in the earlier Beta version for standard querying.

Caregiver Dimension

The Caregiver dimension defines the human, mechanical, and categorical entities that provide care for, or make observations about, a patient in the clinical or research setting. Typical human caregivers are identifiable individuals who provide care to patients, including providers, nurses, case workers, or therapists. Mechanical caregivers are typically clinically engineered devices that monitor patients or dispense medications.

Categorical caregivers represent sources of clinical and observational data from individuals that are not otherwise thought of as caregivers in the clinical setting. These include parents, friends, and other individuals who might accompany a

patient into a clinical care setting. If detailed information is recorded about these individuals, it is typically not done in the Caregiver dimension. These people are much more likely to be stored in the Subject dimension, with the relationship between the patient and that person defined there.

For example, a set of pediatric allergy assessment facts might be stored dimensionalized to two caregivers: the physician who conducted the assessment, and the patient's mother who was present and reported the severity of any allergic reaction to that physician. The first of these two entries is to a dimension row that actually represents the physician as an individual person, while the second entry is to a categorical "Mother" row. Data about the mother, if known, are stored as a row in the Subject dimension with a hierarchy entry indicating the mother-son or mother-daughter relationship to the patient's row in the Subject dimension.

For many analysts, it is not at all obvious that a single Caregiver dimension should incorporate constructs as diverse as humans, devices, and categories; and very few analysts would intuitively grasp the connectedness without some effort. What matters in the decision to include all of this information in a single dimension is the context in which facts are to be placed. For example, a pain level fact might need to be dimensionalized to a caregiver who noted the pain level. That caregiver might be a nurse at the bedside asking the patient about pain, or a medication dispersal device recording a self-medicating event by the patient, or a visitor in the patient's room who mentions to the nurse or doctor that the patient has been complaining about pain. Regardless of the caregiver context, someone or something had to observe the pain in order for it to end up as a fact in the warehouse. We typically think of the Caregiver dimension in terms of the human professionals who are defined there. This is fine as long as the broader context is not ignored altogether. I view the devices and categories in this dimension as extensions of, or proxies for, the clinical professionals.

Interaction Dimension

The Interaction dimension defines the discernible interactions of a subject with the healthcare system. The classic interaction is the hospital encounter, so this dimension is often known as the Encounter dimension. Interactions occur at many levels of detail, and always contain an element of time because an interaction has some form of start and stop, even if unrealized at the time the data are captured.

Different healthcare enterprises will focus attention on interactions at different levels of detail. In a hospital inpatient setting, a patient encounter generally starts with an admission and ends with a discharge, with the intervening time known as the length-of-stay. During that encounter, a nurse or other professional might visit a patient dozens of times per day, yet those visits are typically not recorded explicitly in the EHR or data warehouse. Many of those visits can be inferred from the data because orders occur, vital signs are taken, or medications are administered, implying that visits are occurring. In the home care sector,

the emphasis on interactions is very different. While each encounter still starts with an admission and ends with a discharge, that is not where the data focus resides. In home care, each individual visit to the patient is a recorded event around which a great deal of analysis is eventually done. The home visit is much more explicit than in the hospital inpatient setting.

Ambulatory care represents something between the inpatient and the home care extremes. In many outpatient settings, the encounter and visit are largely synonymous because the entire interaction occurs in that one interaction. In this setting, the admission concept is less applicable. Instead, the concepts of registration and arrival are used, and departures serve the purpose of discharges. If the outpatient care is hospital-based, the interaction will usually be viewed as an encounter. In a physician practice, the same interaction will be viewed as an office visit. The hospital might label the arrival as an admission, because that's what their EMR or EHR calls it.

This diversity of duration and grain on interactions is seamlessly handled by the Interaction dimension because both Encounter and Visit are subdimensions of the dimension. For any particular array of facts, the warehouse supports the use of encounter data alone, visit data alone, or both together. This dimension also supports other less-used interaction constructs, such as appointments that might eventually be realized as encounters or visits but might remain unrealized if they don't happen. It also supports research studies into which patients might enroll. Whatever construct involves a patient interacting with the enterprise for a short or extended period of time can be defined within this Interaction dimension.

Entries in the Interaction dimension constitute containers within which we pour all of our related data. They are ontological contexts that we classify and track, but their purpose is primarily to provide a place to put all of the various data elements that we associate with the aggregation of process activities that makes them up. It's the data within and across these interactions that interest us much more than the interactions themselves.

Subject Dimension

In the healthcare or bioinformatics setting, the subject of interest is typically a cohort, a person, or a specimen. These three constructs share commonalities at three different grains: persons are components of cohorts, and specimens are components of persons. Biomedical facts involving all three are captured and stored, and many queries aggregate data across these three grains. The Subject dimension is often referred to as the Patient dimension, but the three different grains of data make that name less accurate. In research settings, not all subjects of interest would necessarily be referred to as patients (e.g., a lab animal). Even in the clinical care setting, not all specimens would necessarily be associated with someone identified as a patient (e.g., a unit of blood from the Blood Bank). The use of "patient" to identify this dimension is acceptable in common usage as long as the multi-grained distinction isn't lost.

The use of the Subject dimension in defining facts is typically very simple because most facts refer to only one subject entry. However, for certain tissue typing, blood banking, and transplant clinical facts, multi-subject groups are common for aligning donor and recipient roles. In Obstetrics, as well as labor and delivery, multiple subject entries are often required for representing maternal, fetal, and newborn data. Fetal entries (special cases of specimen) typically become independent person entries after birth, inheriting all clinical data previously associated with the maternal parent specimen. Operationally, certain admission-discharge-transfer (ADT) transactions (e.g., A34 patient merge) require referencing multiple person entries.

The data in the Subject dimension present unique challenges because of the diverse relationships among the Person and Specimen subdimensions. Transplant events present opportunities to see a dependent specimen (i.e., the donor's organ) temporarily become an independent specimen (i.e., the harvested organ), before becoming a dependent specimen again (i.e., the transplanted organ) in the recipient. Certain clinical and bioinformatics facts associated with the donor patient become associated with the recipient patient as a result of the transplant.

Object Instantiation

By allowing definitions at the person and specimen level, the Subject dimension, supports a data trend in healthcare that will become increasingly important in the coming years. The disciplines of medical and clinical informatics are moving definitively toward *instantiation* of objects. This is seen most immediately in the significant difference between Health Level 7 (HL7) 2.3 version and its replacement RIM-based HL7 3.0 version.

HL7 3.0 supports instantiating the objects about which data are being communicated. For example, under HL7 2.3, if two different lab results are received for a patient against different lab orders, there isn't necessarily a way to tell whether the tests were run on one single, or two different, blood samples. In another example, if a patient presents with a broken left arm in two appointments 6 months apart, there isn't a way under HL7 2.3 to know whether it is the same broken arm presented twice, or two different arm breaks. Under HL7 3.0, the specimens themselves (e.g., drawn blood, broken arm) have identifiers, and results are reported against the specimen rather than against the patient.

Rather than two results pointing to the patient, we will have each result pointing to the appropriate specimen, with the specimen relating to the patient in a natural hierarchy within the dimension itself. The transition to this form of data instantiation takes considerable time, requiring changes to take place in many clinical systems across the enterprise, and eventually in the data feed to warehouse. In the meantime, users will see an increase in data facts related to Accession Numbers (a temporary fix until systems mature further) on collected specimens and lab results.

Reference Dimensions

The eight reference dimensions in the warehouse—Diagnosis, Pathology, Procedure, Treatment, Material, Facility, Accounting, and -Omics—define the various information of interest within the various facts stored about the master data elements (Figure 4.3). The master dimensions provide the context for facts: Subjects interacting with organizations through caregivers and studies. The details of those interactions are captured through a series of reference dimensions that define the details. To the extent that the Interaction dimension is seen as a container into which biomedical information is placed, these reference dimensions define that information.

Diagnosis Dimension

The Diagnosis dimension defines the results of the diagnostic process in which a provider treating a patient draws conclusions about the condition of the patient, and the resulting care needed by that patient as a result of their diagnoses. Each diagnosis is some form of conclusion drawn by a provider, with different levels of confidence and specificity depending upon where the data are generated in the clinical process. Each entry is an interpretation of some collection of findings and observations.

Most of the entries we place in the Diagnosis dimension will consist of disease and syndrome diagnoses that a user would traditionally think of when looking at the dimension. A diagnosis might be a disease that is associated with a group of

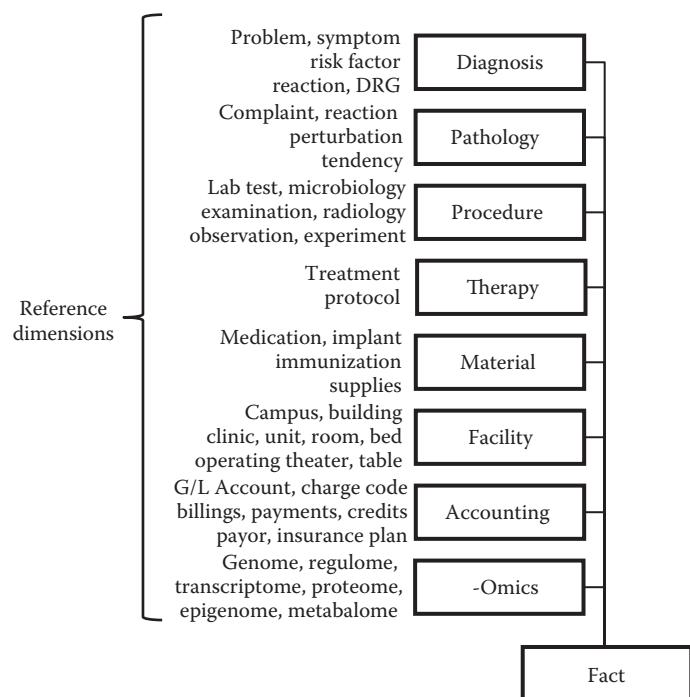


Figure 4.3 Reference dimensions.

pathologies after a series of formal or informal diagnostic interactions between a diagnosing caregiver and the patient. They might represent failures of anatomical organs specifically, or of one or more biological processes generally. Many diseases are defined by their environmental origins, either as communicable or environmentally induced. Some broadly encompassing diagnoses are referred to as syndromes. Diagnosis data and coding structures used widely across the healthcare sector include

- International Classification of Diseases (ICD), whether ICD-9, ICD-10, or ICD-11
- Systematized Nomenclature of Medicine (SNOMED)
- Diagnosis-Related Group (DRG)
- Human Disease Ontology (HDO)

A diagnosis in the data warehouse is typically used as the dimensionalization of a caregiver's conclusion, whether speculative, working, preliminary, or final. Conclusions represent a summarization of the diagnostic process, and are often used in an organization for financial billing and reimbursement purposes. This clinical and financial use of the diagnostic construct can create friction between clinical, operational, financial, and research users of the warehouse because of their differing perceptions of the accuracy and efficacy of any diagnostic conclusions. As a result, researchers often prefer to rely more on detailed pathology and lab data to draw their own inferences in order to avoid any financial reimbursement maximization bias built into the clinical diagnostic data.

This dimension also includes other less traditional diagnostic data that fit our ontological expectations that every entry should be an interpretation or assertion by a provider, based on an analysis of findings. A prognosis is an interpretation made by a provider that would be placed in the Diagnosis dimension. If your organization uses a fixed list of prognosis codes in its clinical system, each possible prognosis could be present in the Diagnosis dimension, against which facts could be dimensionalized. Alternatively, a prognosis might be available as unstructured text in some data sources, so a single row for "Prognosis" might be included in this dimension to anchor those textual facts as diagnostic interpretations.

Pathology Dimension

The Pathology dimension defines the actual clinical data recorded about a patient during the diagnostic or other observational process. While the Diagnosis dimension was interpretive, the Pathology dimension is based on observation. Most of the entries in the Pathology dimension will consist of observable manifestations that we would traditionally think of when discussing pathologies. These include anatomical or functional manifestations of disease or condition

that can be reported or observed in patients. These are typically abnormalities of one or more system components of the patient, such as abdominal pain or headache. Patients experience their symptoms (i.e., pathologies), not their diagnoses. A patient presenting at the emergency department is there to describe her or his pathologies.

The Pathology dimension is often confounded with the Diagnosis dimension, and analytical attention is required to assure that these two dimensions are implemented correctly and consistently. There are distinct entries in each dimension, and there are definitely areas of overlap. For those concepts that overlap, the correct choice for dimensionalization of facts depends upon the semantic meaning and intent of the facts themselves, not the actual anomaly being recorded.

This confounding can be readily observed in one of the most commonly used coding schemes for disease: the ICD-9 codes. ICD-9 includes an entire category of codes that describe “Diseases and Injuries” (001-999.99). That category of disease codings gets loaded into the Diagnosis dimension because it represents what providers are looking for in the diagnostic process: Interpretations of the findings that justify the specifically coded diagnosis. That main category includes a subset of codes that represents “Symptoms and Signs” (780-799.99) that a provider might place on a patient’s problem list, but wouldn’t necessarily be asserted as conclusive diagnoses. This subset belongs here in the Pathology dimension. Another category of codes in the ICD-9 structure that belongs here in the Pathology dimension is the set of codes that represents health propensities: the “Factors Influencing Health Status” (V01-V91.99). These so-called “V” codes represent observed factors that could influence the diagnostic process, so they belong here in the Pathology dimension.

Figure 4.4 illustrates an example of this dimensional overlap. A patient presents complaining of unspecified insomnia and headache. These two pathologies are included in ICD-9 as codes 784.0 and 780.52, respectively. The physician interprets the patient’s symptoms and considers other factors to make a diagnosis of chronic cluster headache (339.02). According to our ontological criteria for analyzing data sources, I would dimensionalize the 784.0 headache observation against the Pathology dimension because it presented as a complaint of the patient when describing symptoms. While in other cases the 784.0 code might be used for recording a diagnosis, in this example it is clearly pathology.

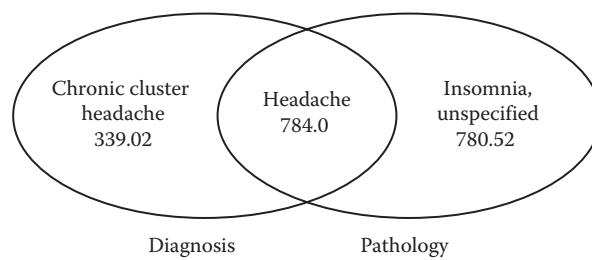


Figure 4.4 Overlap of diagnosis and pathology concepts.

Generally, the ICD-9 codes classified as Symptoms and Signs should dimensionalize to pathology over diagnosis, but since we can't control what happens in our clinical source systems, our safest approach to loading dimensions is to put all of the ICD-9 codes in each of the dimensions where our source clinical systems might use them. In the extreme, this would actually mean that the diagnosis and pathology dimensions would even contain the ICD-9 procedure codes. There's no harm in including these codes, as long as they are never used in those contexts. This is a specific example of a general principle that applies when defining dimensions and mapping data into them: Load whatever data might be needed in a dimension, even if some usage of that data wouldn't make complete sense. It's not the job of the data warehouse to enforce constraints upon our source data. The source data are what it is. We can't dictate whether a complaint of a headache should be recorded as a diagnosis or pathology. We should allow for either, and use queries to evaluate usage and best practice.

Using this principle, I load all ICD-9 codes into the diagnosis and pathology dimensions. In the Gamma version, we'll implement some controls that will help us monitor the usage of such a wide codeset that we already know goes beyond the domain we want in each of the dimensions. If we load all of the ICD-9 codes to the Diagnosis dimension, the list will include the Symptoms and Signs, all of the propensities ("V" codes), and even the ICD-9 procedure codes. Resist the temptation not to load some of these because they won't be used. You'll clearly see symptoms in your diagnosis data, and you'll eventually see a procedure code in your diagnosis list. Omitting the data in the dimension would be your attempt to enforce a rule that your clinical systems aren't enforcing, and it will fail. Load all of the data. The Gamma controls (Chapter 15) will allow you to indicate that the use of some of those codes would be unexpected (diagnoses) and others would be undesired (procedures). When the data are eventually used in those contexts, the ETL jobs will report that usage automatically, using the Gamma version controls. Users can then query the data itself to see what's happening, and take any desired corrective or improvement actions at that time.

Another source of confounding between diagnosis and pathology is what typically occurs in clinical settings where free text is used to record observations and histories from a patient, where the text ends up containing the actual words that were used by both a patient and a provider. In such cases, the use of text rather than coded values might create ambiguities.

I try to map the data that I analyze based on the context within which they were collected rather than relying on any actual inspection of the data itself. For example, if a data value arrives that says "broken arm," it needs to be interpreted. If the data value was collected as a principal complaint from a patient arriving at the emergency department, then I know I am dealing with pathology. If the data value was collected while taking a patient history, then I am likely dealing with a historical diagnosis. As a pathology, I know "broken arm" is actually a proxy for pain, swelling, and lack of mobility; most likely a result of a specific

and recognized trauma that resulted in the patient presenting. The patient might or might not have a fractured arm, but that diagnosis must await the actual diagnostic process.

As a historical diagnosis, “broken arm” is vernacular for some specific fracture diagnosis that occurred in the past, for which the patient doesn’t have more accurate or specific terminology. I’ll record the historical diagnosis as a fact, and make an effort to map the reported history to some specific diagnosis if the data are available. For instance, if a patient reports that she or he broke an arm 2 years ago, and our EHR shows treatment for a fractured radius 27 months ago, then the historical fact is likely aligned with that previous clinical diagnosis.

In addition to setting context, these analytical distinctions often take advantage of the differences between professional and vernacular language. Expressions like “broken arm” have no specific meaning in the diagnostic context: Bones are fractured, not broken. Patients and providers don’t commonly use the same expressions to describe clinical facts, so the correct dimensional context for many entries can be inferred by simply looking at the data values. It’s not fool-proof (e.g., “broken arm” can certainly show up in a radiology report once in a while) but, combined with the clinical setting, the distinction is typically quite clear.

One circumstance where I can be sure I’m sourcing pathology over diagnosis is any setting where I know I am recording the words of a patient or one of the patient’s caregivers. These might arrive because we’re receiving e-mails from a caregiver while treating a patient at home, we’re receiving data from the patient’s personal health record, or we’re monitoring a patient’s Facebook or Twitter feeds for clinical or behavioral content. In settings where we know we’re seeing the words of laypeople, we always dimensionalize the observations as pathology. In this context, it’s safe to say that recorded pathologies are used in the diagnostic process, but they aren’t the diagnoses themselves.

Procedure Dimension

The Procedure dimension defines the procedures or observations that provide context for the relevant facts. Procedures include anything that can be done or observed, and is typically a broader concept that might be inferred from the dimension name itself. Procedures constitute the *verbs* of the warehouse, indicating things that can be ordered, performed, and resulted on behalf of a patient or research subject. Typical procedures include

- Laboratory tests with appropriate reference ranges
- Radiology exams
- Microbiology tests
- Vital signs
- Observations and notes
- Questionnaires and surveys

These procedures are often sourced using one or more of several coding schema associated with procedural data, including

- CPT4 procedures, with modifiers
- ICD-9 procedures
- Logical Observation Identifiers Names and Codes (LOINC)

Procedures represent a wealth of information that can be analyzed when sourcing new data into the warehouse. More so than in any other dimension, the list of procedures often embeds information that would be mapped to other dimensions of the warehouse, if known. In the last chapter, we saw the opportunity to map medication-related procedures to their associated medications (e.g., Assay of Acetaminophen [CPT 82003] to Acetaminophen). Cross-mapping increases the dimensional richness of related facts, allowing correct queries to be built from perspectives that wouldn't be supported by the source systems that rely exclusively on their procedure codes.

Another example includes anatomical references. Many procedures are specific to anatomical entities that are often only identified in the names of the procedures themselves. The CPT 21499 procedure for “Unlisted musculoskeletal procedure, head” is associated with the head of the patient. The CPT 21070 procedure for “Mandibular coronoidectomy” is associated with the mandible. The source system from which these data are received probably doesn't make any of the anatomical connectivity explicit in the data. Here in the warehouse, we'll recognize that distinction in order to ensure that facts associated with these procedures are also dimensionalized to the head or mandible (in the Anatomy dimension). Warehouse queries using anatomy as the selection criteria should be able to find the data for these procedures without the user having to know anything in particular about the procedures themselves.

Treatment Dimension

The Treatment dimension extends the notion of procedure into the details of the clinical and research activities that are associated with patients, but are not done on or to patients. Samples and data derived from procedures are treated in different ways in their handling and processing. This dimension allows those details to be dimensionalized directly against the facts affected by these treatments. Whatever details need to be recorded about something dimensionalized to a procedure can be recorded in this dimension whenever the procedure itself doesn't supply those details. These details are common in lab work and are often recorded in research. For example, blood samples might be centrifuged before processing, or two microarray samples might be subjected to different heat treatments. These treatments don't alter the definition of the underlying procedures but can enrich the dimensionalization

of related facts; offering users an alternative query path based on details not captured in the Procedure dimension.

Sometimes, the procedure in question is being performed on the patient directly rather than to a sample or specimen. In those cases, treatments are those qualifiers that need to be recorded in order to correctly interpret the data. For example, recorded vital signs (procedure) will be interpreted differently based on whether the patient was standing or sitting (treatments). The distinctions drawn in these cases of treating the patient directly are semantically the same as when we treat samples, but the language adopted in such a discussion will seem a bit less natural to users of the data, particularly research users who are accustomed to recording sample treatments.

Material Dimension

The Material dimension defines the materials that are used in the conducting of clinical and research activities. Typical materials include

- Drugs, immunizations, and other pharmaceuticals
- Surgical and other medical implants
- Medical devices and equipment, including eyewear
- Supplies and consumables
- Allergens and environmental factors

Materials often need to be tracked through the various systems in which they are recorded, so the Material dimension includes controls for tracking instances. An instance of material is trackable in the dimension by lot number, serial number, or both (Figure 4.5).

For those material entries that need to have lot number or serial number controls, the ETL jobs for the warehouse can automatically maintain the needed entries in the dimension for those lot numbers and serial numbers without

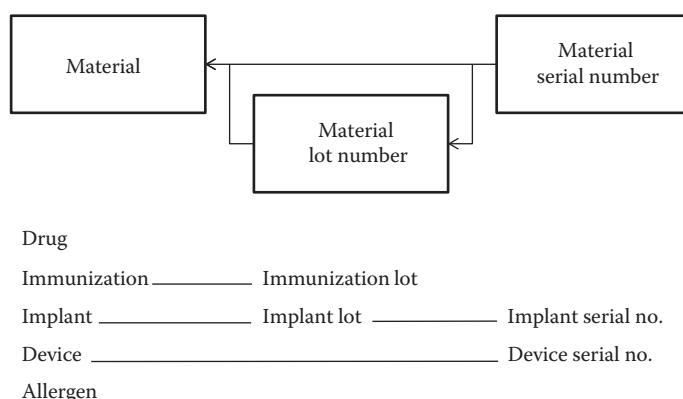


Figure 4.5 Two-level controls for tracking instances of materials.

needing separate source data feeds implemented for their maintenance. Key internal control properties include the following:

- *Lot Number Control Flag*: If *No*, then lot numbers are not being tracked for the Material row. If *Yes*, then lot numbers will be tracked, when available in the data source.
- *Lot Number*: Each distinct lot number results in a separate row in the material dimension, with the “master” row for the material having a NULL lot number. Lot numbers can be used with or without serial numbers.
- *Serial Number Control Flag*: If *No*, then serial numbers are not being tracked for the material row. If *Yes*, then serial numbers will be tracked, when available in the data source.
- *Serial Number*: Each distinct serial number results in a separate row in the material dimension, with the “master” row for the material having a NULL serial number. Serial numbers can be used with or without lot numbers.

This feature entails recognition of multiple keys structures in the warehouse for any controlled source key being defined (e.g., Item, Item-Lot, Item-Serial, Item-Lot-Serial). While I’m describing it here as a component of the Material dimension, it can actually be implemented in any of the warehouse dimensions where this level of instance control is needed. The implementation of these controls is typically deferred until the Gamma version. The use of lot number and serial number controls can result in certain logical errors occurring within the data, and the control features of the warehouse will flag the inconsistencies as they occur. The data will always load successfully, but certain conditions could trigger a warning based on which controls are turned on. Data that arrive without the expected lot numbers or serial numbers will be stored, but warnings will be issued regarding those missing control values.

Facility Dimension

The Facility dimension defines the physical settings in which actions take place. Facilities are typically places where care is delivered, as opposed to elements in the organization chart. Highly decomposable and hierachic, the most deeply grained facilities are places where a patient can receive a portion of her or his expected care or treatment, such as a bed, chair, or table. While most facilities are directly involved in patient care or treatment, many facilities serve ancillary purposes and are not necessarily places that patients would visit, such as a lab or research location. Typical facilities of interest include

- Campuses and buildings
- Nursing or ancillary units, clinics, or labs
- Residential and examination rooms and suites

- Patient beds, tables, chairs
- Transport vehicles, such as ambulances or gurneys
- Research and clinical laboratories

Facilities are always physical places, and should not be confounded with the areas of management responsibility represented by the Organization dimension. While an overall enterprise might have a strong relationship between its organizations and facilities, the data warehouse dimensionality shouldn't try to enforce that relationship. I've seen cases where a piece of data wouldn't load into the warehouse because the data for an oncology patient was assigned a room in a cardiac unit. The mismatch between the organization responsible for the patient and the assigned purpose of the facility couldn't be supported within a single combined dimension. As with every data rule that nominally appears to restrict valid combinations of data, we want the warehouse to be able to store anything it receives. If the source systems don't enforce recognized rules, the warehouse shouldn't try to. However, the data warehouse can serve as a place to run queries specifically designed to identify and report those policy exceptions.

Most of the different subdimensions in the Facility dimension are maintained as actual instances of their types. This means that there are entries for every bed, room, or nursing unit throughout the hierarchy of facilities. The notable exception tends to be the transporter subdimension, which is typically defined as a typing subdimension, meaning that the subdimension is usually loaded with a simple list of types of transport (e.g., ambulance, wheelchair, gurney, hospital bed). If for instance data are required for transport facilities, it can be satisfied by sourcing the instances themselves, or by using the lot-serial control feature for dimensional control.

Accounting Dimension

The Accounting dimension defines the accounts, accounting structures, and financial entities and instruments that provide financial control and context in the warehouse. Typical accounting dimension entries include

- General ledger accounts and cost centers
- Charge and billing codes
- Reimbursements, payments, and adjustments
- Payers, insurance plans, and contracts

The level to which these subdimensions need to be defined and loaded depends upon the overall objectives of the warehouse within the enterprise. Some would argue that financial data don't belong in a biomedical data warehouse that is more focused on clinical and research data, and that financials should be treated as outside of that scope. I disagree with that notion because the segmentation of data between financial and clinical can limit certain queries. Many times, I've

seen financial data used as a proxy for missing clinical data. There have been situations where I wouldn't have known about certain administered medications if the financial charges hadn't been loaded. I've seen improvement projects use the data warehouse to find mismatches between clinical data and related financial transactions to identify gaps that were costing millions of dollars. I've even seen encounters in remote clinics that wouldn't have been in the warehouse at all if the billing data hadn't been sourced into the warehouse.

I want financial data in the warehouse. Sometimes it is all I have to visualize what happened with a patient. Other times, the differences between that financial data and my clinical data will point to process problems that can be addressed. The ability to combine clinical, operational, and financial data in one place, often in ways that no other system in the enterprise can do, is among the most valuable services offered by the data warehouse.

Achieving the goal of using financial data as a proxy for missing clinical data requires some extra effort on the part of the data analysts who work on the sourcing of that data. Just as descriptions of entries in the Procedure dimension contained information about materials that wasn't otherwise contained in facts related to those procedures, you can expect to have similar cross-dimensional opportunities presented in data in the Accounting dimension. For example, a data source that provides the warehouse with financial charges against an account will typically contain cost codes that will be defined in this dimension. The descriptions of those charges might be our only mechanism for identifying the procedures or materials against which those charges have occurred. By analyzing these data, the facts representing those financial charges can be cross-dimensionalized to the appropriate entries in the Procedure and Material dimensions so users of the warehouse will not need to understand the structure and content of the enterprise's cost code master data in order to query information about the use of those procedures and materials.

-Omics Dimension

The -Omics dimension captures the wide and diverse range of data used in bioinformatics to cross-match and analyze data arising from clinical practice and research. Of the 26 dimensions in this design, the -Omics dimension is the most recent addition. As research brings bioinformatics data closer to the point of care through personalized and translational medicine, I'm seeing this dimension starting to be used to dimensionalize facts from microarray and associations studies. Entries in this dimension have, so far, included genomic components (e.g., LOC130951 on 2p15-p11) or proteomic elements (e.g., CRX) that are being associated through a research fact (e.g., Bayesian predictor score) to other clinical elements in this design, usually in the pathology (e.g., night blindness), diagnosis (e.g., retinitis pigmentosa), structure (e.g., eyes), or system (e.g., human) dimensions. I've also seen increasing use of metabolomic data in diabetes research settings. These facts are often only loaded into the warehouse against a cohort of patients

associated with the organization's biobank or clinical research settings, but I expect to see wider and deeper use of this dimension in the future as the bioinformatics sciences further penetrate the areas of clinical practice.

Almanac Dimensions

The seven almanac dimensions—Geopolitics, Environment, Calendar, Clock, Structure, System, and Unit of Measure (UOM)—provide geographic, temporal, scientific, and structural grounding for the facts in the data warehouse (Figure 4.6).

Geopolitics Dimension

The Geopolitics dimension defines the definition and decomposition of geography according to geopolitical boundaries. Although it seems simple, this dimension often introduces complexity (particularly when a high level of rigor is desired) because not all geopolitical entries align themselves in the ways that we might expect based only on a vernacular view. Geopolitical entities of interest in the warehouse include:

- Country, kingdom, or territory
- Subcountry (i.e., State, Province, Canton, Prefecture)
- County, parish, shire (unique within Country–Subcountry)
- City, town, municipality (unique within Country–Subcountry)
- Community, neighborhood (unique within City)
- Site (i.e., Street w/Number) (unique within City)
- Address (i.e., Apartment, Suite, Floor) (unique within Site)
- Postal Zone (unique within Country)

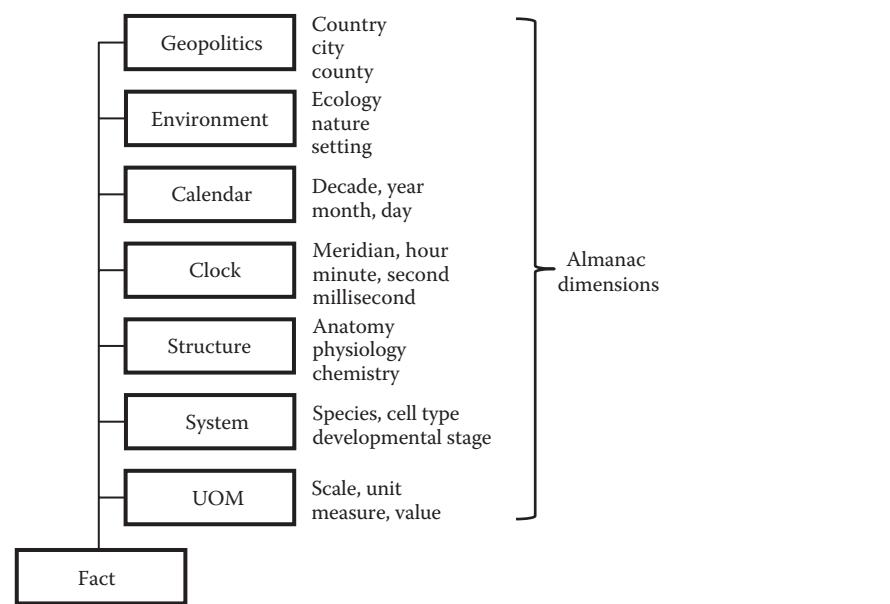


Figure 4.6 Almanac dimensions.

One of the complexities of discussing geopolitical breakdowns is that different geopolitical units are known by different names in different parts of the world. For example, in the United States, subcountries are called states; in Canada, they are called provinces; and in Switzerland, they are called cantons. To make matters worse, the areas referred to as states in the United States aren't even all technically states: The District of Columbia is always on a list of states in the United States, even though it's not a state at all. (Addresses in the District of Columbia often include the "DC" as part of the name of the city and leave the state designation NULL.) The same is true of numerous other districts and territories that end up on the list of valid states in the United States. For example, Guam and Puerto Rico are often listed as parts of the United States in sourced state tables, although they often also appear in sourced country tables. The Canal Zone stopped being a United States territory in 1979, but still often appears in sourced state lists. I've seen Newfoundland and Labrador listed as separate provinces in Canada, often alongside the correct single-province definition. Yukon province is often also present in a data sources as Yukon Territory.

The problem continues as we navigate down the hierarchy in this dimension. Subcountries (states) are made up of counties and cities. For example, in the United States, cities and counties sometimes don't exist in hierarchical relationships. Most people expect a city to be within a county, but there are exceptions where a county exists within a city (e.g., New York City consists of five boroughs, each of which is a New York State county). Even when discussing cities, we must be aware of the fact that lots of geopolitical cities are made up of a host of less formally defined communities by which certain areas of the cities are known. Sometimes postal regulations allow these communities to be used in addresses; and other times not. The rules tend to include whether the community has its own designated postal code within the city.

[Figure 4.7](#) illustrates the subdimension hierarchy as I implement it today in the warehouse. The relationship among the subdimensions isn't quite as linear as we might expect from personal experience. I call the Subcountry dimension State, not because it's accurate but because I've spent years trying to get people to call it Subcountry and I always fail. With apologies to my non-US clients, I give in and call it State. I haven't given in on the Postal Code though: I simply won't call it a Zip Code.

While the design of this dimension includes enough detailed subdimensions to eventually load individual personal or organizational addresses into the dimension, I typically don't do so because patient addresses are HIPAA-protected information. Defining a row in the dimension for every captured address for every possible patient would take a lot of resource just to store a lot of detail that can't be referenced by anyone using a deidentified view of the warehouse. Instead of loading addresses into the dimension, I usually place patient addresses, when sourced, into the Fact table. I dimensionalize those particular facts to the subdimensions in the Geopolitics dimension which I can use even in a deidentified view, usually as City, County, and Postal Code. Knowing these

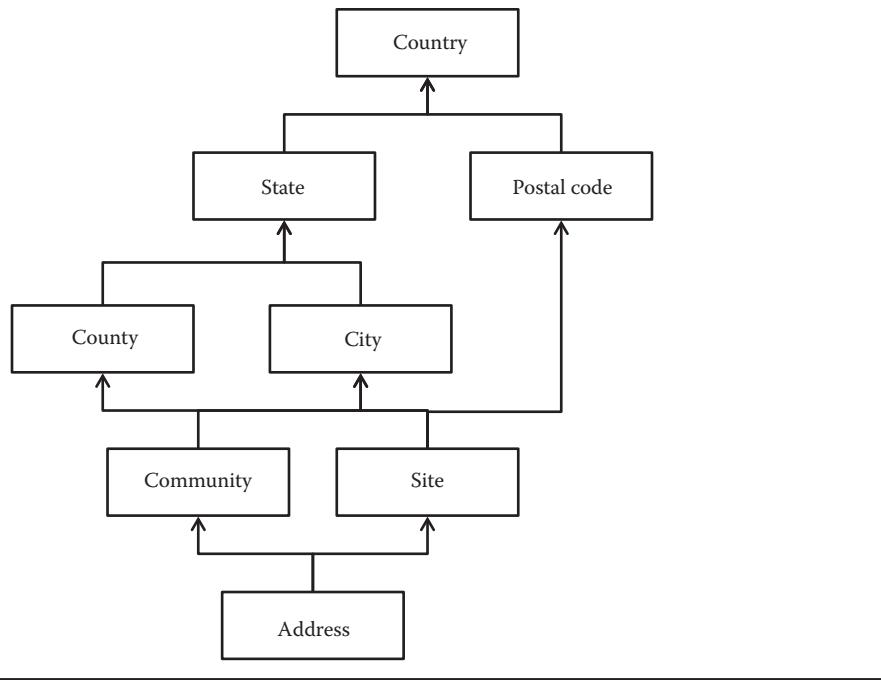


Figure 4.7 Geopolitics hierarchy.

geopolitical components is often sufficient for most epidemiological queries against the warehouse, and users with IRB approval can retrieve the full address from the Fact table.

Because geopolitical data are often among the messiest data in the healthcare data warehouse, design of the natural key structures in this dimension needs to minimize any brittleness caused by bad data. Addresses are typically self-reported by patients, and often end up spelled inconsistently by staff members who enter those addresses into clinical systems during patient registration. There are excellent tools available for cleaning up these data, but the tools are rarely incorporated into today's clinical EMR and EHR systems. Properly and flexibly designed, most bad data can be cleaned up and improved in the warehouse after loading, so clean-up does not interfere with any ability to get facts loaded every day.

Data clean-up is also aided by not placing full addresses into the dimension. It's much easier to clean-up data related to cities, counties, and postal codes than to have to clean up entire addresses with street names and numbers, as well as apartment or floor numbers. The messiest data go in the Fact table and don't need to be able to be cleaned up quickly. Different variations in spelling or layout of address data, even from the same patient, don't result in a lot of extra messy dimension rows that have to be managed later.

One way to think about addresses is to recognize them as free-text data strings. A general rule across this warehouse design is to presume that all free-text strings are patient identifying, and therefore HIPAA-protected. We'll see many large free-text facts sourced into the Fact table as development continues through the Beta version, such as nursing notes, radiology and pathology reports,

and discharge summaries. Addresses are simply an early example. In the Gamma version, we'll develop tools for looking at those free-text facts in order to extract dimensional aspects of the facts, the recording and sharing of which wouldn't violate HIPAA. We'll up-dimensionalize those facts to include those identified dimensional aspects. The resulting annotations will then become available for query access even by users in deidentified access modes. The original free-text facts will remain protected, but the knowledge of the existence of these facts becomes possible through the extra dimensional annotation.

Storing addresses in the Fact table and dimensionalizing those facts to City and Postal Zone in this Geopolitics dimension matches that annotation paradigm. We'll hard-wire the annotation here in the Alpha version of the warehouse, and generalize it in an automated control later. This approach makes managing highly volatile free-text data much easier and helps assure full HIPAA compliance.

Calendar Dimension

The Calendar dimension defines the calendar time-period instances against which data are recorded. The subdimensions supported include

- Calendar Year
- Calendar Quarter
- Calendar Month
- Fiscal Calendar
- Fiscal Year
- Fiscal Quarter
- Fiscal Period
- Day

The vast majority of facts stored in the healthcare data warehouse are date-grained, so the Day subdimension is often the only subdimension actually loaded in early releases. Other subdimensions can be loaded at later, when new facts that require a different calendar grain become available. Sources that provide metrics or budget data to the warehouse are often defined at a higher-level calendar grain, including monthly, quarterly, or annual values.

There is a complexity to the Calendar dimension that sometimes goes unnoticed during early use, but that can cause problems if implemented incorrectly. The general calendar components of the dimension form a hierarchy of data against which data can be queried or aggregated easily ([Table 4.1](#)). A query looking for all facts in a calendar month will typically retrieve only the desired data.

Problems can arise when conducting similar queries against fiscal calendars. There can be many fiscal calendars stored in the Calendar dimension. To query fiscal data, it's important that the user be aware of which fiscal calendar is being included in the query. This mapping usually occurs through one or more facts that indicate which Fiscal Calendar an entry in the Organization dimension uses.

Table 4.1 Calendar Subdimension Example

Category	Year	Quarter	Month	Day
Year	2014			
Quarter	2014	Q4		
Month	2014	Q4	October	
Day	2014	Q4	October	10th

Once the correct calendar has been identified, the entries in that fiscal calendar can be queried or aggregated in the same manner as with the more general calendar entries. In your early implementation cycle, you might only load a fiscal calendar for your own organization, but resist the temptation to define it as the only possible fiscal calendar.

Clock Dimension

The Clock dimension defines the clock time-period instances against which data are recorded. The subdimensions supported include

- Meridian (2 rows: AM, PM)
- Hour (24 rows: 00–23)
- Minute (1440 rows: 00:00 through 23:59)
- Second (86,400 rows: 00:00:00 through 23:59:59)
- Time Zone

The grain you choose for your clock entries will ultimately have an impact on the results of many time-oriented queries. I typically choose to load a row for every second in the Clock dimension. I used to use Minute as the grain of my dimension, but I found that an ordering of certain events in a timeline query could combine facts that I didn't really want intermingled. For example, while we wouldn't expect a lab order result and a medication administration to match the same minute, it happens often enough that ordered queries (i.e., those with ORDER BY clauses referring to the Clock data) sometimes produce results with facts related to those two events being intermixed.

As the warehouse matures and the series of events recorded for patients gets denser, the probability of a second event occurring in the exact same time period as a first event increases. By adopting Seconds as your clock grain, you're far less likely to record two facts for a patient in the same second, so ordered queries tend to look as expected. There's always a chance of overlap though, so users have to be trained to understand what is happening when intermixed facts appear in a query result.

If clock details deeper than seconds are desired, they can be loaded into the Clock dimension as additional subdimensions; however, I have rarely seen

anyone in the biomedical sector do so. With 86,400 seconds already loaded, loading more detailed elements of time grows the dimension very quickly. Examples of deeper Clock grains can include

- Tents of second (00:00:00.0 through 23:59:59.9, 864,000 rows)
- Hundredths of second (00:00:00.00 through 23:59:59.99, 8.6 million rows)
- Thousandths of second (00:00:00.000 through 23:59:59.999, 86 million rows)
- Milliseconds (00:00:00.000000 through 23:59:59.999999, almost a billion rows)

Each of these deeper grains is a viable alternative, though I am usually reluctant to add a billion millisecond rows to the Clock dimension for performance reasons. The other alternative grains don't really scare me, and you should consider them in your implementation if you have facts that will benefit from being stored with deeper temporal specificity. Having them in the warehouse dimension will give options your source analysts might need in order to properly dimensionalize certain facts. By adding these additional rows, you allow for a richer dimension, and that should be a priority.

I've worked with organizations that didn't want to deepen the grain of the clock because they were concerned with the performance of the warehouse. While every database management system performs differently, and that performance curve is changing all the time, it isn't necessarily true that adding a couple of million rows to a dimension will hurt performance. In certain cases, it can improve performance by increasing the cardinality associated with any indexing or partitioning. A grain of 86,400 dimension rows being used by billions of facts can put quite a strain on some indexing algorithms. The same data against a million clock entries might actually perform better. Work with your DBAs before making decisions that might not actually be based on correct assumptions.

So, if performance isn't the issue, why don't I always use the deeper grained clock dimension? It's simple: I don't trust the timestamps I receive from my data sources. It's very easy for a computer system to record a timestamp to the millisecond, but that doesn't mean that I should associate that precision with what was happening in the real world when the event that was being recorded occurred. This precision is usually an illusion. Upon analysis, the values that actually arrive in most of those highly precise timestamp fields are actually rounded or truncated to the minute or second.

If an event is recorded by a clinical user as having taken place at 8:12 p.m., and I receive a data feed containing the timestamp 20:12:00.000000, I don't want to pretend that I know the precise millisecond that the event occurred. I want to store that event against the clock minute for 8:12 p.m. If I receive a value of 20:12:03.7261726, do I really want to store the event as having occurred at that millisecond? I probably do not. The source data's precision tells me something about the system process that recorded the event more than something about the real-world event itself. The source analyst who defines data extraction logic and

mappings into the warehouse should always look past the source database columns to find what really goes into those fields, and then specify extraction and transformation accordingly.

In addition to recording clock times, the Clock dimension also needs to handle any time zone requirements for the warehouse. If all of the expected data in the warehouse will be for the same time zone, then time zone can be omitted; but that precondition is becoming less true in biomedicine as our healthcare network grows, and cross-institutional interactions increase. I admit that my existing biomedical data warehouses haven't had to handle time zone processing yet. I don't have generic algorithms in place that correctly handle ordering of events from multiple time zones. I currently use a Time Zone subdimension in the Clock dimension to optionally capture a time zone for any timestamp I record, and then I deal with conversion in my queries. A word of warning, though: If you start converting timestamps based on time zones as recorded, don't forget that some of the conversions of clock time also require altering the date in the Calendar dimension. Also don't forget to allow for daylight savings time in jurisdictions that use that time-altering system.

Environment Dimension

The Environment dimension defines the various settings in which health-related situations and events arise and could have clinical meaning or significance. Examples might include

- Ecological environments in which certain parasites might be prevalent
- Whether a location would be considered urban or rural
- Observations taken in research versus clinical settings
- Accidents and other occurrences that impact health

A common source of environmental data in healthcare is the “E” code subsystem of ICD-9 which captures external causes of injury. As with the Pathology dimension, I typically load all of the ICD-9 codes to the Environment dimension, marking all but the various “E” codes as unexpected. This results in an ability to further enrich clinical facts beyond diagnosis and pathology. [Figure 4.8](#) illustrates that a patient presenting with headache and unspecified insomnia might be seen differently if it is known that she or he is in the process of getting married, moving to a new residence, or starting a new job. Environmental data such as this, when available in the data sources, can enrich the facts in the warehouse and make them more valuable to users and analysts.

Structure Dimension

The Structure dimension defines the anatomical, physiological, or chemical context of a patient or material represented in a fact. Rows in the dimension refer

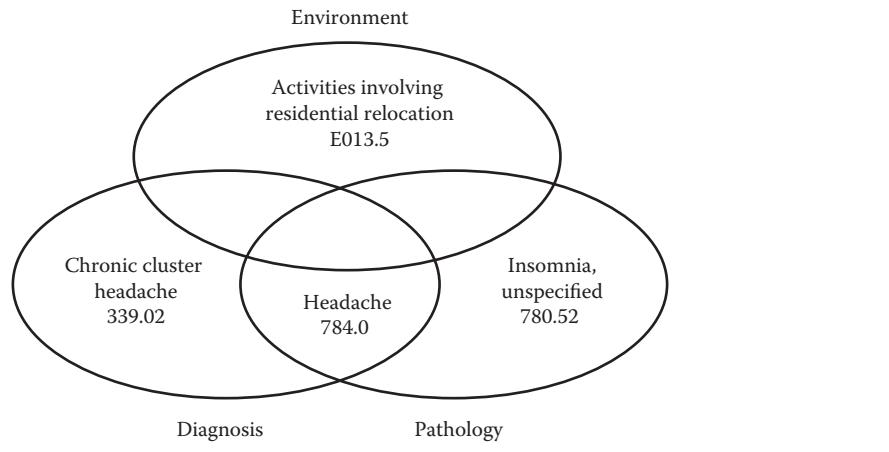


Figure 4.8 Overlap of diagnosis, pathology, and environment.

to types, not instances. For example, this dimension would define Kidney as a type of anatomical entity, but the Subject dimension would be the place to record a particular instance of a kidney, typically as a specimen associated hierarchically to an individual person. Characteristics of kidneys as a type would be recorded here, while characteristics of the instance of kidney would be defined in the Subject dimension. Subdimensions defined in this dimension typically include

- Anatomical entities
- Physiological functions
- Biological processes
- Cellular components
- Molecular functions
- Chemical elements and compounds

System Dimension

The System dimension defines the systemic context of all related facts. Examples include

- Species (e.g., human, mouse)
- Developmental stage (e.g., fetus, newborn, pediatric, adolescent, adult, geriatric)

Unit of Measure Dimension

The Unit of Measure dimension defines the syntactic and semantic elements that are used to place facts into the warehouse. This dimension is a key element of the system of internal controls for the data warehouse, helping to ensure that stored facts are of the right type and range. However, here in the Alpha version

of the warehouse, you'll limit its use to a very narrow range of options that might be needed to support Alpha version sources. There are two levels of detail in the UOM dimension:

1. *Unit*: The syntactic thing being measured by a Fact without any semantic context
2. *Measure*: A semantic meaning of a Fact, within its encompassing unit

Every fact in the warehouse is in a syntactic Unit. The use of a Measure to further define a fact using a semantic construct is optional. The actual units and measures used will largely be a function of the source analysis by the warehouse analyst. Most data sources won't specify units of measure for extracted data, so it will be up to the analyst to determine appropriate units and measures for defining new data to the warehouse. Table 4.2 includes some examples of Units that might eventually be included in the dimension, and [Table 4.3](#) includes some examples of Measures that might be included within those Units.

For this Alpha version, it is acceptable to largely ignore this dimension as initial data are extracted and loaded. However, if a source dataset happens to include an explicit unit or unit measure (e.g., lab results, medications), it can be loaded as mapped to this dimension. The UOM dimension is ultimately comprised of a natural hierarchy of up to six levels of control data, including Scale, Class, Unit, Measure, Language, and Value. A more complete implementation of those control levels will be presented in Chapter 7 during the development of your Beta version.

Table 4.2 Example Units

<i>Unit</i>
Centigrade (°C)
Millimeters of Mercury (mmHg)
Yes/No Flag
Inches
Centimeters (cm)
Count
FLACC Scale
Units/Liter (U/L)
Units/Minute (U/min)
U.S. Dollars (\$)
Euros (€)

Table 4.3 Example Unit Measures

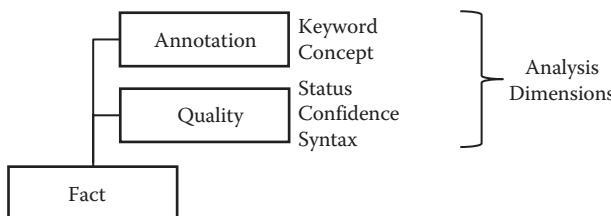
<i>Unit</i>	<i>Measure</i>
Centigrade (°C)	Oral temperature
Fahrenheit (°F)	Oral temperature
Fahrenheit (°F)	Tympanic temperature
Millimeters of Mercury (mmHg)	Systolic blood pressure
Millimeters of Mercury (mmHg)	Diastolic blood pressure
Yes/No Flag	Deceased
Units/Minute (U/min)	Breaths per minute
Units/Minute (U/min)	Beats per minute
Count	Refills
FLACC Scale	Pain level
Wong–Baker Scale	Pain level
U.S. Dollars (\$)	Revenue

Analysis Dimensions

The two analysis dimensions in the warehouse—Annotation and Quality—provide qualitative query points for the factual data (Figure 4.9). The distinction between these two dimensions is arbitrary, but generally annotations are added to data after they are loaded into the warehouse, while qualities are recorded as part of the data loading process. As a result, qualities are usually considered more stable than annotations.

Annotation Dimension

The Annotation dimension defines the *ad hoc* words or phrases that might be used to provide further traceability or access to facts. The process of annotating facts is known as *curation* and is carried out by *curators*. Curation is only rarely attempted during ETL processing because of the burden the process would place on the performance of the time-sensitive ETL jobs. Curation rules that can be

**Figure 4.9 Analysis dimensions.**

completely automated are typically built into spider routines that run against the warehouse outside of scheduled ETL processing windows. Other annotation requires manual curator intervention, and is supported through some form of curation tool—not typically built during early warehouse releases—that allows curators to write queries and apply annotation entries to the results.

The biomedical informatics community, as well as the more research-oriented bioinformatics community, has available numerous ontologies for the annotation of clinical records. The list of available ontologies grows over time, so I recommend a general ETL capability for loading new ontologies into this, and other, dimensions on a regular basis.

Curation against existing ontologies can provide a powerful means of querying data that might not otherwise be available to users who have not yet received IRB approval to look at identified patient data. Since large text blocks of narration are considered identified data (because we don't know what might occur in them), annotating these facts can provide some access to the data without having to first get IRB approval. The presence or absence of some clinical name or phrase is generally not considered identifying. This means that a user could query the warehouse for these phrases in the annotation without needing direct access to the actual HIPAA-protected facts. When the existence of an interesting cohort has been confirmed by querying the annotations, the user can go to the IRB in order to gain access to the actual facts.

Quality Dimension

The Quality dimension defines source data-based constructs that serve as qualitative data around the facts in the warehouse. Example qualities include

- Temporal state (e.g., preliminary, corrected, final)
- Abnormalcy (e.g., critical low, low, normal, high, critical high)
- Confidence (e.g., 0.00–1.00)
- Sequence (e.g., 1, 2, 3, ...)
- Flags (e.g., present on admission, reason for visit)

Control Dimensions

The four Control dimensions—Data State, Operation, Data Feed, and Metadata—of the warehouse provide for the system of controls needed to define, load, use, and manage the contents of the warehouse ([Figure 4.10](#)). While they are dimensions of the warehouse just like all the others, their content is more technical. The UOM dimension presented earlier also plays a heavy control role in the warehouse, but because it includes conventional values that are often sourced from user or standard sources, it was discussed among the Almanac dimensions earlier. Users of these data need more training than with the other

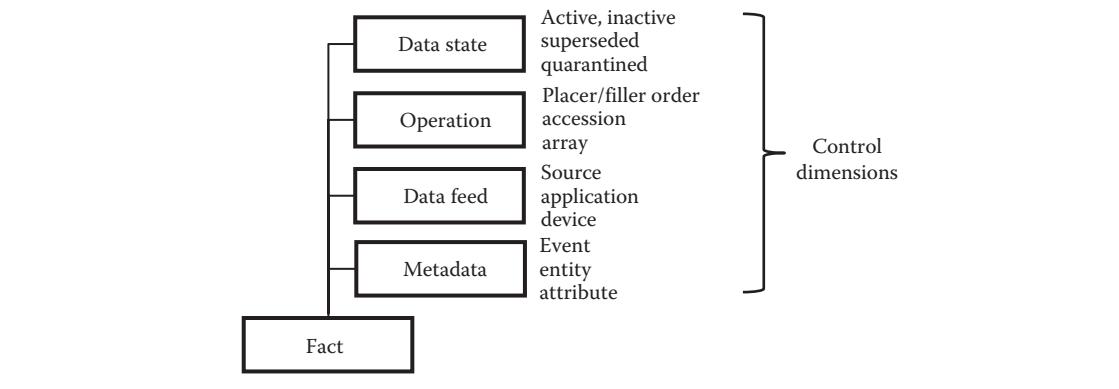


Figure 4.10 Control dimensions.

dimensions in order to completely understand the impact that the data in these dimensions has on various aspects of warehouse processing. I'm describing them at a high level here for the completeness of this chapter, but a thorough understanding of these dimensions won't occur until we've moved through a lot more of the warehouse design in upcoming chapters.

Metadata Dimension

The Metadata dimension is a cornerstone of the warehouse, providing the definition for the value that actually occurs in the value column of the Fact table. Knowing the metadata row for a fact is analogous to knowing which table and column you might query in a more traditional relational database (Figure 4.11). Although many metadata configurations might exist as a result of the analysis of an enterprise's data, common elements of those configurations would be: (a) the identification of the source system from which a fact was received (e.g., the EHR); (b) the type of event that generated the fact (e.g., a Medication Order); and (c) the correct interpretation of the fact value (e.g., Dosage).

Because this dimension actually defines the contents of the Fact table, there will rarely be a query of the warehouse that doesn't select from, or filter on, this dimension. We'll typically be interested in specific types of facts in our queries, and this dimension is how those filters of interest are implemented. The Metadata dimension also serves as the anchor point for the generic architecture of the warehouse ETL system, with each metadata row defining a fact in the Fact table along with all of the data needed to trace the fact back to its source in our biomedical systems.

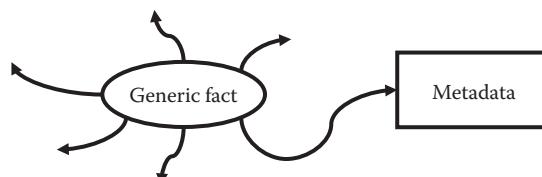


Figure 4.11 Metadata makes generic facts more concrete.

One way to visualize the role that the Metadata dimension plays in the definition of facts in the data warehouse is to imagine this warehouse design *without* the Metadata dimension. Since the metadata tells you what the value column of the one Fact table contains, not having this dimension would mean that you can't use the generic Fact value column to store all of your facts. Instead, you'd have to create a separate database column for every type of fact that you decide to store. This could possibly require you to include a separate Fact table for each type of fact you store. The number of tables you would need to create depends on how far you'd like to take this thought experiment. The Metadata dimension allows you to define separate facts for values that some people might consider a single fact. For example, if you expect to receive a systolic blood pressure from three different sources, you can use the Metadata dimension to differentiate these as separate facts, allowing you to query them by source, if desired. You might need to split these facts into three separate Fact tables to maintain that capability.

If your instinct is to want to add some form of "Source" column to the Systolic Blood Pressure fact table in order to allow those three facts to be placed into the same column of the same table without losing the ability to filter on the source systems, then you've broken this thought experiment. Your new Source column would simply be adding this Metadata back into your design without having to call it a dimension. That would be cheating. To be true to the design without using a Metadata dimension, each type of fact that would have had its own entry in the dimension has to be implemented as its own column in its own Fact table.

At this point in our thought experiment, we've removed the Metadata dimension at a cost of now having potentially hundreds, probably thousands, of different fact tables. You'll very likely want to reduce the number of fact tables by using some concept of normalization to combine many of those fact columns into fewer fact tables that are each wider than the one-column model we've been using. For example, you might try to combine all of the facts that arrive as part of a Vital Signs data feed into a wider Fact table that consolidates the 6–12 individual facts that arrive together on a regular basis. This approach breaks our thought experiment too. That cluster of facts can only be combined into a single wider Fact table if all of the dimensionality of those facts matches. Because the dimensionality of different aspects of a vital signs group differs (e.g., units of measure, body parts), they can't effectively be combined into fewer wider Fact tables without breaking the design.

The data in the Metadata dimension allow for the needed flexibility to place every sourced data element into the appropriate location in the data warehouse. Its use includes making the ETL jobs for the warehouse highly generic, as well as providing a metadata repository for users to browse in learning what the warehouse contains.

Data Feed Dimension

The Data Feed dimension keeps track of all source feeds to the warehouse in order to know exactly when each fact arrives, where it was received, and how it was transformed as it came into the warehouse. Biomedical analysts will typically use this dimension less often than the warehouse support team might, since the warehouse user community isn't typically interested in the technical processes of sourcing data; but the warehouse support person often uses this dimension in queries (e.g., "return all XYZ facts received in last 2 days from lab system ABC").

Data State Dimension

The Data State dimension allows facts to exist in different states over time. The most common state for facts is Active, and most queries of the warehouse are written to look only at Active data. There are other states that data can be assigned to as it evolves in the warehouse, and this dimension is the mechanism for those state changes. Users can include this dimension in queries if they are looking for data that might exist in another state (e.g., "return all preliminary lab results that have been superseded by different final values").

Operation Dimension

The Operation dimension allows facts to be grouped together when their other dimensionality would be insufficient for grouping them together. In the data warehouse literature, this dimension is often called the "junk" dimension. Entries include any values that the source data analyst determines are needed to properly define facts beyond their clinical or biomedical meaning. Example values include order numbers or accession numbers that carry no inherent meaning beyond being able to group facts that share the same values.

Requirements Alignment

Having described the various dimensions in our biomedical data warehouse, we can now help validate the model by checking its basic alignment with the ontologies that we reviewed in Chapter 3. The starting point for this alignment is the Basic Formal Ontology (BFO), our upper-level ontology that divides objects in the world into continuants and occurrents (left side of [Figure 4.12](#)). All of our warehouse dimensions should be traceable back to the elements of the BFO. [Figures 4.12](#) through [4.15](#) illustrate a mapping of our biomedical warehouse dimensions back to the BFO concept classifications that ground their design and use.

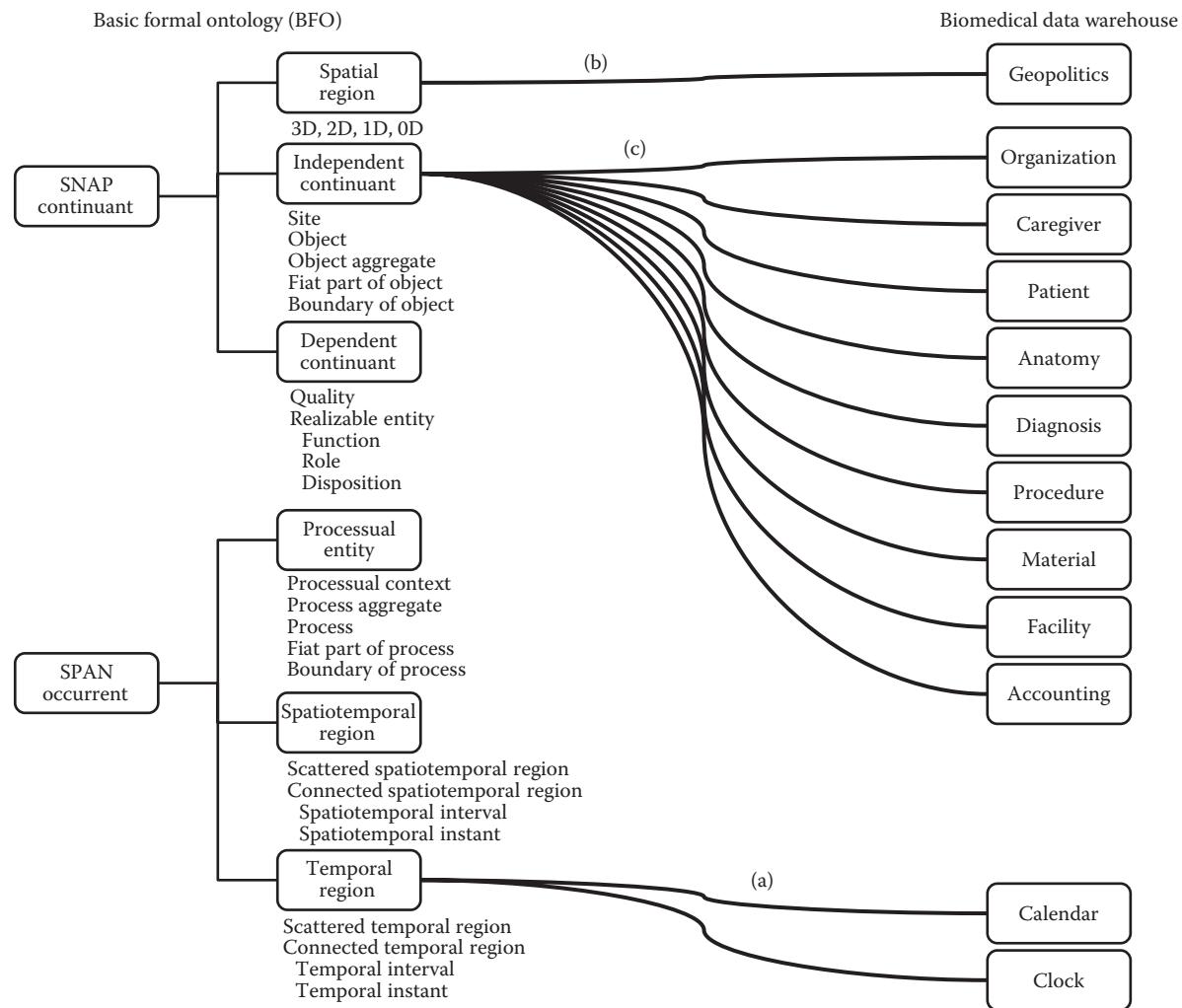


Figure 4.12 Mapping of dimensions to the concepts in the Basic Formal Ontology (BFO): (a) calendar and clock dimensions as BFO Temporal Regions, (b) geopolitics dimension as BFO Spatial Region, and (c) other biomedical dimensions as BFO Independent Continuants.

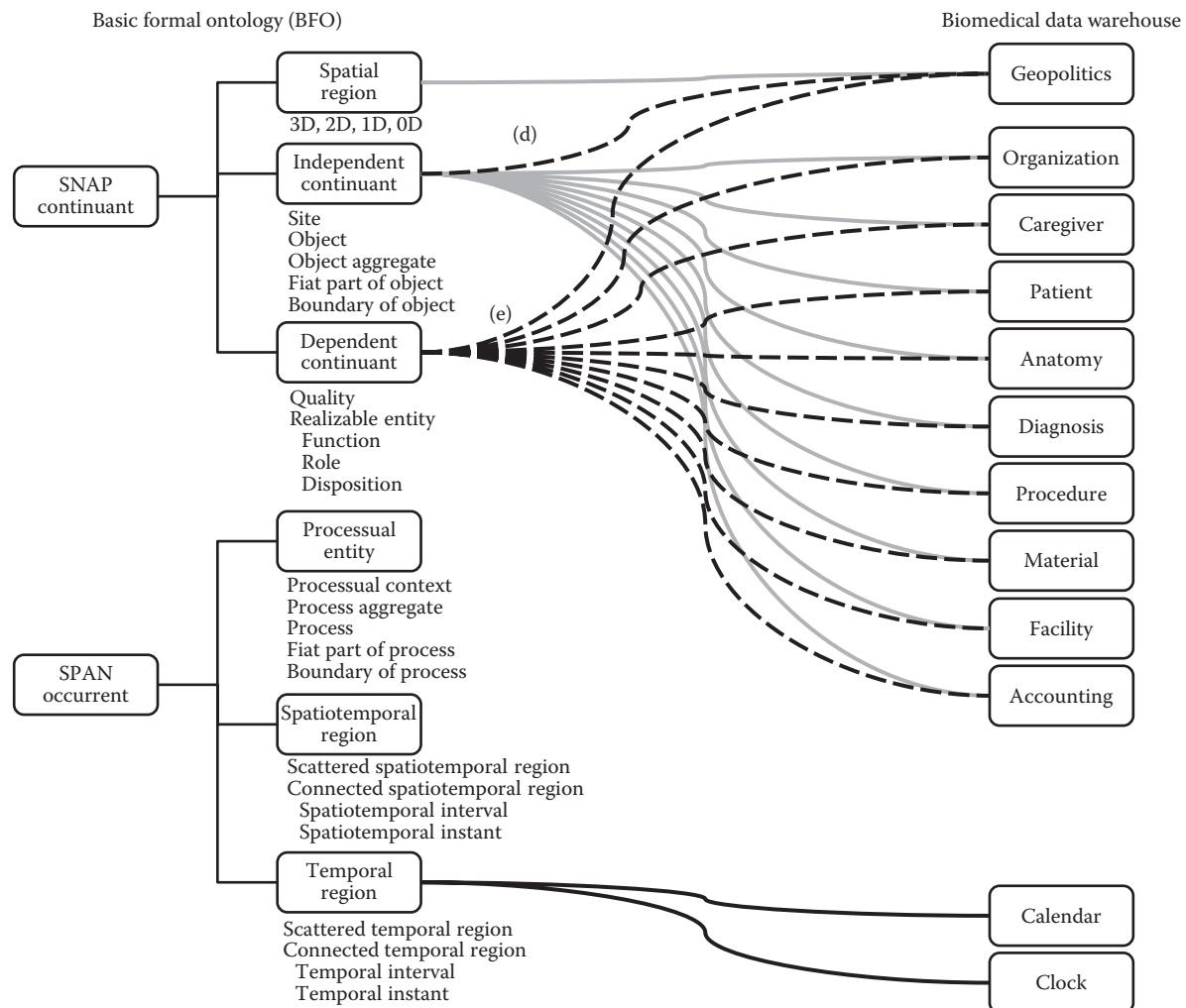


Figure 4.13 Continued mapping of dimensions to the BFO: (d) spatial regions becoming BFO sites as locations are captured and, (e) properties of dimensions defined as BFO Qualities, depending on their underlying BFO Independent Continuants.

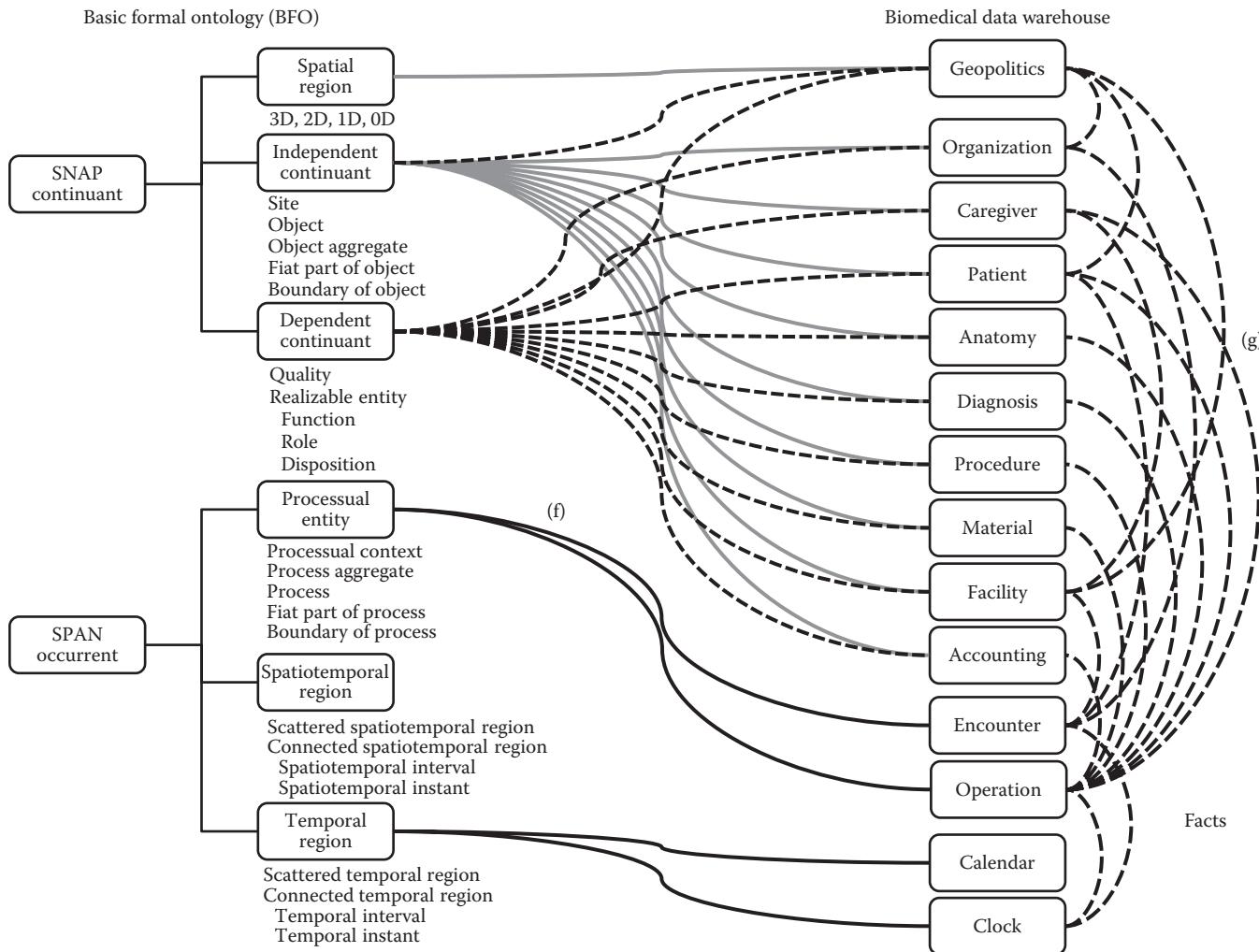


Figure 4.14 Continued mapping of dimensions to the BFO: (f) encounter and operation dimensions as BFO Processual Entities and, (g) facts realizing BFO Roles among BFO Continuants as participants and subjects.

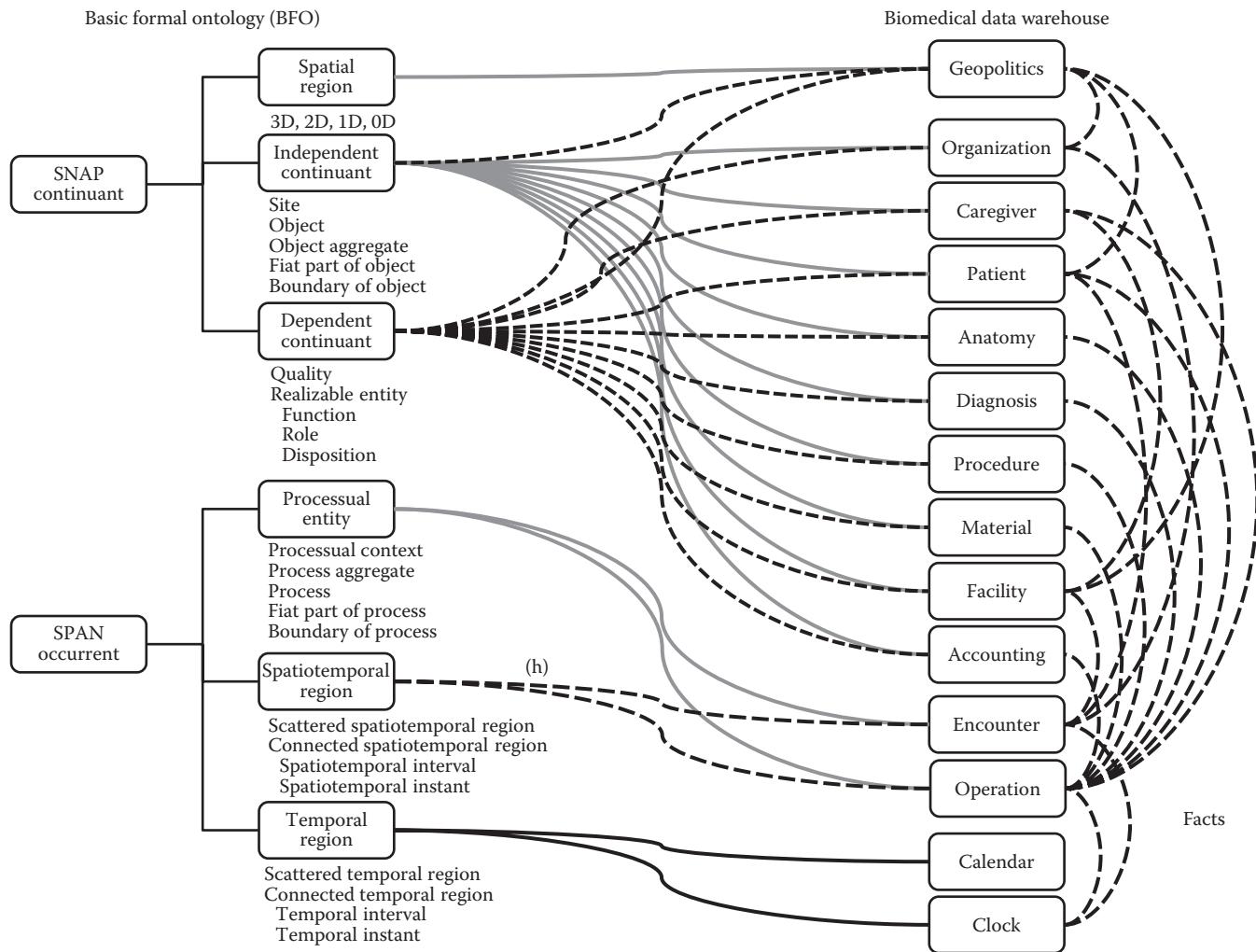


Figure 4.15 Continued mapping of dimensions to the BFO: (h) queries of the warehouse constituting BFO Spatiotemporal Regions, or “world lines” of the subjects of those queries.

Important initial dimensions in any warehouse design involve geopolitical locations and time. The Calendar and Clock dimensions represent BFO Temporal Regions (Figure 4.12a), and the Geopolitics dimension defines BFO Spatial Regions (Figure 4.12b). At this point, Geopolitics is just the surface of the Earth and not distinct addresses. Individual addresses might eventually be present in this dimension, but as BFO Sites and not simply as spatial regions.

Next in the design sequence is the bulk of the biomedical dimensions (Figure 4.12c). Note that since the definition of a warehouse dimension is really a list of entities or constructs that can later be involved with biomedical facts, we shouldn't be surprised to see that the dimensions represent BFO Independent Continuants. The dimensions exist before, and independently of, the facts that will later be stored in the Fact table of the data warehouse. They are *independent* of our use of them in the warehouse.

As data are loaded into the Geopolitics dimension, our warehouse eventually goes deeper than the BFO Spatial Regions of countries, counties, and cities to include the actual BFO Sites of interest to us (Figure 4.13d). We'll be able to tie data to specific sites that exist within those spatial regions (e.g., addresses for organizations, facilities, caregivers, or patients).

With the dimensions aligned to BFO Independent Continuants, our Master Data Management (MDM) capabilities will populate properties of everything defined in a dimension. These dimension properties serve as BFO Qualities (Figure 4.13e) in that they are *dependent* upon the existence of the construct in the dimension. Instances of the BFO Independent Continuants that don't have any known properties, known as "orphans" in the ETL, will play a key role in the loading and quality control of the data warehouse.

In anticipation of the loading of biomedical facts into the warehouse, the design needs dimensions for the Interactions and other Operations (e.g., Orders, Administrations, Procedures, Vitals, Notes) occurring within the context of the already defined dimensions. Since these occur temporally, they are BFO Processual Entities (Figure 4.14f). These dimensions serve as the containers within which all of the facts become integrated. Queries might not pay attention to them, or more correctly, queries will *cross* these containers, but all facts always exist within a processual framework.

With the process dimensions defined, the design can now store the myriad biomedical facts for which the warehouse is being defined (Figure 4.14g). These facts define the integration of all the dimensionality of the warehouse, with each dimension playing a BFO Role in the existence of the facts. These roles also continue to deepen and enrich the use of the BFO Dependent Continuant aspects of BFO Realized Entities, as each dimension serves different roles, and functions in the context of each separate fact loaded into the Fact table.

The end-result of all this warehousing activity is that the Interactions and Operations constitute collections in space and time of whatever is being analyzed in the biomedical data: BFO Spatiotemporal Regions (Figure 4.15h). This spatiotemporal aspect of the data applies to all queries of the warehouse:

Every query is a snapshot of some aspect of what took place in space and time for those aspects of the dimensions that are included in the query. A complete mapping of our dimensional model against the Basic Formal Ontology shows that our design is properly aligned to that semantic framework, and therefore our data storage and query writing will have a semantic consistency not possible without this alignment.

We'll use the expectation of this semantic consistency as a quality control tool in our design of the data warehouse as we proceed through the Beta and Gamma versions toward a production Release 1.0. Any time we map process data (i.e., data mapped to one of the BFO processual classes) into our warehouse, we'll know to look for the various participants in those processes (i.e., data mapped to one of the BFO continuant classes). For example, when we try to map data about a hospital admission event (e.g., a process), we'll know to look for the patient, provider, and nursing unit (e.g., continuants) that played a role in that event. When we want to map changing continuant data into our warehouse, we'll know to look for the process event that resulted in those changes. For instance, if we add a new patient to the warehouse, we'll look for the event that would result in adding a patient. Since there are likely to be many events that might result in adding a patient, we'll work to ensure that they are all defined in ways that result in the patient always being added in the same consistent way.

Using the BFO as a data mapping tool drives our design toward a chaining effect, with processes using and modifying continuants that are used and modified by processes that use and modify other continuants. If we see continuant data changing without a process fact, we'll know we have a gap. If we see processes being recorded without full continuant roles defined, we'll know we have a gap. Having full semantic mappings in our designs, and having confidence that the chaining effect will drive data completeness in our warehouse; we can go ahead and physically implement our warehouse database even though we haven't yet defined the detailed data that we will eventually load into our warehouse.

Chapter 5

Star Dimension Design Pattern

The generic biomedical dimensional star in the prior chapter allows us to immediately have a logical model for the data warehouse without having to conduct extensive requirements or design sessions. Our goal of developing the Alpha version of the warehouse in 3–4 weeks wouldn't be possible if you had to start from the beginning in figuring out what that logical model needed to include. Likewise, the speed with which you want to build the Alpha version also precludes conducting a lot of design activities around the database that will implement that logical model. Fortunately, a star dimension design pattern has emerged across the data warehousing community over the past two decades that allows you to implement your warehouse database almost immediately upon starting the Alpha phase project. This design pattern isn't just a shortcut for the Alpha version; it actually constitutes most of the detailed table design needed for your data warehouse over the long haul. It is the reason why dimensional data warehousing can be done so quickly and effectively: It allows your database to be generated on the first day of the project.

I mean “first day” quite literally. Before I visit a new organization to start a data warehousing project, I typically send the team the list of dimensions in Chapter 4 along with the pattern described in this chapter, so the database for the new warehouse can be generated and ready when I get there. It really can be that fast.

Structure of a Dimension

Over the past 20 years, a design pattern emerged for dimensional data warehouse development that shifted the balance in choosing data warehouse design architectures, placing the dimensional model at a significant advantage over the more traditional relational alternatives. While each architectural design might have carried roughly balanced advantages and disadvantages 15 years ago, today the benefits of the dimensional model far outweigh other traditional approaches. The main reason I can make this assertion is because of the power of the star schema design pattern, an overview of which is shown in [Figure 5.1](#).

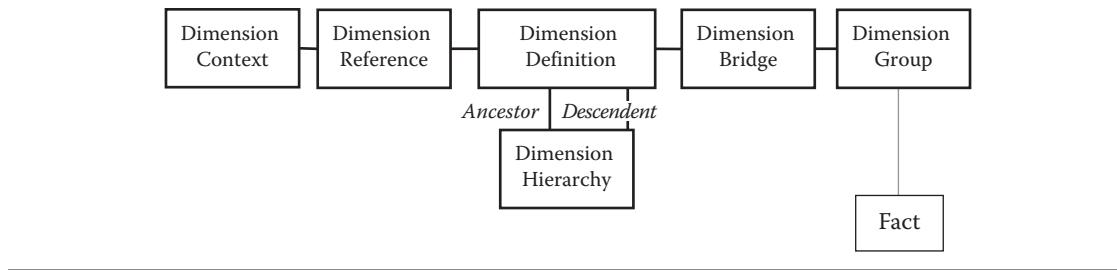


Figure 5.1 Star dimension design pattern.

Portions of the pattern have been around for decades, emerging from the work of Ralph Kimball in the 1990s. I've added other portions more recently in support of healthcare requirements, although each addition would have been very helpful in many of my pre-healthcare implementations.

The essence of the design pattern is that each dimension of the warehouse is made up of exactly the same six database tables. Each table provides a specific capability to the warehouse, and together they form the backbone of the design described in this book. In total, these tables provide for standardized ETL and query functionality to be developed using very generic and reusable logic. They are the basis for the speed with which a biomedical data warehouse can be developed and implemented.

In addition to these six tables, some of the dimensions might have one or more extra tables that store data separately for either performance or security reasons. Logically, all of these extra data would be defined in the definition table of each dimension. Moving the data into separate tables results in what are referred to as *outrigger* tables. They are completely compatible with the generic data and ETL processes described here, but their use has become less frequent as database management systems have begun to offer many other alternative means of achieving the performance and security benefits that they once supported. Since outrigger tables don't change the logical connections or content of the warehouse, I'll defer further discussion of them to Chapter 15 on data controls.

Master Data: Definition Tables

If you were to ask users to imagine what they would see in a dimension of the warehouse, they'd probably answer in terms of some master list of data. To the extent that they imagine a list of patients, a list of drugs, a list of diagnoses, or a list of chromosomes, they are thinking of what you'll load as rows in the Definition table in this design pattern. The definition table is the central anchor of the dimension design pattern (Figure 5.2), serving as your master list of constructs in the dimensions. When users think of a dimension as a form of "master file," they are thinking of the role of the definition table.

The rows of the definition table represent the list of constructs or objects defined within the dimension, making it the table that most users will access

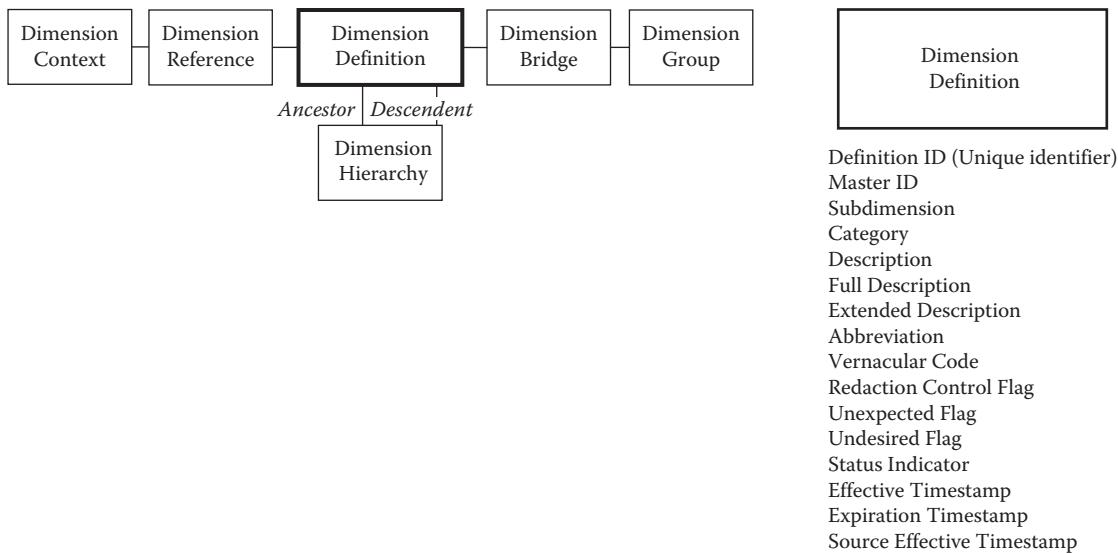


Figure 5.2 Definition table.

when they query data in the dimension. It is the only table that varies in its content across the design pattern. While the columns denoted in Figure 5.2 are common to every definition table in every dimension, the design can also include dimension-specific properties that will serve as filters and display values for users writing queries against the dimension (Figure 5.3). It is through these dimension-specific columns that different dimensions come to be described by

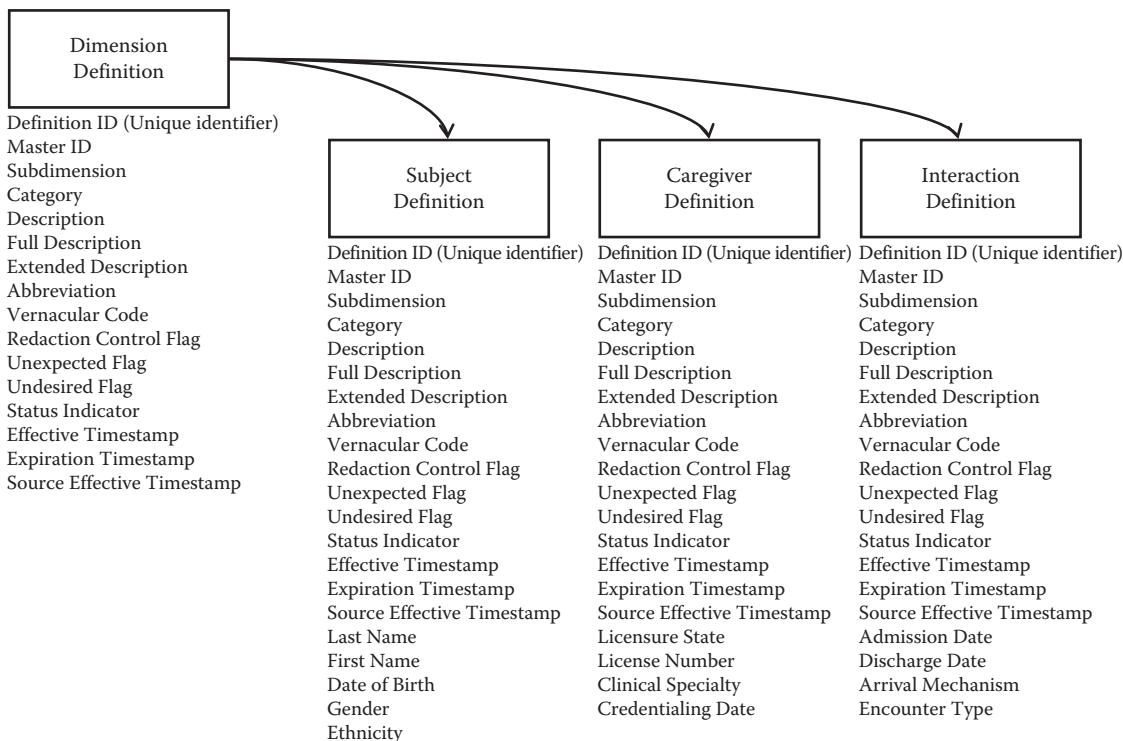


Figure 5.3 Example of dimension-specific definition columns.

ID	Subdimension	Gender	Date of Birth	Last Name	First Name	Specimen Type	Blood Type
12345	Person	Male	1955-04-03	Baker	James		O+
23456	Person	Female	2004-08-13	Smith	Jennifer		A-
34567	Specimen					Whole blood	B+
45678	Specimen					Urine	
56789	Cohort	Male					O+

Figure 5.4 Example of subject dimension definition columns by subdimension.

different properties, as the Subject dimension will have Last Name, First Name, Date of Birth, Gender, and Ethnicity columns that other dimensions are unlikely to contain. Every dimension can have an arbitrary number of columns added to the definition table to support dimension-specific requirements.

Since a dimension can contain multiple subdimensions, each definition table has a column named *Subdimension* that defines which logical subdimension a particular row belongs to. Some of the dimension-specific columns in the table will be specific to the subdimension, making them NULL in rows for other subdimensions.

Figure 5.4 illustrates a hypothetical Subject dimension definition table that defines clinical subjects at three levels of detail: cohorts, people, and specimens. Each level of detail is loaded into the dimension as a different subdimension. The rows in the person subdimension reflect the most traditional view of what most stakeholders would associate with patients. Each individual person is defined, in this example, by a Gender, Date of Birth, Last Name, First Name, and Blood Type. An individual person isn't defined by a Specimen Type, so that column is NULL in the Person subdimension rows.

Each row in the Specimen subdimension is described by a Specimen Type, and optionally by a Blood Type if appropriate for the specimen. Specimens don't have names and birthdays, so those columns are NULL in the specimen subdimension rows. We can imagine capturing gender as part of a specimen definition, but that isn't the case in this particular example, so that column is also NULL for specimen. A row representing a cohort provides data that identify a distinct group of person rows. In this example, cohort 56789 is defined as males with an O+ blood type. Because cohorts are *ad hoc*, these rows might make use of any number of defining columns in the dimension definition row.

To overcome the slight loss of standardization brought about by these subdimension-specific columns, every dimension definition table also has a set of six additional columns—known collectively as the *stub* descriptions—that are always populated with values that are appropriate to the subdimension of the row: Category, Description, Full Description, Extended Description, Abbreviation, and Vernacular Code. These columns can be queried by users, with confidence that they will always return appropriate values even when the query crosses through multiple subdimensions.

Slowly Changing Dimensions

A control feature included in this design pattern is the ability to control changes to the dimensions that occur over time so each version of the dimension definition can be captured and tracked, along with always being able to connect facts to any temporally appropriate definition entries. The changes processed by this feature are meant to be changes in value that occur slowly enough to be managed within the dimension definition itself. Faster-changing values would overwhelm the dimension, so they are loaded into the fact table as new facts. Deciding when to use this feature is an implementation choice that will vary based on the sensitivity to change of each implementation, and any performance concerns related to the extra data in the dimension that this control requires. These issues involve design concerns that don't affect how the feature works logically, so those design concerns are not addressed here.

In order to track versions of definition entries over time, each individual entry will need to be interpreted as being active during a specific time window. Three standard columns are defined in the definition table of each dimension for this purpose:

- *Status Indicator*. This indicator specifies the slowly-changing dimension (SCD) status of the particular row. The most recent row will be Active, with previous SCD versions marked as Superseded.
- *Effective Timestamp*. The earliest timestamp for which the row is considered current. When initially loaded as a newly defined construct, this timestamp is set to the earliest possible timestamp that the database manager will support, truncated to the second.
- *Expiration Timestamp*. The latest timestamp for which the row is considered current. When initially loaded as a newly defined construct, this timestamp is set to the last possible timestamp that the database manager will support, truncated to the second.

Figure 5.5 illustrates the values these three columns are assigned when a new entry is added to a dimension in the definition table.

To understand how these three columns are used to control changing data, it doesn't matter which dimension you're discussing, or what the actual row in the dimension definition describes. These control columns work exactly the same way for any entry across any dimension in the design. The row in

Status Indicator	Effective Timestamp	Expiration Timestamp
A	0001-01-01 00:00:00	9999-12-31 23:59:59

Figure 5.5 Effective-expiration period for new active definition row.

[Figure 5.5](#) describes an active entry that should be considered valid from the beginning of time until the end of time.

To denote the beginning of time, I use a value for midnight on January 1, 0001 (0001-01-01 00:00:00). For the end of time, I use 1 second before midnight on December 31, 9999 (9999-01-01 23:59:59). You might have to use different values if your database management system places constraints on what you can load into a timestamp column. If so, choose extremely early and late dates between which you expect all of your data to occur when you load your data into the warehouse. If you end up sourcing data into the warehouse with source dates earlier than your chosen date range, the generic ETL will automatically advance the source date to the earliest date you've chosen to use. Likewise, if you source data into the warehouse for dates after your chosen date range, the generic ETL will automatically reset the source date to the selected latest date. The generic ETL is designed to make those adjustments because we want to associate facts with the rows in each dimension that were active on the date of the fact. Therefore, source dates cannot occur outside of the earliest and latest timestamps selected for this feature. If your database allows it, I recommend the years 0001 and 9999, as illustrated in the figure.

If none of the columns in a definition table ends up being defined as needing to be tracked across different versions, then the values in these three columns won't change over time. The construct represented by the row in the table will always be valid from the beginning to the end of time. That doesn't mean that the row will never change, only that it will never change in a way that these control columns would be used to track different versions of the column values. If changes occur that are to be tracked through different versions, new rows will appear in the table to represent the different versions of the appropriate column values, and the control column values will change to keep track of the changing data periods. An example of the row depicted in [Figure 5.5](#) having undergone three changes of this type is illustrated in Figure 5.6.

Because the construct in the row has undergone three controlled changes, the definition table ends up having four rows, the last of which still has an Active status while the earlier three all have a status of Superseded. Collectively, the construct in the dimensions is still defined from the beginning to the end of time. The first superseded row is defined as having been active from the

Status Indicator	Effective Timestamp	Expiration Timestamp
S	0001-01-01 00:00:00	2007-05-31 23:59:59
S	2007-06-01 00:00:00	2011-03-31 23:59:59
S	2011-04-01 00:00:00	2013-12-31 23:59:59
A	2014-01-01-00:00:00	9999-12-31 23:59:59

Figure 5.6 Effective-expiration periods for active definition row after three changes.

beginning of time until 11:59:59 p.m. on May 31, 2007. The second superseded row became effective at midnight on June 1, 2007 and expired at 11:59:59 p.m. on March 31, 2011. The third superseded row became effective at midnight on April 1, 2011 and expired at 11:59:59 p.m. on December 31, 2013. The fourth, still active, row became effective at midnight on January 1, 2014 and is valid until the end of time. Collectively, these four rows provide for the definition of the construct from the beginning of time to the end of time. Each row provides for individual column values that were correct during each relevant period.

Note that by looking at the four rows in [Figure 5.6](#) there is no way to know which row was the original row in [Figure 5.5](#). Any of those four rows could have been loaded in any order by the ETL to result in this final configuration. If older data were always loaded before newer data, then we might know the first of the four rows was the original row. If the more recent data were loaded first, with the oldest data being loaded last, then the fourth row would likely be the original. There isn't any way to know which scenario has occurred by simply looking at the end result of the loads. Data can arrive in any order, and the generic ETL will always load them in the correct configuration. Since we typically think of updates arriving in the order in which they occurred, a change that occurs in a different order is referred to as a "late-arriving change." Fortunately, the generic logic that you'll build in the Beta version allows for independence in the load order.

As a result of all of this change-management control logic, if a timestamp is known for any particular fact, an appropriate row in the definition table can be associated with the fact using these three control columns. If the intent of a query is to associate a fact with the current active values in the definition, the query joins the fact to the definitions using a filter that selects only the Active row. If the intent of the query is to associate a fact with the definition values that were active at the time of the fact, the query filter looks for a definition row where the fact timestamp is between the effective and expiration timestamps, regardless of the status of the row. Joining to the currently active definition is referred to as a "Current" query, and is by far the most common way of querying the warehouse. Joining to the definition row that was active at the time of the fact is referred to as an "As Of" query because it is joining to the definition that was valid as of the date of the fact. In fact, the As Of query need not be to the date of a particular fact. Another variant of an As Of query might involve joining all returned facts to the definition rows that were active on any particular date regardless of the dates on the facts or the active periods of the definitions. A query that uses definition entries as of the end of a financial reporting period is an example of this kind of an As Of query.

To implement these SCD controls, you'll need to decide what temporal grain you want to use in your effective and expiration timestamps. In my examples, I've chosen timestamps that provide detail down to the second. The grain you choose to use might eventually impact the number of rows that are loaded into your dimensions. By choosing a grain at the second level, if the data

change almost continuously, you could end up loading a lot of new rows into a dimension. In theory, you could end up with a new row every second. In practice, you probably know your dimension data doesn't change that often, so you're not really concerned about that happening, but as a data analyst you need to be aware of the possibility. This control feature is called SCD because it is intended to support changes to dimension data that happen slowly, and slowly is a relative term.

If data value changes that need this kind of version control occur frequently enough to cause problems in the dimension, I recommend not using this control to manage the versions. Instead, define the columns that are driving the higher volatility as *not* using version controls, and add new facts to the fact table to capture the value of those columns each time they arrive. This will result in the most recent version of these columns being available in the definition table of the dimension, and all previous values of the columns being available in the fact table when needed.

Another approach to handling high volatility in definition data would be to broaden the grain of the effective and expiration timestamps. If the grain is broader, changes that take place at a narrower grain won't get recorded as new entries in the definition table. One common approach I've seen is to restrict the effective and expiration periods to dates alone, rather than timestamps. Changes that occur on different days end up being represented in different rows in the dimension. Changes that occur on the same date are discarded, with only the last received value being included in the definitions. I don't recommend this approach because it's too arbitrary: Data that change just before and after midnight get stored, while changes just before and after noon get discarded.

Keeping the grain of the SCD control columns at the second level generally prevents data that change data from being discarded, but it doesn't eliminate the possibility entirely. Even at a 1-second grain, multiple changes to a data column could theoretically arrive for the same second. All but the last one received will be discarded by the generic ETL in this design. If the data analyst defining requirements for this feature sees these transactional collisions as important, consider defining facts to capture all of the changes to these data columns regardless of temporal grain. Generally, I find that using the 1-second SCD grain solves the problem enough that I'm rarely concerned with the issue as I source data.

To implement the controls for SCD tracking, five columns are required in the definition table of each dimension. Here are the two new columns, and an updated definition for the three we've been discussing:

- ***Master ID.*** An integer identifier is used to group all of the definition rows for the same construct regardless of how many SCD changes the defined construct has gone through. Its value is set to the surrogate key value generated for the first instance of the construct loaded into the dimension in Definition ID. Subsequent SCD versions of the definition will have new and unique Definition ID values, but the Master ID never changes.

- *Source Timestamp*. A timestamp for which the row was considered active at the source. This value allows future SCD transactions to be properly sequenced in the SCD chain for the construct.
- *Status Indicator*. An indicator specifies the SCD status of the particular row. The most recent row will either be Active or Deleted (based on the most recent transaction type), with previous SCD versions marked as Superseded.
- *Effective Timestamp*. The earliest timestamp for which the row is considered current. When initially loaded as a newly defined construct, this timestamp is set to the earliest possible timestamp that the database manager will support, truncated to the second. On subsequent SCD transactions that insert versions earlier than this row, this timestamp is set to the source timestamp of the current row, truncated to the second.
- *Expiration Timestamp*. The latest timestamp for which the row is considered current. When initially loaded as a newly defined construct, this timestamp is set to the last possible timestamp that the database manager will support, truncated to the second. On subsequent SCD transactions that supersede this row, this timestamp is set to 1 second before the source timestamp for the superseding data transaction.

The intent of SCD processing is to allow there to be one—and only one—valid row for every dimensional construct at any one time, and that there be an existing valid row at any point in time, whether past or future. The warehouse ETL always looks up dimension entries through a join to the Master ID (which returns the *set* of definition rows for the defined construct) and a query constraint that the transaction timestamp in the source transaction be between the effective and expiration timestamps (which limits the set to the single row for the construct active at that timestamp). Note that for this to work, the transaction timestamps from the source data are always truncated to the grain of the SCD control (i.e., to the second if you are following my recommendation).

If the defined construct has never gone through an SCD change, the Master ID returns only one row, and the dates in that row are the earliest and latest possible dates. No extra logic is required regardless of how many SCD changes have occurred, resulting in an arbitrary number of definition table rows for each defined construct. Keep in mind: If you want the overall construct, look at the Master ID; if you want a time-sensitive version of the construct, look at the Definition ID.

SCD Example

Let's imagine that we want to keep track of hospitals in our warehouse as entries in the dimension that keeps track of organizations. One hospital in our initial load is City Hospital, and we've got data involving City Hospital dating back to the beginning of June in 1998. On New Year's Day in 2005, City Hospital was renamed City Medical Center. We will use the SCD control feature to ensure we

Before...

Definition ID	Master ID	Status Indicator	Effective Timestamp	Expiration Timestamp	Source Timestamp	Description
123	123	A	0001-01-01 00:00:00	9999-12-31 23:59:59	1998-06-01 00:00:00	City Hospital

After...

Definition ID	Master ID	Status Indicator	Effective Timestamp	Expiration Timestamp	Source Timestamp	Description
123	123	S	0001-01-01 00:00:00	2004-12-31 23:59:59	1998-06-01 00:00:00	City Hospital
345	123	A	2005-01-01 00:00:00	9999-12-31 23:59:59	2005-01-01 00:00:00	City Medical Center

Figure 5.7 Dimension SCD control example.

can control which version of the description for our hospital is made available to queries, some of which always need to retrieve the currently active value, while others want to retrieve the value that was in effect at the time of the fact, and still others want to retrieve a version that was active on any other arbitrary date.

Figure 5.7 illustrates the use of the data control for the City Hospital example. We immediately recognize that the “Before” version has not gone through any SCD changes because the Master ID is equal to the Definition ID, the row is active, and it covers the beginning of time to the end of time (at least years 0001 through 9999 in this example). After the name change took place in 2005, the construct had two rows in the definition table, the first of which was *superseded* by the second. The original row is now recorded as expired 1 second before the new row becomes effective. It’s the new row that now runs into the distant future. If a query accesses this definition for any data in the past or future, the query will return the correct hospital name.

This example illustrates a change taking place over time, where the changed data arrived after the original data. New rows are added to the chain as new changes arrive over time. What about changes that arrive late and out of sequence? Suppose that after all of these data were processed and loaded, someone learned that, as recently as July 1990, City Medical Center had been known as City Sanatorium. Figure 5.8 illustrates this late-arriving change. The new definition row created by the transaction will be considered superseded because it is older data than the more recent versions already in the table. It will have a higher Definition ID because surrogate keys are generated as ascending sequence numbers in the database, so a newer row will have a higher identifier even though it represents older data. This new row will be marked to expire 1 second before the original source transaction date. There might have been additional changes that occurred between the known dates, but they are unknown at this time. If they arrive later, they’ll be inserted into the chain at just the right point.

This example allows for some general observations about definition data. First, having the Master ID equal to the Definition ID shouldn’t be mistaken for the earliest row being viewed. It does indicate that the first definition inserted is being viewed but, because of late-arriving changes, the first

Before...

Definition ID	Master ID	Status Indicator	Effective Timestamp	Expiration Timestamp	Source Timestamp	Description
123	123	S	0001-01-01 00:00:00	2004-12-31 23:59:59	1998-06-01 00:00:00	City Hospital
345	123	A	2005-01-01 00:00:00	9999-12-31 23:59:59	2005-01-01 00:00:00	City Medical Center

After...

Definition ID	Master ID	Status Indicator	Effective Timestamp	Expiration Timestamp	Source Timestamp	Description
123	123	S	1998-06-01 00:00:00	2004-12-31 23:59:59	1998-06-01 00:00:00	City Hospital
345	123	A	2005-01-01 00:00:00	9999-12-31 23:59:59	2005-01-01 00:00:00	City Medical Center
567	123	S	0001-01-01 00:00:00	1998-05-31 23:59:59	1990-07-01 00:00:00	City Sanatorium

Figure 5.8 Dimension SCD control example, late-arriving.

row is not necessarily the earliest row. Second, the effective timestamp is eventually equal to the source timestamp. The only row where this is not true is the earliest row. Don't be fooled into thinking that both columns aren't necessary. Without these columns being separate, there would never be a way to insert a late-arriving change earlier than the first row inserted because no transaction date would ever precede the 0001-01-01 earliest effective date used in this design pattern.

Let's try one last variation in the example. Suppose that we subsequently receive a transaction that tells us to delete the City Medical Center entry from the dimension starting at the beginning of 2012. Figure 5.9 illustrates this deletion. Note that the new row (789) has a status of deleted and should be considered effective on January 1, 2012 through the end of time. It isn't necessarily intuitive that the current definition of a construct in the dimension should be considered

Before...

Definition ID	Master ID	Status Indicator	Effective Timestamp	Expiration Timestamp	Source Timestamp	Description
123	123	S	1998-06-01 00:00:00	2004-12-31 23:59:59	1998-06-01 00:00:00	City Hospital
345	123	A	2005-01-01 00:00:00	9999-12-31 23:59:59	2005-01-01 00:00:00	City Medical Center
567	123	S	0001-01-01 00:00:00	1998-05-31 23:59:59	1990-07-01 00:00:00	City Sanatorium

After...

Definition ID	Master ID	Status Indicator	Effective Timestamp	Expiration Timestamp	Source Timestamp	Description
123	123	S	1998-06-01 00:00:00	2004-12-31 23:59:59	1998-06-01 00:00:00	City Hospital
345	123	S	2005-01-01 00:00:00	2011-12-31 23:59:59	2005-01-01 00:00:00	City Medical Center
567	123	S	0001-01-01 00:00:00	1998-05-31 23:59:59	1990-07-01 00:00:00	City Sanatorium
789	123	D	2012-01-01 00:00:00	9999-12-31 23:59:59	2012-01-01 00:00:00	City Medical Center

Figure 5.9 Dimension SCD control example, deletion.

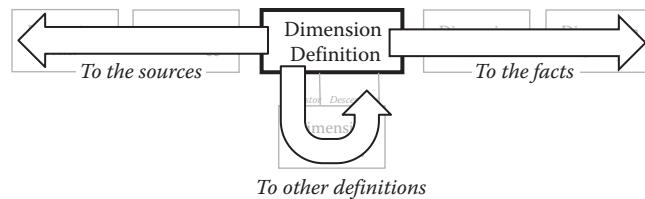


Figure 5.10 Connecting definitions through the design pattern tables.

deleted, but it works. If newly sourced facts arrive for loading, the generic ETL will be able to warn us that new data are still being recorded against entries in the dimensions that had previously been indicated as being deleted.

The rest of the star dimension design pattern includes ways in which the five remaining tables in the pattern allow the data in the definition table to be connected to other data (Figure 5.10).

Each dimension will have paired context and reference tables to support tracing data back to their sources (left of Figure 5.10), and allow data from different source applications to be merged seamlessly into a single warehouse. A hierarchy table will support connections among different definition entries within the dimension (bottom of Figure 5.10), and paired group and bridge table (right of Figure 5.10) will support pathways between the definitions and the facts that use them. The SQL CREATE statement will vary for each dimension as dimension-specific attributes are added to the dimensional design. The basic CREATE statement will include

```
CREATE TABLE PROCEDURE_D
(
    DEFINITION_ID          INT           NOT NULL,
    MASTER_ID               INT           NOT NULL,
    SUBDIMENSION            VARCHAR(30)   NULL,
    CATEGORY                VARCHAR(20)   NULL,
    DESCRIPTION              VARCHAR(30)   NULL,
    FULL_DESCRIPTION        VARCHAR(100)  NULL,
    EXTENDED_DESCRIPTION    VARCHAR(250)  NULL,
    ABBREVIATION             VARCHAR(15)   NULL,
    EFFECTIVE_TIMESTAMP     DATETIME     NULL,
    EXPIRATION_TIMESTAMP    DATETIME     NULL,
    STATUS_INDICATOR        VARCHAR(1)    NULL,
    SOURCE_TIMESTAMP         DATETIME     NULL,
    ORPHAN_FLAG              VARCHAR(1)    NULL,
    REDACTION_CONTROL_FLAG  VARCHAR(1)    NULL,
    UNEXPECTED_FLAG          VARCHAR(1)    NULL,
    UNDESIRED_FLAG           VARCHAR(1)    NULL,
    CREATE_TIMESTAMP          DATETIME     NULL,
    CREATE_DATAFEED_ID       INT          NULL,
    MODIFY_TIMESTAMP          DATETIME     NULL,
    MODIFY_DATAFEED_ID       INT          NULL
);
```

By varying the name of the table in this statement, all of the definition tables in your warehouse can be created very quickly.

Source Keys: Context and Reference Tables

The Context table defines the various source system keys and identifiers that define data within the context of those source systems. Every application system from which the warehouse will source data will have its own ways of identifying data within its own databases and files. Some of those source keys will occur naturally in the data, while others will be defined exclusively within the scope of one or more application systems. Often, the same real-world objects will have different keys defined in different systems, and cross-mappings among those redundant keys will not always be defined or available. Each of these types of keys, whether unique to one application or shared across multiple systems, represents an identifying context that must be recognized and processed by the data warehouse ETL functionality. Examples of these contexts include

- A 9-digit medical record number (MRN) used to identify patients in the Electronic Health Record (EHR), that is also shared with almost every other application system
- A 10-digit account number created by the EHR for each encounter, that is shared across most systems
- A 5-digit drug identifier created in the pharmacy system that is also used by the computerized provider order entry applications
- A physician's 10-character National Provider Identification identifier
- An 8-character network user identification that allows users access to each application on the network and is usually used by those applications to identify users in transactions
- A Current Procedural Terminology, Fourth Revision (CPT-4) procedure or treatment code
- An International Classification of Diseases, Ninth Revision (ICD-9) diagnosis or procedure code

Although the warehouse never uses these natural keys or system identifiers as the identifiers within the warehouse, they must be stored and cross-referenced to the appropriate dimension definition rows in order to allow for ETL mapping to take place. The data will be stored using a series of surrogate keys that are only meaningful within the data warehouse, but those keys need to be mapped back to the original source keys that were used in the contexts of those source systems. The context table is a starting point for access to the dimension. It defines the set of source system identifiers that map to dimensional entries (e.g., “MRN” might be a context that is used to map the instances of MRN to each patient’s row in the definition table of the Subject dimension). The reference table serves

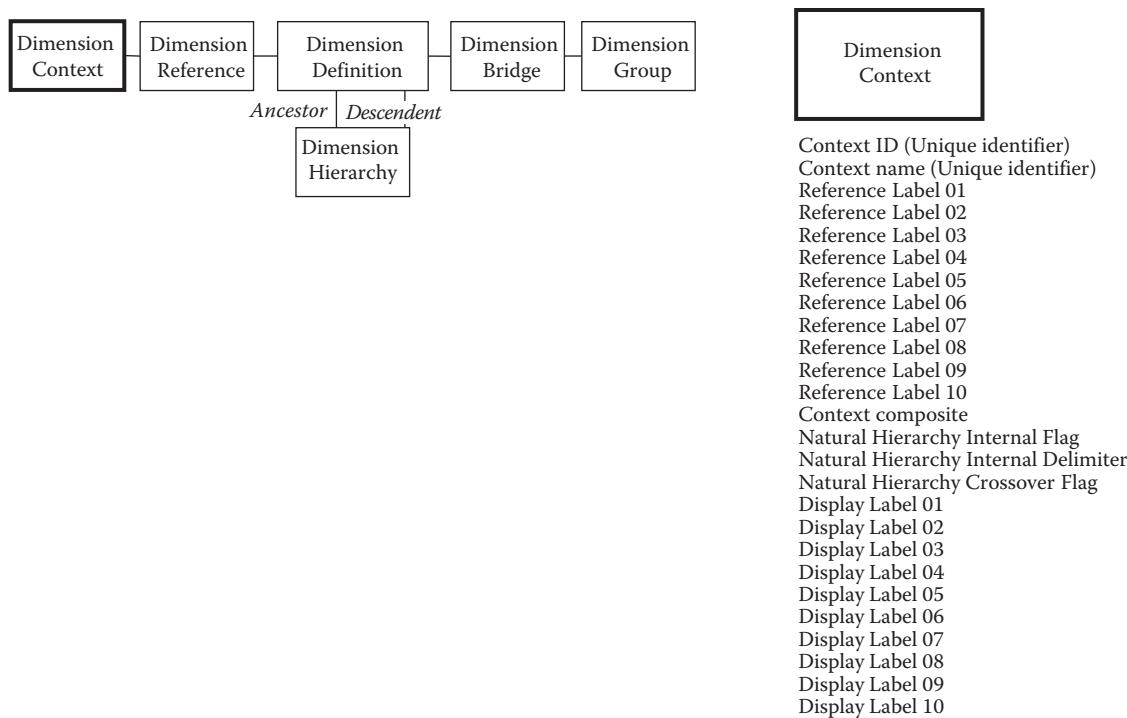


Figure 5.11 Context table.

to capture those instances, resolving the many-to-many relationship between the context and definition tables. Each reference table row maps a definition table entry to the required context, thus integrating the source natural or systemic keys into the data warehouse.

Each individual context that identifies data in the dimension needs to be present in the dimension's context table (Figure 5.11). Contexts are pre-built by the warehouse team during the implementation of the warehouse, with new contexts being added over time as new data sources that contain new application source keys are added to the warehouse. Compared to most other tables in the data warehouse, the context tables are extremely stable. Each context is identified by a surrogate identifier that makes it unique across the warehouse system, and a context name that makes it unique within the dimension.

Beyond its identifying keys, each context also provides information about what the instances of application or source keys will look like to the ETL system. This generic design allows source keys to be made up of up to 10 component values, so the context table identifies how many components will be used, and what the components are. The column list in Figure 5.11 includes two sets of 10 labels, the “Reference” set being used as the internal control values for warehouse processing, and the “Display” set being available for users to include in their queries as column headings or search filters. Once the contexts are established in the context table, the actual mappings of definitions to contexts take place in the reference table (Figure 5.12). Reference entries are not pre-loaded, they only occur as new data are added to the warehouse by the ETL subsystem.

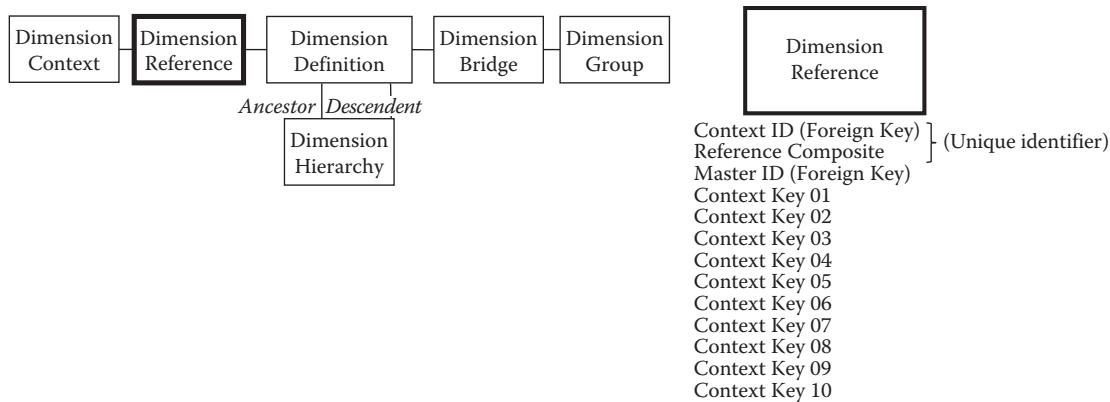


Figure 5.12 Reference table.

The Reference table contains the actual application source key component values that define data in the dimension in up to 10 columns known as Context Keys. The number of context keys populated must match the number of Reference Labels that were populated in the associated context. The context key values are also pipe-delimited (i.e., separated by the “|” character known as a “pipe”) as a single string in the Reference Composite column. It is the Reference Composite value, along with the associated Context ID, that makes a reference entry unique.

These Context and Reference tables (Figure 5.13) enable the level of generic processing made possible by this design pattern because they allow the ETL to perform without any embedded knowledge of the natural or system key structures of the data being processed. The number of keys supported in the context–reference interaction is an implementation choice within the pattern, and in recent years I’ve settled on 10. I haven’t needed 10 yet, but I regularly see a few keys needing 7 or 8, so I suggest being prepared. Additional keys can always be added later, but it requires very extensive coding and regression testing changes to be done. For now, 10 is a safe design parameter.

Figure 5.14 depicts a multipart key context for identifying laboratory tests, where the application key requires the use of three different data columns from the source application. The Context ID is 36, and the Context Name is “Lab Test.” In the context table, the first three Reference Label columns have been used to

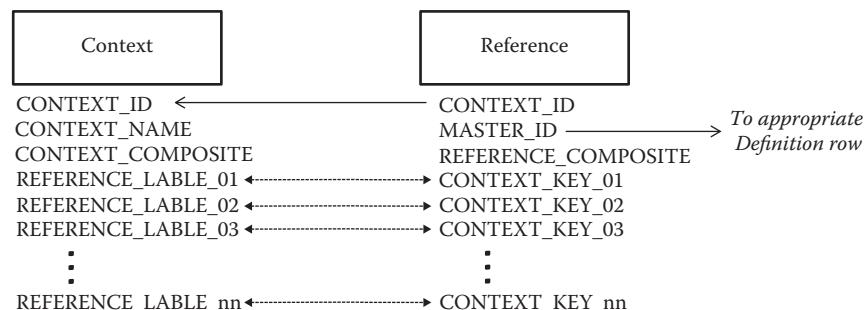


Figure 5.13 Context and Reference table interaction.

Context	Reference
CONTEXT_ID {36}	CONTEXT_ID {36}
CONTEXT_NAME {Lab test}	MASTER_ID {123456}
CONTEXT_COMPOSITE {Department Test Number Specimen Type}	REFERENCE_COMPOSITE {LB 05367 BL}
REFERENCE_LABEL_01 {Department}	CONTEXT_KEY_01 {LB}
REFERENCE_LABEL_02 {Test Number}	CONTEXT_KEY_02 {05367}
REFERENCE_LABEL_03 {Specimen Type}	CONTEXT_KEY_03 {BL}

To appropriate Definition row

Figure 5.14 Context and reference “Lab Test” example.

identify the Department, Test Number, and Specimen Type columns that are needed to form the multipart application key. The Context Composite value is also set to the pipe-delimited combination of these three labels.

The right side of the figure depicts reference table values, where the three context keys contain the appropriate values that allow the Master ID to point to the correct entry in the definition table (Master ID = 123456). The Reference Composite column is also set to the pipe-delimited concatenated values from the three context keys. This composite allows queries to access the reference entries without needing to know exactly how many reference keys were defined in the associated context. Although dimension references represent external data that are mapped into the warehouse primarily in support of the ETL process, it is also common for users to include certain popular or familiar reference codes (e.g., ICD-9 codes, Unit Number, MRN) in their queries, so the values in these two tables are also typically available to users in the query environment for the warehouse. Users are typically taught that if they are seeking data that would be a whole or partial key in the source applications, they should expect to find them in the reference tables.

User training needs to ensure that users understand the potential Cartesian implications of querying a join between the reference table and the definition table in a dimension. Many definitions in the dimensions will have keys in multiple contexts, and it is customary to filter joins to the reference table on the intended Context ID. Even so, a Cartesian product is still possible if a row in the dimension is identified by multiple keys in the same context. An example of this situation is where a patient is known by two different MRNs after a patient merge transaction has been processed. The power of this capability to have any number of different keys associated with the same master data increases the need for good user training and awareness.

If a key structure is in common enough usage to have entered the vernacular language of the user community, it can also be placed in the Vernacular Code column of the definition table to ease user access; however, note that the definition column only allows one vernacular version of the code structure while the context and reference constructs allow multiple

natural and system keys to be defined for each defined construct. Vernacular codes placed into the associated definition tables typically include the patient's MRN, the encounter's account number, and ICD-9 and CPT-4 codes. Designating a vernacular code for a definition entry can help avoid the Cartesian join problem when it occurs. For example, a patient might be known by multiple MRNs because of previous merge transactions that corrected for duplicate entries. If so, only one of those MRNs is an active identifier in the source systems. Placing that MRN in the Vernacular Code in the definition table helps resolve the duplication without users having to understand the complexity of the original source data. The active MRN is available quickly in the definition table, while the entire list of MRNs is available in the reference table, if needed.

The context table for each dimension is created using an SQL CREATE statement that varies only by the name of the table being made specific to each dimension:

```
CREATE TABLE PROCEDURE_C (
    CONTEXT_ID INT NOT NULL,
    CONTEXT_SOURCE VARCHAR(64) NOT NULL,
    CONTEXT_NAME VARCHAR(64) NOT NULL,
    CONTEXT_COMPOSITE VARCHAR(800) NOT NULL,
    REFERENCE_LABEL_01 VARCHAR(64) NOT NULL,
    REFERENCE_LABEL_02 VARCHAR(64) NULL,
    REFERENCE_LABEL_03 VARCHAR(64) NULL,
    REFERENCE_LABEL_04 VARCHAR(64) NULL,
    REFERENCE_LABEL_05 VARCHAR(64) NULL,
    REFERENCE_LABEL_06 VARCHAR(64) NULL,
    REFERENCE_LABEL_07 VARCHAR(64) NULL,
    REFERENCE_LABEL_08 VARCHAR(64) NULL,
    REFERENCE_LABEL_09 VARCHAR(64) NULL,
    REFERENCE_LABEL_10 VARCHAR(64) NULL,
    NATURAL_HIERARCHY_INTERNAL_FLAG CHAR(1) NULL,
    NATURAL_HIERARCHY_INTERNAL_DELIMITER CHAR(1) NULL,
    NATURAL_HIERARCHY_CROSSOVER_FLAG CHAR(1) NULL,
    DISPLAY_LABEL_01 VARCHAR(64) NULL,
    DISPLAY_LABEL_02 VARCHAR(64) NULL,
    DISPLAY_LABEL_03 VARCHAR(64) NULL,
    DISPLAY_LABEL_04 VARCHAR(64) NULL,
    DISPLAY_LABEL_05 VARCHAR(64) NULL,
    DISPLAY_LABEL_06 VARCHAR(64) NULL,
    DISPLAY_LABEL_07 VARCHAR(64) NULL,
    DISPLAY_LABEL_08 VARCHAR(64) NULL,
    DISPLAY_LABEL_09 VARCHAR(64) NULL,
    DISPLAY_LABEL_10 VARCHAR(64) NULL,
    ENTERPRISE_SPECIFIC_FLAG CHAR(1) NULL
);
```

After creating the context table, the reference table CREATE typically follows:

```
CREATE TABLE PROCEDURE_R
(
    CONTEXT_ID          INTEGER      NOT NULL,
    MASTER_ID           INTEGER      NOT NULL,
    REFERENCE_COMPOSITE VARCHAR(640) NOT NULL,
    CONTEXT_KEY_01       VARCHAR(64)  NOT NULL,
    CONTEXT_KEY_02       VARCHAR(64)  NULL,
    CONTEXT_KEY_03       VARCHAR(64)  NULL,
    CONTEXT_KEY_04       VARCHAR(64)  NULL,
    CONTEXT_KEY_05       VARCHAR(64)  NULL,
    CONTEXT_KEY_06       VARCHAR(64)  NULL,
    CONTEXT_KEY_07       VARCHAR(64)  NULL,
    CONTEXT_KEY_08       VARCHAR(64)  NULL,
    CONTEXT_KEY_09       VARCHAR(64)  NULL,
    CONTEXT_KEY_10       VARCHAR(64)  NULL
);
```

By varying the names of the table in these statements, all of the Context and Reference tables in your warehouse can be created very quickly.

Fact Participation: Group and Bridge Tables

While the Context and Reference tables provide access to the Definition table through the natural and system keys in the data source applications, the Group and Bridge tables provide access to the Definition table from the point of view of the facts. The essence of these constructs is that a single fact can be connected to more than one entry in a single dimension. Since there is only one foreign key in the fact table for each dimension, the key of the definition table won't support that requirement. Instead, each fact points to a group of entries in the Definition table through the Group and Bridge tables.

A group is an arbitrary construct created to embody some combination of connections between a fact and the entries in a dimension (Figure 5.15). It is simply defined as a surrogate identifier known as a Group ID. The only column other than Group ID in the group table is the Group Composite, which is a delimited aggregation of the information for the group that is being defined in the bridge table depicted in Figure 5.16.

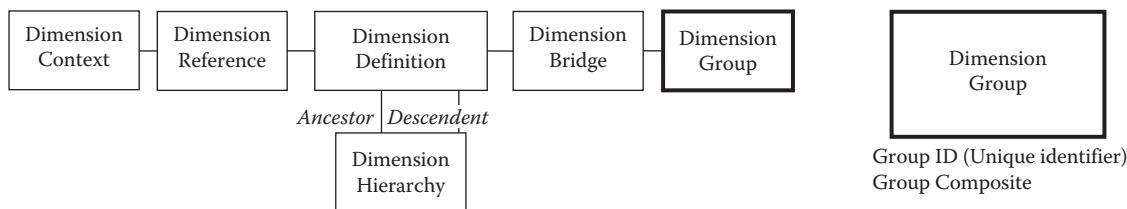


Figure 5.15 Group table.

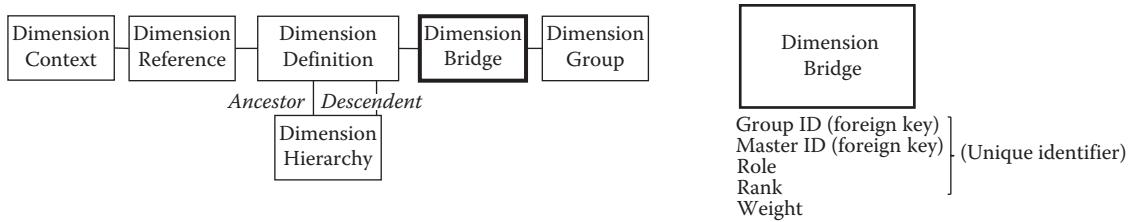


Figure 5.16 Bridge table.

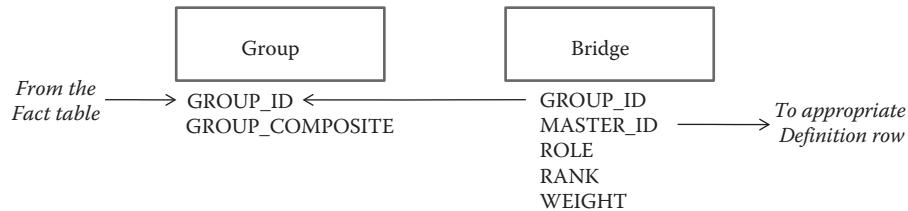


Figure 5.17 Group and bridge table interaction.

A bridge is an entry in the Bridge table that connects one Definition table entry to one Group table entry. The set of bridges that point to the same Group ID constitutes the group. A row in the Group table is a wide version of the information in the deep rows of the Bridge table. The interaction of the Group and Bridge tables is depicted in Figure 5.17.

Through the group, a fact can now be connected to any arbitrary number of definition entries. A group with only one bridge entry—a very common occurrence—is a special case of a group. The bridge table contains three columns that allow the various connections between the Fact and Definition tables to be differentiated. The Role provides a textual description of the definition's involvement in the fact. The Rank provides uniqueness to the bridge entry, for cases where multiple definitions serve the same Role. The rank is typically set to one since most roles are distinctive within a group. The Weight is a real number between 0 and 1 that signifies the proportion of any aggregated data across the bridge that should be allocated to the particular role and rank. The sum of the weights in a group must equal one. This allows data in the fact table to be prorated or allocated during aggregations, although this feature is not used at all for nonquantitative data. For syntactic correctness, the weight is always assigned, even for data for which algebraic aggregation is not anticipated.

The Group Composite column in the Group table is used for a pipe-delimited concatenation of bridge composite values, which are themselves each a plus-delimited concatenation of the data in the bridge entry. This composite allows the ETL to access the appropriate group through a single well-indexed query to the group table rather than conducting a cursor-based search of the bridge table to see if any particular group exists in these tables. No two groups will contain the same value for the Group Composite, allowing groups to be safely reused across many facts.

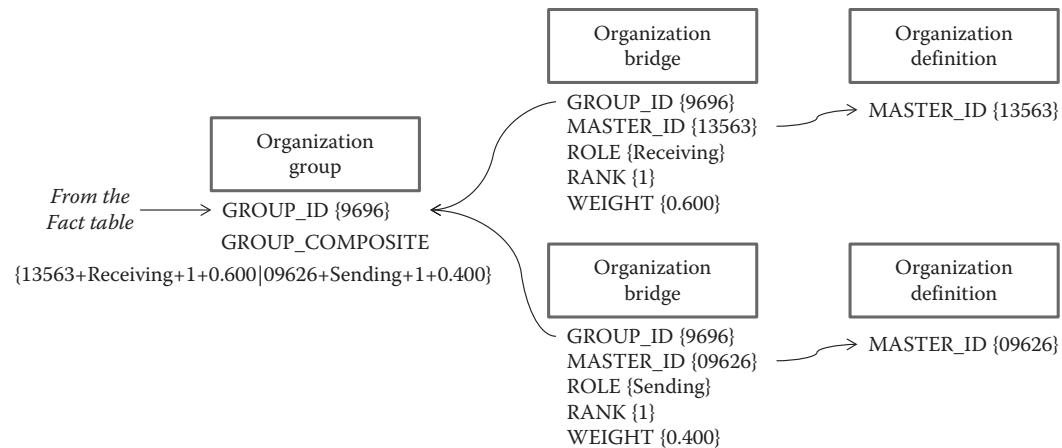


Figure 5.18 Group and bridge organization example.

Figure 5.18 depicts a hypothetical fact connected to two different definitions in the Organization dimension. One of those organizations is defined in the “Receiving” role, weighted at 60% of any aggregation. The other organization is defined in the “Sending” role, with the remaining 40% of any aggregation. The weighting factor in the bridges allows for correct aggregation of quantitative facts. Suppose a clinical procedure charge of \$100 was in the fact table pointing to this example group. Aggregating that fact up to the indicated organizations without using the bridge weighting factors would produce an incorrect result where each organization is allocated the entire \$100, and the grand total for the aggregation is inflated to \$200. A correct aggregation query would multiply the \$100 value by the weighting factors, resulting in \$60 being allocated to one organization, and \$40 to the other. Each individual aggregate total would be correct, and all higher-level totals would also be correct.

We don’t know in advance how many bridge entries will be defined in any particular group, so it isn’t possible to accurately predict how much an aggregate total will be inflated if aggregation queries are performed incorrectly. Had there been four organizations in the example group, the total would have been inflated by \$300 rather than only \$100. Correct use of the weighting factor in each bridge prevents this problem. The design constraint that the sum of the weighting factors in every group must equal one results in a correct and complete allocation of any quantitative fact into any calculated aggregate values.

Note the value of the Group Composite in Figure 5.18. The Group ID can be retrieved by querying on the composite value “13563+Receiving+1+0.600|09626+Sending+1+0.400” without needing to access the Bridge table. The four pieces of data that constitute each bridge are plus-delimited, while the different bridges are pipe-delimited. This design feature of being able to find an existing group without needing to cursor through the Bridge table will be a key element in generalizing the ETL logic in the chapters that follow. The Group table for each

dimension is created using an SQL CREATE statement that varies only by the name of the table being made specific to each dimension:

```
CREATE TABLE PROCEDURE_G
(
    GROUP_ID          INTEGER      NOT NULL,
    GROUP_COMPOSITE VARCHAR(640) NOT NULL
);
```

After creating the group table, the bridge table CREATE typically follows:

```
CREATE TABLE PROCEDURE_B
(
    MASTER_ID        INTEGER      NOT NULL,
    GROUP_ID         INTEGER      NOT NULL,
    BRIDGE_ROLE     VARCHAR(20)  NOT NULL,
    BRIDGE_RANK     INTEGER      NOT NULL,
    BNridge_WEIGHT DECIMAL(4,3) NOT NULL
);
```

By varying the names of the tables in these statements, all of the Group and Bridge tables in your warehouse can be created very quickly.

Interconnections: Hierarchy Tables

As the standard design pattern has unfolded, we've seen how natural and system keys can be used to access dimension definition rows through the context and bridge constructs. The group and bridge constructs provide access to the definitions from each fact. The hierarchy table in each dimension provides a mechanism for relating pairs of definition entries. It is the final piece of the six-table pattern, providing a way for different definition rows to refer to each other.

Each row in the Hierarchy table represents an ancestor-descendent relationship between two constructs in the Definition table. The relationship is established by two foreign keys in the Hierarchy table, both pointing to the Master ID column of the definition table. Each entry identifies the type of hierarchy structure that is being defined between the ancestor and the descendent in the Structure column. This value determines most of the edit criteria that the ETL should apply to the entry when it is defined. The three types of structural relationships, in increasing order of risk, are

- *Tree*. In a tree structure, any particular node can have only one parent, and no node can become a descendent of itself. This is the safest type of structure to navigate in a query because no node will be double-counted no matter how the hierarchy is navigated.

- **Directed Acyclic Graph (DAG).** In a DAG structure, a node can have any number of different parent nodes, and no node can become a descendent of itself (i.e., it is *acyclic*). This type of structure offers some risk during query navigation because a node could be double-counted as entries are aggregated up through multiple parent nodes.
- **Network.** In a network structure, a node can have any number of parents, and there is no restriction on an ancestor node eventually becoming its own descendent. This type of structure can easily provide unexpected or incorrect query results if the same nodes appear in many places in the network.

When most of us think of a hierarchy, we think of a tree-type diagram, such as an organization chart. The entries in the hierarchy table represent each node-to-node connection—ancestor and descendent pair—possible in the tree diagram we are imagining, meaning there are many rows in the table for any particular hierarchy we might imagine. The Perspective column in the hierarchy table ([Figure 5.19](#)) provides a name for the entire hierarchy image that emerges when all of the hierarchy rows in the same perspective are accumulated. The Relation column provides the name of the specific ancestor–descendent relationship within the perspective. The hierarchy table can be used to define an arbitrary number of user- or source-defined perspectives, and those different perspectives are not required to agree with each other. The notion of safely traversing a hierarchy or DAG without encountering a cyclic error only applies *within* each perspective. Querying *across* perspectives always runs the risk that the data will form a Network structure regardless of how any of the included perspectives were defined.

[Figure 5.19](#) shows the additional columns that provide internal control and rigor in the Hierarchy table. The Depth From Ancestor (DFA) column records the number of hierarchic levels between the ancestor and the descendent entries. If the DFA is zero, the ancestor and descendent nodes are the same entry. Integer values indicate increasing lineage, with one indicating a child, two a grandchild, three a great-grandchild, four a great-great-grandchild, and so on. If, within the same perspective, the ancestor node does not appear as a descendent of any other node, then the Origin flag is turned on. If, within the same perspective, the descendent node has no descendent nodes of its own, then the Terminus flag is turned on.

[Figure 5.20](#) provides an abbreviated example of the Hierarchy table entries (right) required to support the simple tree structure (left). Note that the number of rows in the table is greater than the number of connecting lines in the tree diagram. When we draw tree structures, we diagram only direct parent–child relationships (i.e., those where the DFA would be one), with deeper lineage implied by the structure of the tree. In the hierarchy table, both explicit and implicit ancestor–descendent relationships must be defined. Node A always has its Origin flag turned on, while nodes D, E, and F always have their Terminus flags turned on.

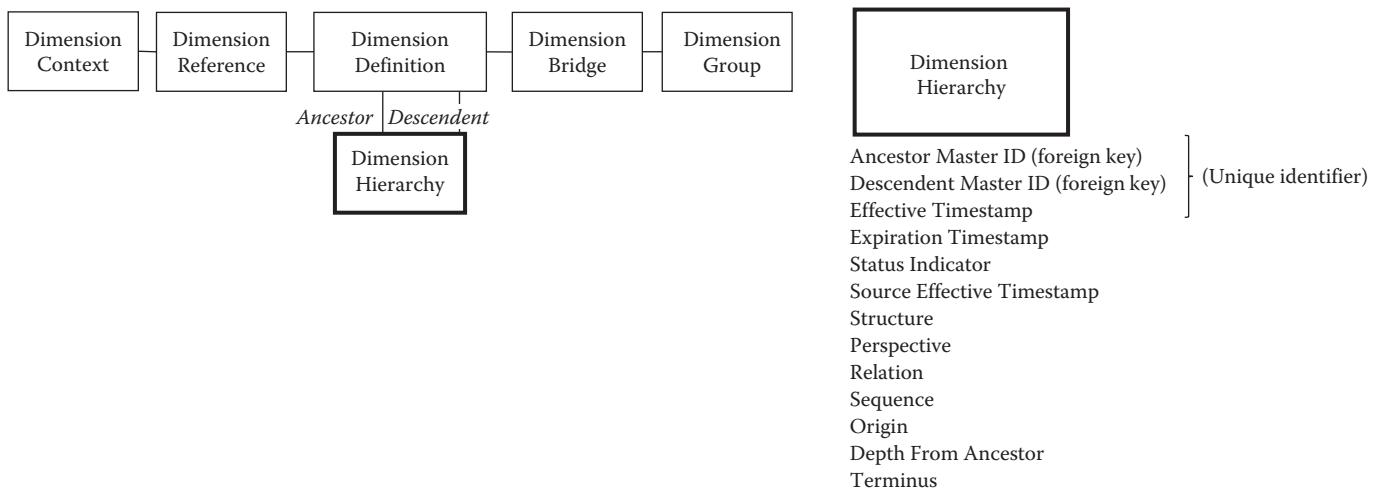


Figure 5.19 Hierarchy table.

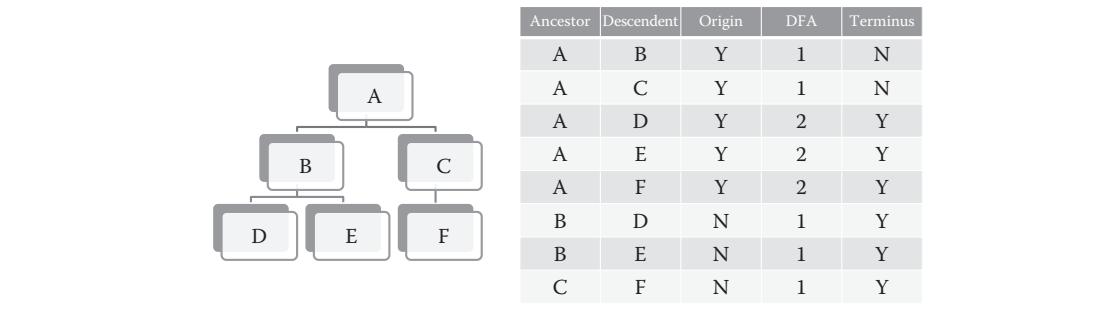


Figure 5.20 Dimension fiat hierarchy example.

Hierarchy definitions take advantage of controls for SCDs in the same way that entries do in the Definition table. The logic is the same as used in definitions, with the exception that since the Hierarchy table doesn't have its own surrogate key as the Definition table does with the Definition ID, the Hierarchy table uses the effective timestamp as part of its primary key in order to support multiple time-sliced variations on each hierarchy relationship. Any hierarchy entry can be managed as slowly-changing except natural hierarchies which are omitted because they are defined on key structures that can't change once established.

Natural Hierarchies

The generic warehouse ETL can automatically create tree entries for pairs of definition constructs where one has a key structure in the reference table that is a subset of the reference table entry for another definition construct. These auto-generated ancestor–descendent relationships are referred to as a *natural* hierarchy because the relationship is naturally embedded in the key structures. Other hierarchies are described as *fiat* hierarchies because the relationship between ancestor and descendent is known only because it is explicitly asserted by a source data feed (i.e., by “fiat”).

There are two control flags in the context table that determine how keys in the context will be interpreted for natural hierarchy processing. Natural hierarchies occur when one context defines a source key structure that is a natural subset of another source key. If the Natural Hierarchy Internal Flag is turned on, the ETL will automatically create hierachic relationships between the two keys *in the same context* where one instance of the key (the ancestor) is a subset of another (the descendent), delimited by the character in the Natural Hierarchy Internal Delimiter column of the context table (Figure 5.21). It's the internal natural hierarchy that allows automatic hierarchies to be built by the ETL for codes like the ICD-9 (e.g., 384 is parent of 384.2).

If the Natural Hierarchy Crossover Flag in the context table is turned on, the ETL will automatically create hierachic relationships between two keys *in different contexts* where one instance of the key is a subset of the other (Figure 5.22). It is the crossover natural hierarchy that allows automatic hierarchies to be built by the generic ETL for compound keys that overlap (e.g., “USA” is parent of “USA|FL”).

Natural Hierarchy Internal Flag = 'Y'
Natural Hierarchy Internal Delimiter = ?

Context Key 01	Context Key 02	Context Key 03	Context Key 04	...
123				
123.4				
123.4.5				

Figure 5.21 Internal natural hierarchy example.

Natural Hierarchy Crossover Flag = 'Y'

Context Key 01	Context Key 02	Context Key 03	Context Key 04	...
USA				
USA	FL			
USA	FL	Orlando		

Figure 5.22 Crossover natural hierarchy example.

Ref. Label 01	Ref. Label 02	Ref. Label 03	Ref. Label 04	...
Country				
Country	State			
Country	State	City		

Figure 5.23 Context reference labels for crossover natural hierarchy example.

For crossover hierarchies to be processed, the crossover flag must be turned on in both associated contexts and the overlapping Reference Labels must match (Figure 5.23). Matching the labels prevents two keys from being matched based only on their values being the same. Matching labels assure that the *meanings* of those values are the same, as well. Figure 5.23 illustrates the Reference Labels that might be appropriate for the crossover hierarchy example in Figure 5.22. If the label values didn't match across the three contexts, it would mean that the matching of the actual reference values is only coincidental. The contexts determine the meaning of the reference values, and, therefore, determine whether or not a crossover hierarchy entry is correct and meaningful. The hierarchy table for each dimension is created using an SQL CREATE statement that varies only by the name of the table being made specific to each dimension:

```
CREATE TABLE TEST.PROCEDURE_H
(
    ANCESTOR_MASTER_ID      INTEGER        NOT NULL,
    DESCENDENT_MASTER_ID    INTEGER        NOT NULL,
    CLASS                   VARCHAR(7),
    PERSPECTIVE              VARCHAR(64)    NOT NULL,
    RELATION                 VARCHAR(64),
    ORIGIN_FLAG               VARCHAR(1),
)
```

```

DEPTH_FROM_ANCESTOR      INTEGER      NOT NULL,
TERMINUS_FLAG             VARCHAR(1),
STATUS_INDICATOR          INT,
EFFECTIVE_TIMESTAMP       DATETIME,
EXPIRATION_TIMESTAMP      DATETIME,
SOURCE_EFFECTIVE_TIMESTAMP DATETIME
);

```

By varying the name of the table in this statement, all of the hierarchy tables in your warehouse can be created very quickly.

Connecting to Facts

With the full dimensional model implemented, the 26 dimensions are finally brought together in the warehouse Fact table (Figure 5.24). The Fact table is given its own ascending surrogate key as a Fact ID, and includes the 26 foreign keys needed to refer back to the dimensions. The foreign keys in the fact table reference the Group IDs of the associated dimensions, with the exception of the Control dimensions that don't use the group construct because a fact can only connect to one entry in those dimensions. Instead, the foreign key to each Control dimension is to the dimension Master ID.

The two nonkey columns in the fact table are the actual value column, and the source timestamp. The value column will contain the actual qualitative or quantitative measure of interest. The source timestamp is assigned in the ETL based on the timestamp supplied as part of the source dataset for that purpose. This timestamp represents when the fact value is to be considered the current value in the source system that sent the data to the warehouse. It is not necessarily the timestamp that would be used if a query is needed to

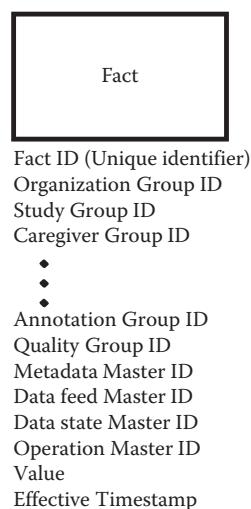


Figure 5.24 Fact table.

associate the fact to the dimension entries that were active at the time of the fact. The temporal period for the fact is defined in the association of the fact to the Calendar dimension (in the Event role). If the original fact is updated in the source system, a new fact with its own source timestamp will be stored in the fact table, still referencing the same Calendar entry.

The fact table is created using an SQL CREATE statement that only needs to be used once if you implement my recommendation to keep your design at one fact table:

```
CREATE TABLE TEST.FACT (
    FACT_ID              INT NOT NULL AUTO_INCREMENT,
    ORGANIZATION_GROUP_ID INT NULL,
    SUBJECT_GROUP_ID     INT NULL,
    ENCOUNTER_GROUP_ID   INT NULL,
    CAREGIVER_GROUP_ID   INT NULL,
    CALENDAR_GROUP_ID    INT NULL,
    CLOCK_GROUP_ID       INT NULL,
    ACCOUNTING_GROUP_ID INT NULL,
    DIAGNOSIS_GROUP_ID   INT NULL,
    MATERIAL_GROUP_ID    INT NULL,
    PROCEDURE_GROUP_ID   INT NULL,
    FACILITY_GROUP_ID   INT NULL,
    QUALITY_GROUP_ID     INT NULL,
    UOM_GROUP_ID         INT NULL,
    METADATA_MASTER_ID   INT NULL,
    DATAFEED_MASTER_ID   INT NULL,
    DATASTATE_MASTER_ID   INT NULL,
    OPERATION_MASTER_ID   INT NULL,
    VALUE                VARCHAR(800),
    SOURCE_TIMESTAMP      DATETIME,
    PARTITION_KEY        VARCHAR(64),
    PRIMARY KEY (FACT_ID)
);
```

I allow the foreign keys in the fact table to be NULL to allow for easy development and testing, even in situations where you'll be developing code and test data for facts that don't need to use all of the dimensions. In the long-term, you won't have any NULL keys in the fact table, but allowing for them in the short-term makes implementation go much more smoothly.

Dimension Navigation

As a result of this common star dimension design pattern, each dimension of the warehouse has a standard navigation path that can be used to query data in the fact table. The pathway runs from the outer Context table through to the Fact table (Figure 5.25).

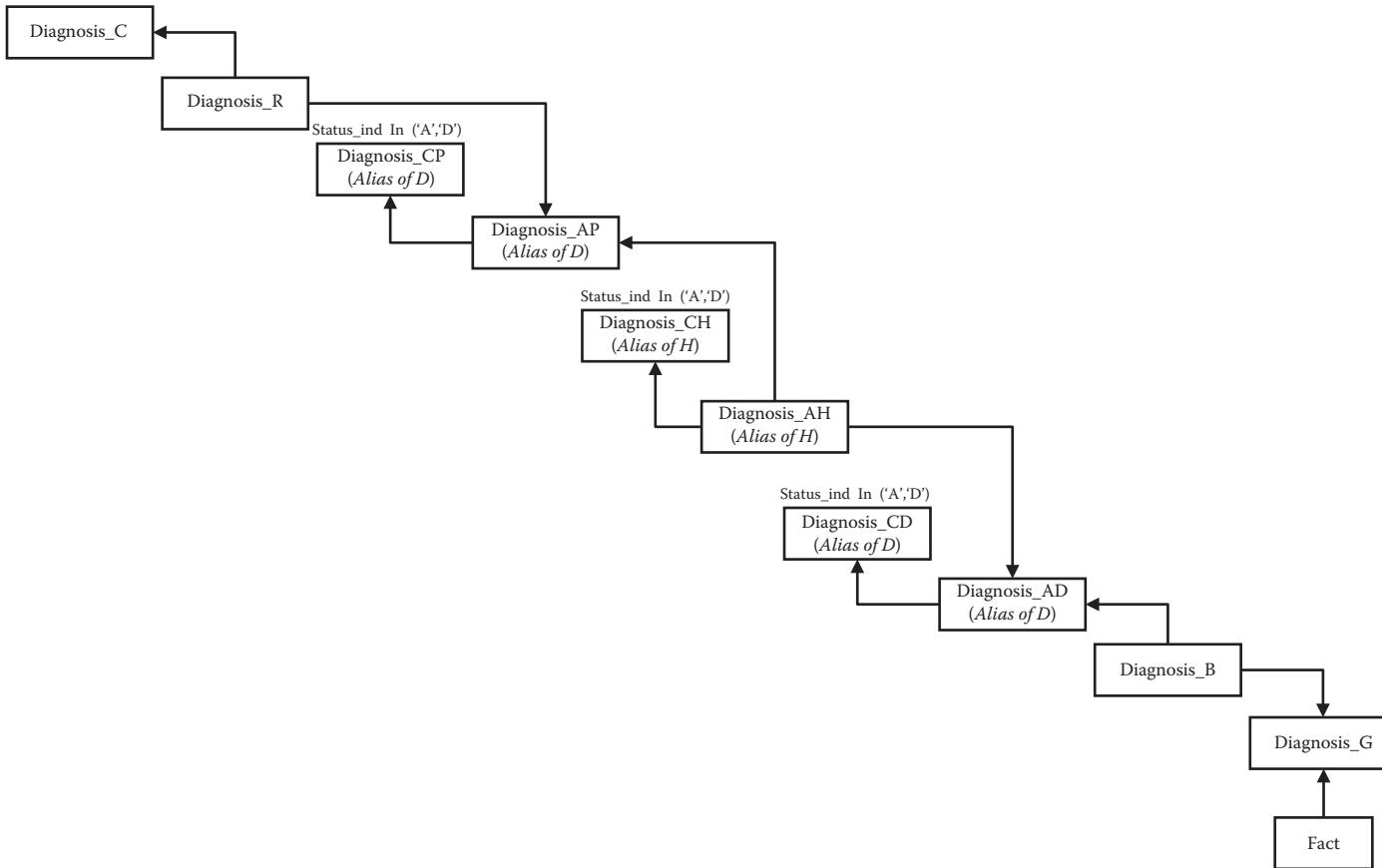


Figure 5.25 Navigating the star design pattern.

The pathway supports navigation of hierarchies in the data, as well as differentiating current and past SCD versions of each definition and hierarchy entry. Navigation across a dimension always includes the hierarchy table so users won't have to know in advance whether hierarchies have been defined against data definitions that they might be using in their queries. The standard query joins also presume that a user query might be looking for both current and "As Of" data in the definitions, again without having to know whether the slowly changing controls are being used for any of the definitions that might be retrieved. Users can traverse a dimension in their queries without any prior knowledge of which features of this design pattern have been used to implement the data they are interested in.

Navigating this common pathway requires traversing different aliases of the Definition and Hierarchy tables as they serve different purposes at different parts of the queries (Figure 5.26):

- *Current Ancestor (CA)*. The currently active view of the definition of interest in a query. The status indicator will typically be Active, although a Deleted entry would also be considered the current active view of a definition.
- *As of Ancestor (AA)*. The set of definition entries of interest in a query, including the current ancestor plus any previously superseded versions of the same entry.
- *As of Descendent (AD)*. The set of all definition entries that are hierarchical descendants of any of the ancestor definitions.
- *Current Descendent (CD)*. The currently active view of the hierarchical descendants of the ancestor definition set, either Active or Deleted.
- *As of Hierarchy (AH)*. The set of all hierarchy definitions, including both currently active and historically superseded versions.
- *Current Hierarchy (CH)*. The hierarchy entries that are currently Active or Deleted.

These alias views of the definition and hierarchy tables allows users to implement the widest range of queries that can be expected against any dimension. The vast majority of queries will navigate the pathway looking at

		Definition			Hierarchy
		Ancestor	Descendent		
SCD variant	Current	CA view	CD view	CH view	Hierarchy
	As Of	AA view	AD view	AH view	

Figure 5.26 Alias traversal views of definition and hierarchy tables by SCD variant.

currently active data. These queries tend to qualify joins on the Active status indicator while navigating through the pathway:

```
SELECT *
  FROM DIAGNOSIS_C C
 INNER JOIN DIAGNOSIS_R R
   ON (C.CONTEXT_ID = R.CONTEXT_ID)
 INNER JOIN DIAGNOSIS_D CA
   ON (R.MASTER_ID = CA.MASTER_ID
       AND CA.STATUS_INDICATOR = 'A')
 INNER JOIN DIAGNOSIS_H CH
   ON (CA.MASTER_ID = CH.ANCESTOR_MASTER_ID
       AND CH.STATUS_INDICATOR = 'A')
 INNER JOIN DIAGNOSIS_D CD
   ON (CH.DESCENDENT_MASTER_ID = CD.MASTER_ID
       AND CD.STATUS_INDICATOR = 'A')
 INNER JOIN DIAGNOSIS_B B
   ON (CD.MASTER_ID = B.MASTER_ID)
```

Alternatively, a query can navigate through the pathway using an “As Of” time obtained from the source timestamp of each fact. This type of query ignores the status indicators and instead joins on the basis of the fact’s source timestamp being between each definition’s effective and expiration timestamps:

```
SELECT *
  FROM DIAGNOSIS_C C
 INNER JOIN DIAGNOSIS_R R
   ON (C.CONTEXT_ID = R.CONTEXT_ID)
 INNER JOIN DIAGNOSIS_D AA
   ON (R.MASTER_ID = AA.MASTER_ID
       AND FACT.EFFECTIVE_TIMESTAMP
           BETWEEN AA.EFFECTIVE_TIMESTAMP AND AA.EXPIRATION_TIMESTAMP)
 INNER JOIN DIAGNOSIS_H AH
   ON (AA.MASTER_ID = AH.ANCESTOR_MASTER_ID
       AND FACT.EFFECTIVE_TIMESTAMP
           BETWEEN AH.EFFECTIVE_TIMESTAMP AND AH.EXPIRATION_TIMESTAMP)
 INNER JOIN DIAGNOSIS_D AD
   ON (AH.DESCENDENT_MASTER_ID = AD.MASTER_ID
       AND FACT.EFFECTIVE_TIMESTAMP
           BETWEEN AD.EFFECTIVE_TIMESTAMP AND AD.EXPIRATION_TIMESTAMP)
 INNER JOIN DIAGNOSIS_B B
   ON (AD.MASTER_ID = B.MASTER_ID)
```

Finally—but much less frequently—a query will select a result that includes both the currently active and historically superseded data for a definition. This type of

query needs the entire set of definition aliases described above in order to return all of the expected data in wide result rows:

```
SELECT *
FROM DIAGNOSIS_C C
INNER JOIN DIAGNOSIS_R R
  ON (C.CONTEXT_ID = R.CONTEXT_ID)
INNER JOIN DIAGNOSIS_D CA
  ON (R.MASTER_ID = CA.MASTER_ID
    AND CA.STATUS_INDICATOR = 'A')
INNER JOIN DIAGNOSIS_D AA
  ON (R.MASTER_ID = AA.MASTER_ID
    AND FACT.EFFECTIVE_TIMESTAMP
      BETWEEN AA.EFFECTIVE_TIMESTAMP AND AA.EXPIRATION_TIMESTAMP)
INNER JOIN DIAGNOSIS_H AH
  ON (AA.MASTER_ID = AH.ANCESTOR_MASTER_ID
    AND FACT.EFFECTIVE_TIMESTAMP
      BETWEEN AH.EFFECTIVE_TIMESTAMP AND AH.EXPIRATION_TIMESTAMP)
INNER JOIN DIAGNOSIS_D CD
  ON (CH.DESCENDENT_MASTER_ID = CD.MASTER_ID
    AND CD.STATUS_INDICATOR = 'A')
INNER JOIN DIAGNOSIS_D AD
  ON (AH.DESCENDENT_MASTER_ID = AD.MASTER_ID
    AND FACT.EFFECTIVE_TIMESTAMP
      BETWEEN AD.EFFECTIVE_TIMESTAMP AND AD.EXPIRATION_TIMESTAMP)
INNER JOIN DIAGNOSIS_B B
  ON (AD.MASTER_ID = B.MASTER_ID)
```

Using queries like these can navigate any permutations of current, historical, or combined data from each of the dimensions. Note that the Group table is omitted from the examples above because the Group table exists to facilitate group lookup during fact loading in the ETL. There's nothing of interest to a user in that table, so we typically omit it from pathway discussions. I usually don't define the table to the view I create in my Business Intelligence semantic layer. Generally, facts can be joined directly to the bridge table using the Group ID.

That completes the description of the design pattern for a dimension in the warehouse, including what the Fact table looks like in the center of the star, and the resulting standard pathway that emerges for navigating through data in the warehouse. The database can typically be generated in about an hour, and you can now turn your attention to loading the Alpha version.

Chapter 6

Loading Alpha Version

The whole point of the Alpha version is to have some form of the data warehouse available almost immediately upon starting the implementation project. Previous chapters have looked at getting the database up and running, as well as identifying and analyzing some of the initial data to be loaded. The only thing remaining is to implement the code for actually loading those data into our preliminary database. Because the database design is extremely generic, it won't take a lot of complicated distinct code. We're not trying to prepare a production-ready version at this point, so we will take some coding shortcuts, but the algorithm for quick loading is simple enough to repeatedly clone the same code for each source we want to load in this version. Most of the learning occurs with the earliest sources, and the subsequent Alpha sources help us refine our understanding. The Alpha warehouse should be available for querying within days, within the first week of the project for the earliest sources, and the first month for the rest.

Throw-Away Code

The main purpose of the Alpha version is for everyone associated with the warehouse development effort to learn the warehouse model and how institutional data are going to fit into that model. It is *not* about creating a system that then gets incrementally altered until it's ready to be placed into production: The code being created in this chapter must be considered throw-away code. We're going to create a coded Alpha version of the warehouse that loads by brute-force, and that discards a lot of perfectly good data because we're not going to slow down to handle any nuances.

If you read ahead to the Beta version chapters, you'll see a design for a fully generic ETL subsystem that can handle any data with any nuance. Please don't be tempted to jump ahead. It would be dangerous to embark on that process until you really understand the model and its components precisely because the

generic Beta version will rely on a level of abstraction that hides most of the details of the model from everyone. Building this Alpha version will provide that knowledge so the things that are happening in the Beta version will make sense when you get there. Consider this version more about training the warehouse development team than about actual warehouse development.

Selecting and Preparing Sources

To build the Alpha version quickly, you'll need to select sources of data to which you already have access. The Alpha version gets loaded very quickly, and is ultimately not retained going forward, so you don't want to spend a lot of time working to gain access to data that wouldn't otherwise be available easily to your team. The emphasis in the Alpha version is to access some data very quickly, and get the new warehouse loaded in a week or so. You want the most representative data you can get your hands on, and the highest priority is to gain quick and early access.

Most initial Alpha loads start with some basic dimensional master data. Some forms of Patient or Encounter data, or both, are typically among the first data loads. What we really want in the Alpha load are facts; but few facts get loaded without some form of reference to patients and those encounters, so getting some of those master data dimensions loaded early offers some quick payback. After that, the other dimensions you choose to load will depend upon the dimensionalization needs of your available facts.

I often choose to load my first batch of facts into the Alpha version by generating facts off the master loads I've already completed. Remember, you're learning about the model by doing this; you're not trying to find the definitive clinical facts that will change the world. Simple is good, if you can get it quickly. Presuming you've found a source that allowed you to load patient and encounter data into the Subject and Interaction dimensions, you probably already have some facts at your fingertips: *dated events*. Every date in a dimension row is a signal that some event happened on that date. You can generate a corresponding fact for each of those dates. If you've got patients and encounters, you've probably got

- A birth fact for each patient
- A death fact for some patients
- An admission fact for each inpatient encounter
- A discharge event for most inpatient encounters
- A registration event for each ambulatory encounter
- A departure event for most ambulatory encounters

That's six quick and easy facts that can be drawn from master data you've probably already decided to load into the Alpha version. These facts are excellent learning tools because they correspond to real events you know about, they're

conditional on the associated date not being NULL, some require filtering on other columns in the source (e.g., inpatient vs. ambulatory), and they're easy to test after loading because the facts should align precisely with the dimension definitions; and the query to map those two sides to each other is very standard. Easy to get, easy to load, easy to test!

Another early choice that is probably available to you, if you found patient data, is the residential address data for those patients. Most patient datasets include some form of address data, sometimes several addresses for home or work. I like to load residential and work address data in Alpha because it typically requires building groups in the Geopolitics dimension that are simple enough to be built easily and quickly, but also complex enough to really teach us how to do bridges and groups. The dimension data is also highly hierarchical, so it usually gives you your first opportunity to build some hierarchy entries. Address data in the sourced patient table will give us one or more facts, one for each address in the data, often home and work. As with event date facts, you can practice loading only facts for addresses that are present in the source, remembering not to load facts for null data.

Remember that all of this is supposed to be happening very quickly, so you'll need to consider dependencies among all of these data sources in order to keep things moving. Of the ideas generated so far, here's the order in which I'd implement them, testing each stage before going on to the next:

1. *Load patients*: Limit the first load to about 100 patients so you get extremely fast execution, and a simpler set of test results. Since this cycle of Alpha loads is used repeatedly as you learn the model, wipe out data, and try again; try adding another 2000 patients to each cycle until you've got all patients loaded. You want the number of patients to keep increasing because the data that are loaded about patients will be a mosaic, and you don't know for certain which combinations of data will actually occur in the patients you load. For example, the first 100 patient might all be alive, or will have had only ambulatory encounters. As you increase your sample, the needed combinations that represent each branch of your load design will tend to take care of themselves.
2. *Load calendar*: The facts you'll load will have variable dimensionality; but the calendar is never missing, so you'll need calendar entries for anything you want to load. Ensure that any dates you need for facts you want to load are present since your Alpha code will rely heavily on inner joins in connecting facts with dimensions, and if dates are missing from the dimension, the data will fall out of the load. This is why you get your calendar data from the datasets you're already loading. For this pass, take the distinct set of nonnull dates of birth and death in any patients who are being loaded.
3. *Load patient birth facts*: These facts can be created for every patient already loaded into the Subject dimension. For every loaded patient who has a nonnull date of birth, load a fact that's also dimensionalized to the calendar

for that date of birth. The fact itself is *factless* (i.e., a logical fact that simply represents the connections among the dimensions without needing any value in the value column) since you're just connecting the patient to the date, with the Event set to "Birth" in the metadata.

4. *Load patient death facts:* These facts can be created for every patient already loaded into the Subject dimension who has a nonnull date of death. The code for this load is a virtual clone of the birth fact load, with the event changed to "Death" and the source coming from the date of death column. This similarity is why you want to ensure you've completed the birth fact load before beginning the death fact load. Don't clone code that hasn't been tested; it'll just slow you down. Once the birth fact load works, it'll only take ten minutes before the death facts are loaded.
5. *Load encounters:* Unlike patient loads, don't limit the encounter load to any particular number of entries. Load all encounters in your source into the Interaction dimension for any patients you've already loaded into your Subject dimension. It's okay if you've loaded patients who haven't had any encounters, but you don't want the reverse for now. If too few encounters load, go back and load more patients.
6. *Load calendar:* You can clone the calendar load we used earlier, but this time you want all of the distinct nonnull event dates for registration, admission, departure, and discharge in the encounters you've loaded to the Interaction dimension.
7. *Load admission facts:* For any encounter you've loaded that fits the criteria you've established for an *inpatient encounter* and has a nonnull admission date, generate a new fact dimensionalized to the encounter (in Interaction), the patient (in Subject), and the admission date (in Calendar). The metadata event can be assigned the "Admission" value. The code for this load will be cloned several times in the next few steps, so be sure to test it thoroughly before continuing.
8. *Load registration facts:* Clone the admission fact load, changing the selection filter to *ambulatory encounters* with a nonnull registration date. Source the registration date to create your new fact, using "Registration" as the metadata event.
9. *Load discharge facts:* Clone the admission fact load again, sourcing any nonnull discharge date to create your new facts, using "Discharge" as the metadata event.
10. *Load departure facts:* Clone the registration fact load (which already filters on ambulatory encounters), sourcing the departure date to create your new facts, and using "Departure" as the metadata event.
11. *Load countries:* Select all of the distinct nonnull country codes from the patient source data and load them into a Country subdimension in Geopolitics. If there is no country code associated with an address in the source, you'll have to decide what countries you want to represent in your Alpha load. If you know all of the addresses will be from one country,

then invent a code for yourself and load that code into the dimension. In my work, I typically see data that are predominantly from the United States, with Canada addresses making up most of the exceptions; so, if there is no explicit country code, I might load two entries into the dimension: USA and CAN.

12. *Load states:* Select all of the distinct nonnull combinations of Country and State from the patients that have been loaded. If I was forced to invent countries in the prior step, I'll hard-assign them here. I might use "USA" for any address with a 5-digit postal code, and "CAN" for all others. Load all of these distinct combinations into the Geopolitics dimension as a State subdimension. Many entries won't be correct, but it'll keep the Alpha load moving. Some of the state codes will be invalid as well, but don't worry about that for now. You'll see all of that bad data in your test queries, but you have to load the bad dimension data or else the bad facts that originated that bad data won't load.
13. *Load cities:* Select all of the distinct nonnull combinations of Country, State, and City in any patient address. If necessary, assign the country code using the same logic as used for states. Load these entries into the City subdimension of Geopolitics. At this point, cities are likely to be fully spelled out since you're getting them from full addresses, but that's okay. The name of the city becomes a key, but can be mapped to coded structures later if necessary. This means that misspelled names will end up as new cities, but that's one of the things that makes querying the Alpha version so interesting. It gives us a chance to see how messy some of our source data might be, and that knowledge will be useful in the more formal analysis we conduct later for the Beta version.
14. *Load postal zones:* Select all distinct nonnull country code and postal code combinations in any patient address. If necessary, once again assign the country code using the same logic as used for states and cities. Load these combinations into the Postal Zone subdimension of Geopolitics. Since this load involves a two-part key, it goes fastest if you clone code for the State load since that load also involved a two-part key.
15. *Load calendar:* Every fact is dimensionalized to Calendar, so sometimes you have to be creative in the choice of dates that you want to source. Source the address data for patients next, but it's unlikely that the patient data you have include a specific date that tells you when the address was updated. Many source datasets include a date that the information in each row was last updated, if so, that date will be fine. You know the record was correct then, so the address must have been correct on that date. It might have been correct earlier, but you have no way to know that. For patients with nonnull addresses, select all of the distinct nonnull update dates (or whatever other column you've decided to use) and load them into the Calendar dimension.

16. *Load home address facts:* For each nonnull home address in each loaded patient, select the country, state, city, postal code, and update date. Use these to connect a new fact to the Calendar for the update date, and to Geopolitics using a two-bridge group for the city and postal code. Remember that city implies state and country, so no explicit bridge to them is needed, because the dimension hierarchy in Geopolitics will take care of that. For the fact itself, set the value to “~Redacted~” in order to indicate that the value is HIPAA protected and shouldn’t be viewed in a deidentified test system. In the final production warehouse, you’ll load the address into the Fact value, and the business intelligence layer will automatically replace that value with “~Redacted~” when you query in a deidentified view of the warehouse. That functionality won’t be built until the Gamma version, so for now simply put that value in the Fact table yourself. Since you used the record updated date as your Calendar entry, I recommend that the metadata event be set to “Snapshot” instead of “Address”. Because you don’t actually know the event that resulted in the patient row being updated, and you don’t want to artificially indicate an event occurred that really didn’t. I use the “Snapshot” event whenever I capture data in a source based only on the update date. In this case, I’d set the Attribute to “Home Address” in the metadata.
17. *Load work address facts:* Clone the home address code to source the nonnull work addresses in the patient data. The logic is the same as for the home address, with “Work Address” used for the metadata attribute setting.
18. *Load geopolitics hierarchies:* Using the country, state, and city data loaded earlier into Geopolitics, load the hierarchy table entries that will connect the layers to each other: Country–State, Country–City, Country–Postal Zone, State–City.

The code needed to carry out all of these loads is laid out in the rest of this chapter. If you have access to the patient and encounter data presumed by these examples, all of the above loads can be built and tested in about 1 day. If you’re using different data, plan on approximately this level of dimension and fact complexity on the first day. Once the data are loaded, start playing with it. Have some fun! Query the warehouse to re-create the source tables for comparison. Count facts by subdimensions, qualified by properties of the dimensions. For example, which state is the home of the largest segment of patients by encounter type? Of the patients who don’t live and work in the same state, which address type most corresponds to the state where your institution is? Be creative. The point is to quickly demonstrate that, because of the generic architecture of the warehouse, almost any question you might ask of the data requires some relatively small variation on the same basic query. I refer to this 1-day slice of the Alpha version of the warehouse as a *proof-of-concept*. The Alpha version development

will continue, but the general visualization of the model that some people might be looking for from the warehouse team is now available for sharing.

Alpha development continues along two parallel tracks. On the first track, you want to keep identifying more data that can be sourced and loaded into the warehouse. The main emphasis is on loading lots of facts of different types. Your main constraint is that you want to select facts for patients and encounters you are already loading, although that constraint opens up quickly as you reduce your load limitation on patients in each iteration of your testing. Look for facts that users will find most interesting. In my experience, a load of lab results and medication administrations into your Alpha version allows for some very meaningful analysis to take place even in this extremely early version of the warehouse. If you get medications, try to get allergies, too. I like to load allergy data very early precisely because I have found it to be very error prone in every healthcare institution where I've implemented this model. It's very interesting to query the warehouse for mismatches between medication and allergy data. I'm always told there won't be any, but there are always lots of results to look through. It's a quick way to demonstrate warehouse value using every early data.

The second track you'll follow in your Alpha loading is fact enrichment. This applies to those first-day facts you loaded, as well as anything you've loaded since then. Facts are meaningful to the extent that they are heavily dimensionalized. To keep the Alpha loading simple and fast, I always try to load facts dimensionalized only to patient, encounter, and calendar because that code for the load is very standard, and I have lots of pre-tested code to clone. I also look for other dimensional data in my sources that I could come back and use to enrich the facts I've already loaded. For example, let's suppose those encounters you loaded the first day had included the attending physician in the data. You ignored it on the first day, but perhaps you'd like to include it in a subsequent round of testing your load. You might start by looking for a source of data for providers that would include those attending physicians identified by the same keys that the encounter data source includes. If you find one, you can load that source into the Alpha version as new Caregiver dimension data. If you don't have a data source, you can load it into Alpha by taking distinct values from the encounters you've loaded and load them into the dimension. Since many clinical systems have denormalized their data to include physician names along with their keys in the transactional data, this option often isn't as bad as it seems. However, if the encounter source data are well normalized, you should try to find a provider table to use as a source for your caregiver load. Either way, with caregivers loaded, you can go back and modify the original code that is loading admissions, discharges, registrations, and departures to also dimensionalize the resulting facts to the attending physician in the Caregiver dimension. This enrichment of the fact could slow

down the first-day loads if you tried to fully dimensionalize those early facts. Since we want to be able to purge the data in Alpha and reload at any time, I prefer to enrich facts later when opportunities arise that won't slow down the overall effort at loading the Alpha version.

Always be on the lookout for enrichment data in your Alpha sources. Beyond caregiver, the encounter data probably also include the associated nursing unit for the encounter, or even the individual bed. If so, load units and beds into facility either by, as with the attending physician above, finding an easily accessed source for the data or by selecting the distinct values from the encounter data, and update the fact loads to include facility. Note that some analysis is needed here: Assuming the encounter includes a bed assignment, that bed wasn't necessarily the admission bed. You might consider only loading beds to discharge facts since the bed in the encounter probably didn't change post-discharge. Perhaps you could enrich the admission facts with only the nursing unit. Just keep in mind that you're trying to learn the model in this version. As long as the way you enrich the facts is reasonable, it doesn't have to be flawless to serve its purpose.

Continue to enrich the data you've already loaded, and keep adding simple variations of new data. Cycle through the data continuously, each time adding more simple facts and enriching facts from prior iterations. Your emphasis is on loading facts, but you'll find that you end up loading many subdimensions into the dimensions to support your fact enrichment. Within a few days, you'll have dozens of different types of facts that are starting to look dimensionally complete. Within a few weeks, you might have 50–100 different types of facts, and your fact table will be measured in the millions of rows. After a month, the fact table will have tens of millions of rows, and you'll be able to start seeing a fairly complete picture of each encounter, including ADT data, lab tests, allergies, medications, imaging data, surgical procedures, and vital signs. If you run into data that you're not quite sure how you want to dimensionalize, then load them in each of the different ways you are considering (with different metadata assignments) so you can query the data in different ways and see which way makes sense. The Alpha version is only limited by the availability of new data to source. Play with the data as much as you can. The point is to learn the model, not to go into production. The more variations you include in the ways you load your data, the more you'll learn from this Alpha version.

Generating Surrogate Keys

A critical component of loading data into the new Alpha version warehouse, and therefore the very first piece of code you need, is the ability to generate surrogate key values that will serve as identifiers in the various dimension tables. Because the same value is needed in multiple places during a load, You'll need

to be able to generate those values prior to actually inserting your data into those tables. To accomplish this, create a specific table for this purpose:

```
CREATE TABLE NEW_SURROGATES (
    PROCESS      VARCHAR(64),
    DIMENSION    VARCHAR(20),
    SUBDIMENSION VARCHAR(64),
    CONTEXT_ID   INT,
    COMPOSITE    VARCHAR(772),
    NEW_SURROGATE INT NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (NEW_SURROGATE)
);
ALTER TABLE NEW_SURROGATES AUTO_INCREMENT = 10001;
```

This table will be used to generate all of the surrogate keys required anywhere in any of the dimensions of the warehouse; including Master IDs and Definition IDs in the Definition tables, and Group IDs in the Group tables. The columns of this New Surrogates table provide mechanisms for assigning surrogate keys to many types of entries at the same time. The Process column is an arbitrary value that you will assign in each block of code that you create in order to allow multiple blocks of code to be executing in the Alpha ETL system at the same time. The Dimension, Subdimension, Context ID, and Composite columns represent the data for which a new surrogate is being assigned. The use of these columns will vary based on the type of key you are generating. Finally, note that the code establishes a new seed value for the auto-incremented value at 10,001. This represents the minimum value that you'll allowed to be assigned to a new dimension entry. Values below 10,001 are being reserved for warehouse configuration data, most of which won't be added until the Beta version, although a few entries will occur in the Alpha version. Using this value, you'll be able to easily reset the warehouse during testing by simply deleting all rows with identifiers greater than 10,000.

In order to obtain new surrogate values for entries in a source table that is to be loaded to the warehouse, the key values from the source are simply passed through this table first. Suppose you have an address table that contains addresses for all of your patients or locations. Each address has a country code within it, and you'd like to use this table to load your Country subdimension of the Geopolitics dimension. You can extract all the country codes from the table, and insert them into the New Surrogates table to assign new keys for the load:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS = "Alpha Country"

INSERT INTO NEW_SURROGATES (PROCESS, DIMENSION, SUBDIMENSION, CONTEXT_ID, COMPOSITE)
SELECT "Alpha Country", "Geopolitics", "Country", COUNTRY_CODE FROM SOURCE.ADDRESS
INNER JOIN GEOPOLITICS_C C ON CONTEXT_NAME = "Country Code";
```

Note that the first DELETE statement removes any entries already in the New Surrogates table for this process; most likely from a previous execution or test.

This deletion can also be done immediately following the test in which the new keys are created, but I find that it is sometimes helpful to have them left in the table after my test to aid in diagnosing any problems encountered during a test run. Generally I delete as late as possible, although if I've created millions of identifiers in a single load, I often clean up my data sooner independent of any subsequent loading. Regardless, I still execute the Delete just before the Insert just in case there's some old data lingering in the table.

There are several problems with this simple block of code. First, the country code isn't the identifier for rows in the source address table, so the same country is likely to appear many times in the source table. You only want to create a new dimension entry for each *distinct* country. In general, it's always safest to presume that extracting keys for new surrogates should be based on the set of distinct values that define the composite key for which a new surrogate value is being generated.

Second, it's possible that you've already loaded one or more of those countries into the dimension. Don't load them again if they are already there; our source extraction should only look for country codes that aren't already present in the dimension. You will typically eliminate the duplicates with a left outer join to the Reference table in the dimension to ensure that the target composite isn't already present. Third, to keep the code as simple as possible, I usually omit the use of the Dimension and Subdimension columns of the New Surrogate table if there is only a single value for each within the Process value, as in this case.

The following code will assign a new surrogate identifier to every distinct country code in the source table that hasn't already been loaded into the warehouse:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS = "Alpha Country"

INSERT INTO NEW_SURROGATES (PROCESS, CONTEXT_ID, COMPOSITE)
SELECT DISTINCT "Alpha Country", C.CONTEXT_ID, S.COUNTRY_CODE
FROM SOURCE.ADDRESS S
INNER JOIN GEOPOLITICS_C C ON CONTEXT_NAME = "Country Code"
LEFT JOIN GEOPOLITICS_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
    AND R.CONTEXT_KEY_01 = S.COUNTRY_CODE)
WHERE R.REFERENCE_COMPOSITE IS NULL;
```

You should examine this sample code very carefully because it contains the core elements that you'll use repeatedly while building your Alpha warehouse version. It takes the distinct values of some natural key from a source table, ensures that those identifiers aren't already loaded into the dimension, and assigns a new identifier to each. Those identifiers will end up in the Reference, Definition, Hierarchy, and Bridge tables five times as Master IDs, and once in the Definition table as a Definition ID. Now that you are able to generate your identifiers, you can start loading data into the warehouse.

Simple Dimensions and Facts

Among the early data sources identified for the Alpha version, there are probably a few tables that represent master data that can be loaded into the dimensions. There might also be some fact-based sources that contain foreign keys to master tables that have not been selected for loading in the Alpha version. You'll want to use those foreign keys in those fact sources to load our Alpha version dimensions as well.

First Source: Patient Master Data

We'll jump in with an easy one: Suppose we have an available source table that contains master data for Patients that we can load into the Subject dimension. Columns might include, among others:

Medical Record Number
First Name
Last Name
Date of Birth
Gender Code

While this isn't all the data you'd want to know about patients, it'll do for an Alpha load just fine. The source has a single row for each dimension entry you want, and the Medical Record Number (MRN) was defined as a target Context in the last chapter. The other four columns were defined as definitional columns in the Subject Definition table when we created our warehouse database. Everything is in place for a quick Alpha load.

The first step of a new load is the assignment of new surrogate identifiers to the source keys that will be loading. We'll call our process "Alpha Patient," and take the set of distinct MRNs from our source table.

```
DELETE FROM NEW_SURROGATES WHERE PROCESS = "Alpha Patient"

INSERT INTO NEW_SURROGATES (PROCESS, CONTEXT_ID, COMPOSITE)
SELECT DISTINCT "Alpha Patient", C.CONTEXT_ID, S.MEDICAL_RECORD_NUMBER
FROM SOURCE.PATIENT S
INNER JOIN SUBJECT_C C ON CONTEXT_NAME = "MRN"
LEFT JOIN SUBJECT_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
    AND R.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
WHERE R.REFERENCE_COMPOSITE IS NULL;
```

Note that this code is almost exactly the same as our sample code earlier for establishing new surrogate values. The Process name is appropriate to the data we're loading, and the source column and context name have been altered to reflect this case. We can then load our new Reference table entries. This step doesn't actually need the source table because all of the identifying data needed to create the Reference entries is in the New Surrogates table, along with the

newly assigned surrogate identifiers for each patient. All we need to do is read them back and insert them into the Reference table:

```
INSERT INTO SUBJECT_R
  (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE, CONTEXT_KEY_01)
SELECT NS.CONTEXT_ID,
       NS.NEW_SURROGATE,
       NS.COMPOSITE,
       NS.COMPOSITE
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = "Alpha Patient";
```

In the case of these patients, as well as much of the data that will end up being processed into the warehouse, the Medical Record Number is the entire natural key. This allows your Insert to place the sourced Reference Composite into both the Reference Composite and Context Key 01 of the new Reference row. Later you'll handle more complicated data that have multiple parts to the natural key, and will place different data into these columns. Next, you insert the Definition entries. The surrogate keys we need will be taken from the New Surrogates table again, but for Definitions we also join back to the source table to pick up the various defining or descriptive columns that we have chosen to load.

```
INSERT INTO SUBJECT_D
  (SUBDIMENSION,
   MASTER_ID,
   DEFINITION_ID,
   DESCRIPTION,
   VERNACULAR_CODE,
   FIRST_NAME,
   LAST_NAME,
   DATE_OF_BIRTH,
   GENDER)
SELECT "Person",
       NS.NEW_SURROGATE,
       NS.NEW_SURROGATE,
       MAX(LAST_NAME) + "," + MAX(FIRST_NAME),
       S.MEDICAL_RECORD_NUMBER,
       MAX(FIRST_NAME),
       MAX(LAST_NAME),
       MAX(DATE_OF_BIRTH),
       MAX(GENDER_CODE)
  FROM SOURCE.PATIENT S
 INNER JOIN NEW_SURROGATES NS
    ON (NS.PROCESS = "Alpha Patient"
        AND NS.COMPOSITE = S.MEDICAL_RECORD_NUMBER)
 GROUP BY NS.NEW_SURROGATE, S.MEDICAL_RECORD_NUMBER;
```

Note that our selection of data in the query requires that the data be grouped by the New Surrogate and MRN values since the process that created the surrogates

took distinct entries from the source. That code was written to allow for the same MRN to be present in the source table more than once in order to avoid any errors associated with duplicate entries. Because of this, your load code needs to use that GROUP BY clause to ensure that only one entry is created in the dimension for each MRN, even if there were multiple entries in the source table. To accomplish this syntactically, your load code takes a MAX value for each of the other columns being extracted to assure that only a single value is retrieved.

This logic has no impact on what gets loaded to the dimension if there was only a single source row for an MRN. However, some unexpected results are possible for MRNs that occur multiple times in the source. If a patient has two rows in the source table with different dates of birth, only the latest one will be used in our load because the max clause takes that latest value.

More peculiar, the name of the patient we load into the dimension might not ultimately be the name of either instance of the patient in the source table. For example, if one name was “Walter Jones” and the other was “Steven Taylor,” the name loaded to the dimension will be “Walter Taylor.” As your max logic takes the greater values for the First Name and Last Name independently. That’s okay because in the Alpha version we want data loaded quickly, without stopping to address nuances. Since they most likely reflect errors in your source systems, different birthdays and names for the same patient in different rows of the source table are simply considered too nuanced to worry about in this Alpha version. The Beta version will handle it just fine, and since we’re throwing this code away in a couple of weeks, we don’t want to stop and customize code to handle obscure conditions.

Note also that the Gender column in the Subject Definition table gets populated with the Gender Code from the source data. The Beta version will include logic to decode the inbound coded value so the Definition row will ultimately contain “Male” or “Female” in the Gender column regardless of how any particular source system coded the data. For the Alpha version, we simply take the coded value.

Next, you will insert the single-bridge Group entries for the Definition rows you loaded. As with the Reference entries, you don’t need the source table for this operation. You simply want to create a Group for every new surrogate created. Ordinarily, you will typically only associate Groups with Fact loads, but we create this single-entry group during the Alpha version load precisely because it will allow our Alpha version fact loads to be simplified by not having to worry about finding or creating groups that turn out to require only the single default bridge.

```
INSERT INTO SUBJECT_G
  (GROUP_ID,
   GROUP_COMPOSITE)
SELECT NS.NEW_SURROGATE,
       NS.NEW_SURROGATE + "+Subject+1+1.0"
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = "Alpha Patient";
```

Next, you insert the actual single-entry Bridges. The data are the same as in the Group, but the Role, Rank, and Weight are discrete columns rather than a character composite as in the Group load.

```
INSERT INTO SUBJECT_B
  (GROUP_ID,
   MASTER_ID,
   ROLE, RANK, WEIGHT)
SELECT NS.NEW_SURROGATE,
      MS.NEW_SURROGATE, "Subject", 1, 1.0
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = "Alpha Patient";
```

Finally, you insert the self-referencing entries in the Hierarchy table. Every entry in the Definition table needs a row in the Hierarchy that allows the row to point to itself when users query via the Hierarchy. The entry only needs the surrogate identifier, so once again the source table is not needed:

```
INSERT INTO SUBJECT_H
  (ANCESTOR_MASTER_ID,
   DESCENDENT_MASTER_ID,
   PERSPECTIVE, DEPTH_FROM_ANCESTOR)
SELECT NS.NEW_SURROGATE,
      NS.NEW_SURROGATE, "~Self~", 0
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = "Alpha Patient";

DELETE FROM NEW_SURROGATES WHERE PROCESS = "Alpha Patient";
```

This last step includes the `DELETE` statement that clears out the no-longer-needed new surrogates. It can be included here to keep the New Surrogate table clear, or it can be omitted to keep the data available for debugging. If so, the initial `DELETE` statement above will clean out those values if the code should be run again. I recommend regularly running blocks of code like these a second time as a test of the logic that makes sure that entries don't get added to the dimension if they are already there. This means that if the code is run a second time, no data should be modified or inserted to the dimension by the second execution. Verifying this logic works is an important part of the integration testing of this Alpha version.

Alternative G-B-H Processing

As you accelerate your loading of data into the Alpha version, you'll find that you are cloning many variants of this code for each new source that you identify. The last three steps that created the Groups, Bridges, and Hierarchies actually didn't reference anything in the original source table. This suggests an alternative

approach to loading those three tables that wouldn't require those three steps to be included with every dimension load that gets coded. It would entail a separate small process that simply creates the necessary entries in those tables for *any* Definition entry that doesn't yet have them. These blocks of code could be executed once, after any number of dimension load executions, to finalize those three tables for anything that had been loaded into the dimension by any of the executions.

```

INSERT INTO GEOPOLITICS_G (GROUP_ID, GROUP_COMPOSITE)
SELECT D.MASTER_ID, CONCAT(D.MASTER_ID, "+Geopolitics+1+1.0")
FROM GEOPOLITICS_D D
LEFT JOIN GEOPOLITICS_G G
ON (D.MASTER_ID = G.GROUP_ID
    AND G.GROUP_COMPOSITE = CONCAT(D.MASTER_ID, "+Geopolitics+1+1.0"))
WHERE G.GROUP_ID IS NULL
AND D.MASTER_ID = D.DEFINITION_ID;

INSERT INTO GEOPOLITICS_B (GROUP_ID, MASTER_ID, ROLE_NAME, ROLE_RANK, ROLE_WEIGHT)
SELECT D.MASTER_ID, D.MASTER_ID, "Geopolitics", 1, 1.0
FROM GEOPOLITICS_D D
LEFT JOIN GEOPOLITICS_B B
ON (D.MASTER_ID = B.GROUP_ID
    AND B.ROLE_NAME = "Geopolitics"
    AND B.ROLE_RANK = 1
    AND B.ROLE_WEIGHT = 1.0)
WHERE B.GROUP_ID IS NULL
AND D.MASTER_ID = D.DEFINITION_ID;

INSERT INTO GEOPOLITICS_H
(ANCESTOR_MASTER_ID,
DESCENDENT_MASTER_ID,
PERSPECTIVE,
DEPTH_FROM_ANCESTOR)
SELECT D.MASTER_ID, D.MASTER_ID, "~Self~", 0
FROM GEOPOLITICS_D D
LEFT JOIN GEOPOLITICS_H H
ON (D.MASTER_ID = H.ANCESTOR_MASTER_ID
    AND D.MASTER_ID = H.DESCENDENT_MASTER_ID
    AND H.PERSPECTIVE = "~Self~"
    AND H.DEPTH_FROM_ANCESTOR = 0)
WHERE H.PERSPECTIVE IS NULL
AND D.MASTER_ID = D.DEFINITION_ID;

```

Note that the WHERE clauses in this new code contain a condition that wasn't needed in the base line code to ensure that the Master ID and Definition ID match. When the master load creates new surrogates identifiers and Definition rows, the two identifiers in the Definition table are assigned the same value on insertion. Since this alternative code is intended to be run at any time, sometimes long after the definition rows were created, it's possible that other processing might have occurred in the Definition table that would create rows where those two identifiers are not the same (e.g., slowly-changing dimension logic, which is typically excluded from Alpha processing). Since only the

initial rows with equal identifiers require the Group, Bridge, and Hierarchy rows to be created, the extra WHERE clause makes sure no other extraneous rows are inserted.

This kind of post-processing step—sometimes referred to as a *spider* because it crawls around in the data independent of any particular source—simplifies coding and testing, as long as you remember to run the step often. Since it does nothing if not needed, it can simply be tacked onto any dimension load processes. As long as it executes before any fact loads that would use the data, there is no need for concern.

Second Source: Patient Address Facts

Next, suppose you have an available source table that contains an address for each patient. You could use this source to extract master data for Geopolitics (Figure 6.1), and then insert a Fact entry to store the patient address dimensionalized to Subject and Geopolitics. Columns might include:

Medical Record Number
Country Code
Country Name
State Code
State Name
City Code
City Name
Street Line 1
Street Line 2

This example is simplified from a more real-world example to keep the logic fairly simple in terms of the Alpha version algorithms we’re developing. I’ve included the country, state, and city names denormalized into this data because that’s the way the data often appears. You’ll combine the two street address lines in order to keep the facts simple as well.

First, you’ll extract your natural keys in order to get your new surrogate identifiers. This sample code loads the New Surrogate table for all three subdimensions at the same time because there are advantages gained later

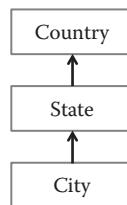


Figure 6.1 Country–State–City hierarchy for Geopolitics dimension.

when creating hierarchies for these data if all three are present in the New Surrogate table at the same time:

```

DELETE FROM NEW_SURROGATES WHERE PROCESS IN
    ("Alpha Country", "Alpha State", "Alpha City");

INSERT INTO NEW_SURROGATES (PROCESS, CONTEXT_ID, COMPOSITE)
SELECT DISTINCT "Alpha Country", C.CONTEXT_ID, S.COUNTRY_CODE
    FROM SOURCE.ADDRESS S
    INNER JOIN GEOPOLITICS_C C ON CONTEXT_NAME = "Country Code"
        LEFT JOIN GEOPOLITICS_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
            AND R.CONTEXT_KEY_01 = S.COUNTRY_CODE)
    WHERE R.REFERENCE_COMPOSITE IS NULL
UNION
SELECT DISTINCT "Alpha State", C.CONTEXT_ID,
    CONCAT(S.COUNTRY_CODE, "|", S.STATE_CODE)
    FROM SOURCE.ADDRESS S
    INNER JOIN GEOPOLITICS_C C ON CONTEXT_NAME = "State Code"
        LEFT JOIN GEOPOLITICS_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
            AND R.CONTEXT_KEY_01 = S.COUNTRY_CODE
            AND R.CONTEXT_KEY_02 = S.STATE_CODE)
    WHERE R.REFERENCE_COMPOSITE IS NULL
UNION
SELECT DISTINCT "Alpha City", C.CONTEXT_ID,
    CONCAT(S.COUNTRY_CODE, "|", S.STATE_CODE, "|", S.CITY_CODE)
    FROM SOURCE.ADDRESS S
    INNER JOIN GEOPOLITICS_C C ON CONTEXT_NAME = "City Code"
        LEFT JOIN GEOPOLITICS_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
            AND R.CONTEXT_KEY_01 = S.COUNTRY_CODE
            AND R.CONTEXT_KEY_02 = S.STATE_CODE
            AND R.CONTEXT_KEY_03 = S.CITY_CODE)
    WHERE R.REFERENCE_COMPOSITE IS NULL;

```

This example illustrates a common situation that arises in many data sources: a set of multipart natural keys that are related to each other through their natural structure. The implied hierarchy of cities being in states, and states being in countries is a common pattern in many data domains. This will make a difference when loading the Reference table entries because different columns need to be populated to represent the components of those different key structures. Since the source key to country is the one-part Country Code, the Reference load for countries looks just like the one for patients above:

```

INSERT INTO GEOPOLITICS_R
    (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE, CONTEXT_KEY_01)
SELECT NS.CONTEXT_ID,
    NS.NEW_SURROGATE,
    NS.COMPOSITE,
    NS.COMPOSITE
    FROM NEW_SURROGATES NS
    WHERE NS.PROCESS = "Alpha Country";

```

Unlike the one-part key Country, the State and City natural keys require compound keys. This will require you to join back to the source data table to pick up those components for the Reference load:

```

INSERT INTO GEOPOLITICS_R
  (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE,
   CONTEXT_KEY_01, CONTEXT_KEY_02)
SELECT NS.CONTEXT_ID,
       NS.NEW_SURROGATE,
       NS.COMPOSITE,
       S.COUNTRY_CODE,
       S.STATE_CODE
  FROM NEW_SURROGATES NS
 INNER JOIN (SELECT DISTINCT COUNTRY_CODE, STATE_CODE FROM SOURCE.ADDRESS) S
    ON (NS.COMPOSITE = CONCAT(S.COUNTRY_CODE, "|", S.STATE_CODE))
 WHERE NS.PROCESS = "Alpha State";

INSERT INTO GEOPOLITICS_R
  (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE,
   CONTEXT_KEY_01, CONTEXT_KEY_02, CONTEXT_KEY_03)
SELECT NS.CONTEXT_ID,
       NS.NEW_SURROGATE,
       NS.COMPOSITE,
       S.COUNTRY_CODE,
       S.STATE_CODE,
       S.CITY_CODE
  FROM NEW_SURROGATES NS
 INNER JOIN (SELECT DISTINCT COUNTRY_CODE, STATE_CODE, CITY_CODE FROM SOURCE.ADDRESS) S
    ON (NS.COMPOSITE = CONCAT(S.COUNTRY_CODE, "|", S.STATE_CODE, "|", S.CITY_CODE))
 WHERE NS.PROCESS = "Alpha City";

```

Next, you can load the Definition rows for your three subdimensions of almanac data:

```

INSERT INTO GEOPOLITICS_D
  (SUBDIMENSION,
   MASTER_ID,
   DEFINITION_ID,
   DESCRIPTION,
   FULL_DESCRIPTION,
   EXTENDED_DESCRIPTION,
   COUNTRY)
SELECT "Country",
       NS.NEW_SURROGATE,
       NS.NEW_SURROGATE,
       MAX(S.COUNTRY_NAME),
       MAX(S.COUNTRY_NAME),
       MAX(S.COUNTRY_NAME),
       MAX(S.COUNTRY_NAME)
  FROM SOURCE.ADDRESS S
 INNER JOIN NEW_SURROGATES NS
    ON (NS.PROCESS = "Alpha Country"
        AND NS.COMPOSITE = S.COUNTRY_CODE)
 GROUP BY NS.NEW_SURROGATE

INSERT INTO GEOPOLITICS_D
  (SUBDIMENSION,
   MASTER_ID,
   DEFINITION_ID,
   DESCRIPTION,

```

```

        FULL_DESCRIPTION,
        EXTENDED_DESCRIPTION,
        COUNTRY,
        STATE)
SELECT "State",
       NS.NEW_SURROGATE,
       NS.NEW_SURROGATE,
       MAX(S.STATE_NAME),
       CONCAT(MAX(S.STATE_NAME), " ", " ", MAX(S.COUNTRY_NAME)),
       CONCAT(MAX(S.COUNTRY_NAME), " ", " ", MAX(S.STATE_NAME)),
       MAX(S.COUNTRY_NAME),
       MAX(S.STATE_NAME)
  FROM SOURCE.ADDRESS S
INNER JOIN NEW_SURROGATES NS
  ON (NS.PROCESS = "Alpha State"
      AND NS.COMPOSITE
      = CONCAT(S.COUNTRY_CODE, "|", S.STATE_CODE))
GROUP BY NS.NEW_SURROGATE;

INSERT INTO GEOPOLITICS_D
  (SUBDIMENSION,
   MASTER_ID,
   DEFINITION_ID,
   DESCRIPTION,
   FULL_DESCRIPTION,
   EXTENDED_DESCRIPTION,
   COUNTRY,
   STATE,
   CITY)
SELECT "City",
       NS.NEW_SURROGATE,
       NS.NEW_SURROGATE,
       CONCAT(MAX(S.CITY_NAME), " ", " ", MAX(S.STATE_NAME)),
       CONCAT(MAX(S.CITY_NAME), " ", " ", MAX(S.STATE_NAME), " ", " ", MAX(S.COUNTRY_NAME)),
       CONCAT(MAX(S.COUNTRY_NAME), " ", " ",
              MAX(S.STATE_NAME), " ", " ", MAX(S.CITY_NAME)),
       MAX(S.COUNTRY_NAME),
       MAX(S.STATE_NAME),
       MAX(S.CITY_NAME)
  FROM SOURCE.ADDRESS S
INNER JOIN NEW_SURROGATES NS
  ON (NS.PROCESS = "Alpha City"
      AND NS.COMPOSITE
      = CONCAT(S.COUNTRY_CODE, "|", S.STATE_CODE, "|", S.CITY_CODE))
GROUP BY NS.NEW SURROGATE;

```

With your references and definitions loaded, we can use the alternative G–B–H loading code described above to generically fill out those remaining tables in Geopolitics. Recognizing that the data we just loaded into Geopolitics are hierarchical, we can also load our own natural hierarchy entries. Using the surrogate keys we've just generated, we'll create a hierarchy entry for every State in its Country, and every City in its State and in its Country:

```

INSERT INTO SUBJECT_H
  (ANCESTOR_MASTER_ID,
   DESCENDENT_MASTER_ID,
   PERSPECTIVE, DEPTH_FROM_ANCESTOR)

```

```

SELECT A.NEW_SURROGATE, D.NEW_SURROGATE, "~Natural~", 1
  FROM NEW_SURROGATES A
 INNER JOIN NEW_SURROGATES D
 WHERE A.PROCESS = "Alpha Country"
   AND D.PROCESS = "Alpha State"
   AND D.COMPOSITE LIKE CONCAT(A.COMPOSITE, "%")
UNION
SELECT A.NEW_SURROGATE, D.NEW_SURROGATE, "~Natural~", 2
  FROM NEW_SURROGATES A
 INNER JOIN NEW_SURROGATES D
 WHERE A.PROCESS = "Alpha Country"
   AND D.PROCESS = "Alpha City"
   AND D.COMPOSITE LIKE CONCAT(A.COMPOSITE, "%")
UNION
SELECT A.NEW_SURROGATE, D.NEW_SURROGATE, "~Natural~", 1
  FROM NEW_SURROGATES A
 INNER JOIN NEW_SURROGATES D
 WHERE A.PROCESS = "Alpha State"
   AND D.PROCESS = "Alpha City"
   AND D.COMPOSITE LIKE CONCAT(A.COMPOSITE, "%");

```

A common convention in this code is in the assignment of aliases to the tables. I typically use the letter “A” as my alias for the ancestor entry in the hierarchy I am creating, and the letter “D” as my alias for the Descendent entry. Also note that this hierarchy logic depends upon all three subdimensions being loaded from this same source and processed at the same time. If they were processed separately, the New Surrogate table would lack the necessary data to create these natural hierarchy entries so easily. When present, the logic for building these hierarchies is quite simply: The composite key for the descendent must be a subset of the composite key for the ancestor, denoted by appending the “%” operator to the ancestor composite value. If the data aren’t present in the New Surrogate table at the same time, similar logic can be built by selecting the same data from the appropriate Reference table, although the qualification of the join gets a little more complicated.

Finally, with the Geopolitics load from the Address source table complete, you can clean up your surrogate assignments:

```

DELETE FROM NEW_SURROGATES WHERE PROCESS IN
 ("Alpha Country", "Alpha State", "Alpha City");

```

It’s finally time to load something to the fact table! A patient’s street address is a fact that dimensionalizes to both Patient and Geopolitics. In reality, these facts will involve many more dimensions, but we’re keeping it simple here to illustrate the basic model and load concepts.

```

INSERT INTO FACT
(SUBJECT_GROUP_ID,
GEOPOLITICS_GROUP_ID,
VALUE)
SELECT SR.MASTER_ID, GR.MASTER_ID,
CONCAT(S.STREET_LINE_1, "; ", S.STREET_LINE_2)
FROM SOURCE.ADDRESS S
INNER JOIN SUBJECT_C SC ON (SC.CONTEXT_NAME = "MRN")
INNER JOIN SUBJECT_R SR ON (SC.CONTEXT_ID = SR.CONTEXT_ID
AND SR.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
INNER JOIN GEOPOLITICS_C GC ON (GC.CONTEXT_NAME = "City")
INNER JOIN GEOPOLITICS_R GR ON (GC.CONTEXT_ID = GR.CONTEXT_ID
AND GR.CONTEXT_KEY_01 = S.COUNTRY_CODE
AND GR.CONTEXT_KEY_02 = S.STATE_CODE
AND GR.CONTEXT_KEY_03 = S.CITY_CODE);

```

You now have an address fact for each Patient, properly dimensionalized to the correct City. This load is very simple compared to more advanced loads with more complicated data that you'll encounter later, but it illustrates the basic loading pattern. You used the Address source table to create the dimensions in order to use the dimensions to then load the facts. The process was particularly simple because you took advantage of your knowledge that the new facts you were creating only had to dimensionalize to one patient, and one city; so you knew that you could use the single-bridge groups you had created during the dimension load. Furthermore, you took advantage of your knowledge that in those cases, your dimension load used the same surrogate for both the Definition entry and the Group entry. This allowed you to identify your designated Group IDs for the facts by just looking at the Reference table. When you get to more complicated facts with multiple-bridge dimensionality, you lose that assumption, so a connection to a dimension will include many more table joins. We'll get to that later in the Lab Test example.

One very important omission in this first fact load is that it didn't use the Metadata Dimension to identify that these were, indeed, patient address facts. Without metadata, a row in the fact table is not recognizable as any particular kind of information. Here is the insert again with the metadata handled correctly:

```

INSERT INTO FACT
(SUBJECT_GROUP_ID,
GEOPOLITICS_GROUP_ID,
METADATA_MASTER_ID,
VALUE)
SELECT SR.MASTER_ID, GR.MASTER_ID, META.MASTER_ID, "~Redacted~"
FROM SOURCE.ADDRESS S
INNER JOIN METADATA_D META ON (META.SOURCE = "EMR"
AND META.EVENT = "Demographic"
AND META ENTITY = "Patient"
AND META.ATTRIBUTE = "Address")

```

```

INNER JOIN SUBJECT_C SC ON (SC.CONTEXT_NAME = "MRN")
INNER JOIN SUBJECT_R SR ON (SC.CONTEXT_ID = SR.CONTEXT_ID
    AND SR.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
INNER JOIN GEOPOLITICS_C GC ON (GC.CONTEXT_NAME = "City")
INNER JOIN GEOPOLITICS_R GR ON (GC.CONTEXT_ID = GR.CONTEXT_ID
    AND GR.CONTEXT_KEY_01 = S.COUNTRY_CODE
    AND GR.CONTEXT_KEY_02 = S.STATE_CODE
    AND GR.CONTEXT_KEY_03 = S.CITY_CODE);

```

Now, when queried, the result can identify the content of the value column in the Fact table as the Address of a Patient (i.e., Attribute of Entity). This second example also changed what is being placed in the value column of the Fact table to indicate that the value has been redacted to comply with HIPAA rules that patient addresses be treated as patient identifying information. In the Beta version, you'll implement automatic controls so patient identifying information is directly redacted in the query when a user of the warehouse is operating in a view of the warehouse that is considered deidentified. Once those controls are in place it will be safe to load facts with addresses, or other identifying information, into the warehouse. Because none of those controls exist in the Alpha version, I recommend against actually loading identifying information. Substitute the “~Redacted~” value into any loads of protected data until the Alpha version is complete. I sometimes load the identifying information temporarily if I need to test a particular piece of logic regarding the data extraction or loading, but I typically remove the data after testing are complete. Consult with your organization’s HIPAA Controller to establish a consistent policy on the use of identifying data in your development project prior to Beta version controls being in place.

Quick Fact Queries

Now that you’ve created some simple facts to associate with your patients in the Subject dimension, and cities in the Geopolitics dimension, you can pause to execute some common and basic queries against your data. These queries will test your loaded data, and reinforce your learning about facts and their relations with the dimensions. Because there’s not much data loaded so far, most of the test queries you can execute at this point involve counting things, typically facts grouped by one or more elements of the related dimensions. I typically start with a basic count of facts by metadata. This metadata-fact count is among the most common queries you’ll execute during testing of the warehouse:

```

SELECT SOURCE, EVENT, ENTITY, ATTRIBUTE, COUNT(*)
    FROM WHSE.FACT F
INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_MASTER_ID)
    GROUP BY SOURCE, EVENT, ENTITY, ATTRIBUTE
    ORDER BY SOURCE, EVENT, ENTITY, ATTRIBUTE;

```

At this point you have only one type of fact in the warehouse, so the results of this query won't become truly interesting until you've loaded much more data. If next you qualify your query to look at this particular fact, you can profile other aspects of your data by grouping your counts around dimensional properties of interest. Suppose you simply want to know what countries your patients tend to reside in:

```
SELECT GD.COUNTRY, COUNT(*)
  FROM WHSE.FACT F
INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID
                                 AND META.SOURCE      = "EMR"
                                 AND META.EVENT       = "Demographic"
                                 AND META.ENTITY      = "Patient"
                                 AND META.ATTRIBUTE   = "Address")
INNER JOIN GEOPOLITICS_D GD ON (GD.MASTER_ID = F.GEOPOLITICS_GROUP_ID)
GROUP BY GD.COUNTRY
ORDER BY GD.COUNTRY DESCENDING;
```

By adding the DESCENDING clause to your ordering, you create a ranked list; so the top entry will be a count for the country in which the largest number of your patients reside. By adding the Patient dimension to the query, you can quickly segment those country counts by patient gender:

```
SELECT GD.COUNTRY, SD.GENDER, COUNT(*)
  FROM WHSE.FACT F
INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID
                                 AND META.SOURCE      = "EMR"
                                 AND META.EVENT       = "Demographic"
                                 AND META.ENTITY      = "Patient"
                                 AND META.ATTRIBUTE   = "Address")
INNER JOIN SUBJECT_D SD ON (SD.MASTER_ID = F.SUBJECT_GROUP_ID)
INNER JOIN GEOPOLITICS_D GD ON (GD.MASTER_ID = F.GEOPOLITICS_GROUP_ID)
GROUP BY GD.COUNTRY, SD.GENDER
ORDER BY COUNT(*) DESCENDING;
```

Because you actually connected each fact to a city, and not just a country, you can just as easily segment your counts by state within each country:

```
SELECT GD.COUNTRY, GD.STATE, SD.GENDER, COUNT(*)
  FROM WHSE.FACT F
INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID
                                 AND META.SOURCE      = "EMR"
                                 AND META.EVENT       = "Demographic"
                                 AND META.ENTITY      = "Patient"
                                 AND META.ATTRIBUTE   = "Address")
INNER JOIN SUBJECT_D SD ON (SD.MASTER_ID = F.SUBJECT_GROUP_ID)
INNER JOIN GEOPOLITICS_D GD ON (GD.MASTER_ID = F.GEOPOLITICS_GROUP_ID)
GROUP BY GD.COUNTRY, GD.STATE, SD.GENDER
ORDER BY COUNT(*) DESCENDING;
```

Finally, segmenting by city requires only that the city be added to the count groupings:

```
SELECT GD.COUNTRY, GD.STATE, GD.CITY, SD.GENDER, COUNT(*)
  FROM WHSE.FACT F
  INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID
                                   AND META.SOURCE      = "EMR"
                                   AND META.EVENT       = "Demographic"
                                   AND META.ENTITY      = "Patient"
                                   AND META.ATTRIBUTE   = "Address")
  INNER JOIN SUBJECT_D SD ON (SD.MASTER_ID = F.SUBJECT_GROUP_ID)
  INNER JOIN GEOPOLITICS_D GD ON (GD.MASTER_ID = F.GEOPOLITICS_GROUP_ID)
 GROUP BY GD.COUNTRY, GD.STATE, GD.CITY, SD.GENDER
 ORDER BY COUNT(*) DESCENDING;
```

To get all of the counts at all of these levels at the same time, you could work toward a union of the results of these three queries, or you could use the Geopolitics hierarchy instead of the natural hierarchy columns in the Definition rows:

```
SELECT GAD.EXTENDED_DESCRIPTION, SD.GENDER, COUNT(*)
  FROM WHSE.FACT F
  INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID
                                   AND META.SOURCE      = "EMR"
                                   AND META.EVENT       = "Demographic"
                                   AND META.ENTITY      = "Patient"
                                   AND META.ATTRIBUTE   = "Address")
  INNER JOIN SUBJECT_D SD ON (SD.MASTER_ID = F.SUBJECT_GROUP_ID)
  INNER JOIN GEOPOLITICS_D GDD ON (GDD.MASTER_ID = F.GEOPOLITICS_GROUP_ID)
  INNER JOIN GEOPOLITICS_H GH ON (GDD.MASTER_ID = GH.DESCENDENT_MASTER_ID)
  INNER JOIN GEOPOLITICS_D GAD ON (GAD.MASTER_ID = GH.ANSCESTOR_MASTER_ID
                                   AND GAD.SUBDIMENSION IN "Country", "State", "City" )
 GROUP BY GAD.EXTENDED_DESCRIPTION, SD.GENDER
 ORDER BY COUNT(*) DESCENDING;
```

This query takes advantage of the natural hierarchy among the countries, states, and cities that you loaded in the Geopolitics dimension to count each fact against each of the participating subdimensions. Because cities roll into states, and states roll into countries, your query results will always show the counts for the countries as being larger than the counts for any particular part of the countries. There are many situations where you'll want to take advantage of this specific feature of natural hierarchies to be able to write aggregating queries without having to know the actual structure of the aggregating hierarchies in advance.

Recap of Simple ETLs

Let's recap this basic Alpha version logic before going on to more complicated and realistic examples. So far, you've loaded some master data into your dimensions; from both actual patient master tables designated for

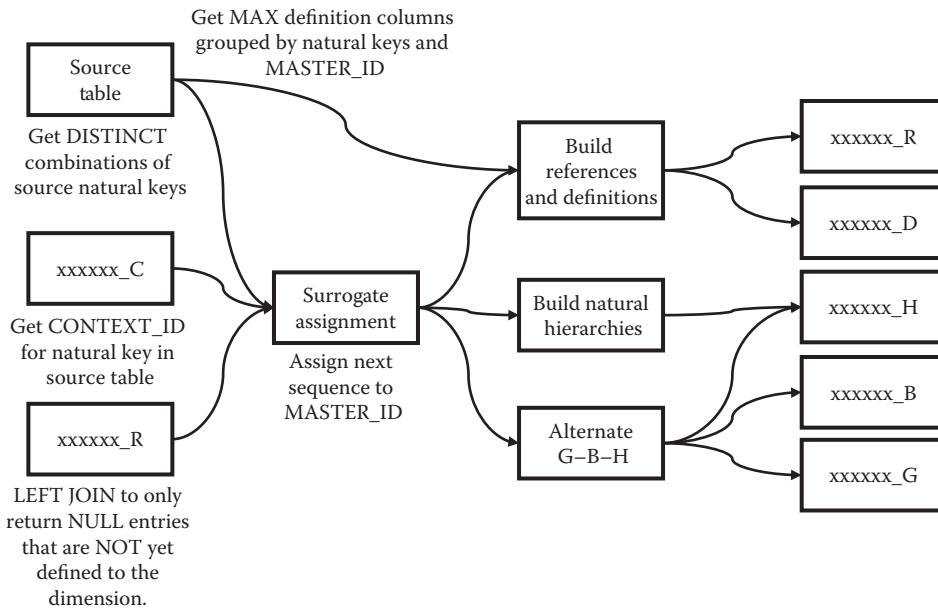


Figure 6.2 Brute-force dimension load (Alpha).

that purpose, and source address fact tables that happened to contain some denormalized dimensional data.

Figure 6.2 highlights the logic used to perform your brute-force loading of this dimensional data. First, you accessed your source data to obtain the set of distinct natural keys that you needed to assign new surrogates for. You accomplished this by passing the data through an explicit step in which you had the database assign its next sequence number to each of your natural keys, any duplicate keys having been removed via a left outer join to the Reference table to prevent keys already present in the dimension from being created again. We also extracted distinct keys to prevent the same source natural key from being assigned a new surrogate multiple times. I suggested always using the distinct clause even when analysis indicated that the rows of the source table were indeed keyed on the natural key in order to keep the quickly developing Alpha code very consistent, and avoid getting bogged down in exceptions if that analysis turned out to be incorrect.

Once surrogate keys had been generated, you were ready to insert the data into the dimension tables. Building and inserting the data into the Reference and Definition tables simply involved joining the New Surrogate table to the source table, the combination of which gave you any data needed to populate those two tables. Your first patient example involved a fairly typical one-part key of MRN, which meant that the source table actually wasn't needed to create the Reference entries, but since many keys are actually multipart, I suggest always writing code to support multipart keys to maintain consistency.

After inserting the Reference and Definition entries, you went on to insert natural hierarchy entries into the Hierarchy table. Since natural hierarchies are built using only comparisons of natural keys (e.g., the ancestor key is always a

left leading substring of the descendent key), the Hierarchy entries were built without referring back to the source data at all. Care must be taken in this approach to natural hierarchies because this algorithm depends upon having all of the ancestor and descendent keys in the New Surrogate table at the same time. If this isn't true, the load will be incomplete. The main reason it might not be true would be that the logic that loaded the dimension excluded certain values (through the left join logic) because they were already in the dimension. To avoid this, these hierarchies should only be built in this Alpha version when the dimension is starting out empty.

You then closed out the dimension load with the alternate Group–Bridge–Hierarchy (G–B–H) logic that simply inserts the necessary entries to the Group, Bridge, and Hierarchy tables without any regard to the source data, or the order in which various loads were executed against the dimension. This last G–B–H step need only be executed once after a series of dimension loads, rather than as a third step of each load, although there's no harm in running it each time if that keeps work and test scheduling simpler.

Figure 6.3 highlights the logic you then used to load your initial Address facts. You selected all of the fact value and natural key columns from your source, but unlike the dimension loads, you didn't use a distinct clause in this fact selection because you loaded multiple facts if they existed on the source table for given combinations of keys and values. You then accessed the Metadata dimension to obtain the appropriate Metadata ID for the facts being loaded in this iteration.

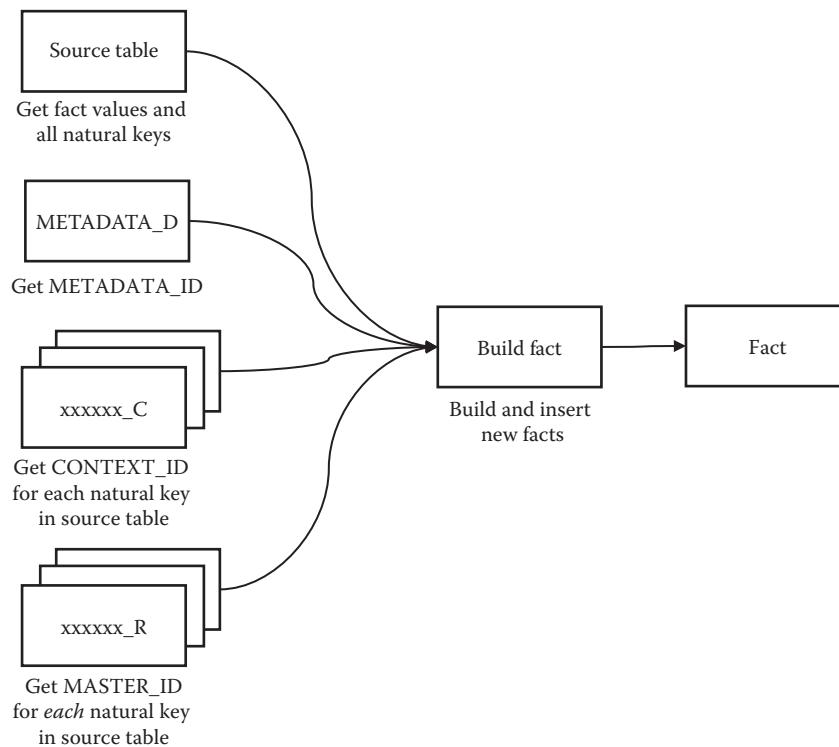


Figure 6.3 Brute-force fact load (Alpha).

You were able to obtain the Group IDs needed for the new Fact row by using the natural key data from the source to select a Master ID from the Reference table. In doing it this way, you were taking advantage of the knowledge that every Master ID in a dimension row is referenced by a default single-bridge Group where the Group ID is the same as the Master ID. This will often *not* be true as we build facts with more complicated relationships to multiple entries in the dimensions, and we'll look at those more common examples next.

Complicated Dimensions and Facts

The first two ETL load examples represented simple cases that you'll encounter repeatedly when analyzing, extracting, and loading data into the warehouse, including simple dimension loads and dimension loading as a by-product of data available in a source being used to load some early facts. Those two forms constitute the bulk of many sources that will be loaded into the early data warehouse. As the data in your warehouse gets richer and denser, you'll find more complicated sources that use more dimensions, involve more complicated bridges, and often contain qualitative entries that need to be dimensionalized beyond what has already been loaded. Each source will be unique in some way, but patterns will emerge very quickly that will allow you to keep your Alpha version loading moving fairly quickly.

Third Source: Basic Lab Results

Lab results are a wonderful example of the kind of more complicated data you'll be moving to fairly quickly in the Alpha version. The data will be more complex than the patient addresses loaded above, but that generally will only mean you have to include more code in your development. The actual new code will still be some variation on coding structures you used in the simple loads, however the load strategy is the same as with those simple loads. Let's assume you have a source table with the following columns:

Medical Record Number
Attending Provider ID
Ordered Panel ID
Lab Test ID
Clinical Result
Result Comment
Result Date
Clinical UOM
Reference High
Reference Low
Abnormalcy Indicator

In addition to having more dimensionality, the more significant complexity of these data is in the definition of the lab panel and test. All of these data end up in the Procedure dimension, and the facts we create will need to point to two procedures: the panel *ordered*, and the actual test *resulted*. Additionally, some of those resulted tests will have reference range data, and others will not. To correctly capture all of this detail for our facts, our dimensions will need to be loaded with information for the various procedures at the three levels of detail: ordered panels, resulted tests, and reference ranges. This will require building three sets of surrogate keys using our standard logic:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS IN
    ("Alpha Panel", "Alpha Lab Test", "Alpha Ref Range")
```

The logic to extract the data for new surrogates is just like what we did for our address data. The panel and lab test are each one-part natural keys, but the lab test with the reference range is not: it's a two-part natural key. On first glance, we might think the reference range would be three-part, with the Low and High values each being distinct pieces. However, upon analysis we learn that sometimes a reference value for a test is a single value (e.g., “Normal,” “Yellow”), so the conjectured third part would be missing, and would cause problems. Instead, we'll just define Reference Range as a single column in our design, and we'll concatenate the Low and High values together to create that single Reference Range.

You'll use the standard logic that we've been developing to assign surrogate keys to the different procedures, using a set of union clauses to insert all of the needed natural keys into the New Surrogates table with a single block of SQL code:

```
INSERT INTO NEW_SURROGATES (PROCESS, CONTEXT_ID, COMPOSITE)
SELECT DISTINCT "Alpha Panel", C.CONTEXT_ID, S.ORDERED_PANEL_ID
    FROM SOURCE.LAB_RESULT S
    INNER JOIN PROCEDURE_C C ON CONTEXT_NAME = "Lab Panel"
        LEFT JOIN PROCEDURE_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
            AND R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
    WHERE R.REFERENCE_COMPOSITE IS NULL
UNION
SELECT DISTINCT "Alpha Lab Test", C.CONTEXT_ID, S.LAB_TEST_ID
    FROM SOURCE.LAB_RESULT S
    INNER JOIN PROCEDURE_C C ON CONTEXT_NAME = "Lab Test"
        LEFT JOIN PROCEDURE_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
            AND R.CONTEXT_KEY_01 = S.LAB_TEST_ID)
    WHERE R.REFERENCE_COMPOSITE IS NULL
UNION
SELECT DISTINCT "Alpha Ref Range", C.CONTEXT_ID,
    CONCAT(S.LAB_TEST_ID, "|", S.REFERENCE_LOW, "-", S.REFERENCE_HIGH)
    FROM SOURCE.LAB_RESULT S
```

```

INNER JOIN PROCEDURE_C C ON CONTEXT_NAME = "Reference Range"
LEFT JOIN PROCEDURE_R R ON (C.CONTEXT_ID = R.CONTEXT_ID
    AND R.CONTEXT_KEY_01 = S.LAB_TEST_ID
    AND R.CONTEXT_KEY_02 = CONCAT(S.REFERENCE_LOW, "-", S.REFERENCE_HIGH))
WHERE R.REFERENCE_COMPOSITE IS NULL;

```

To create the Reference table entries, the Panel and Lab Test entries require only a one-part natural key, so the Source table isn't needed. Both sets of entries can be inserted to the Reference table with a single command

```

INSERT INTO PROCEDURE_R
    (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE, CONTEXT_KEY_01)
SELECT NS.CONTEXT_ID,
    NS.NEW_SURROGATE,
    NS.COMPOSITE,
    NS.COMPOSITE
FROM NEW_SURROGATES NS
WHERE NS.PROCESS IN ("Alpha Panel", "Alpha Lab Test");

```

The Reference Range entries use a two-part key, so they'll use our more generalized code which brings the source data back in:

```

INSERT INTO PROCEDURE_R
    (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE, CONTEXT_KEY_01, CONTEXT_KEY_02)
SELECT NS.CONTEXT_ID,
    NS.NEW_SURROGATE,
    NS.COMPOSITE,
    S.LAB_TEST_ID,
    CONCAT(S.REFERENCE_LOW, "-", S.REFERENCE_HIGH)
FROM NEW_SURROGATES NS
INNER JOIN SOURCE.LAB_RESULT ON (NS.COMPOSITE
    = CONCAT(S.LAB_TEST_ID, "|", S.REFERENCE_LOW, "-", S.REFERENCE_HIGH))
WHERE NS.PROCESS = "Alpha Ref Range";

```

Note that as you proceed, the scale of all of this data will be determined by the dimensional scope of the facts you're looking to load. The facts will be more complicated, but these dimensional loads will follow the straightforward logical flow. There's more code, but not more complexity to the coding. You'll follow the same basic logic to load the Definition entries, needing the source data for the Reference Ranges, but not for the Panels and Lab Tests:

```

INSERT INTO PROCEDURE_D
    (SUBDIMENSION,
    MASTER_ID,
    DEFINITION_ID,
    DESCRIPTION,
    FULL_DESCRIPTION,
    EXTENDED_DESCRIPTION)
SELECT "Lab Panel",
    NS.NEW_SURROGATE,
    NS.NEW_SURROGATE,
    NS.COMPOSITE,

```

```

CONCAT("Panel: ", NS.COMPOSITE),
CONCAT("Panel: ", NS.COMPOSITE))
FROM NEW_SURROGATES NS
WHERE NS.PROCESS = "Alpha Panel"

UNION
SELECT "Lab Test",
NS.NEW_SURROGATE,
NS.NEW_SURROGATE,
NS.COMPOSITE,
CONCAT("Lab Test: ", NS.COMPOSITE),
CONCAT("Lab Test: ", NS.COMPOSITE))
FROM NEW_SURROGATES NS
WHERE NS.PROCESS = "Alpha Lab Test"

INSERT INTO PROCEDURE_D
(SUBDIMENSION,
MASTER_ID,
DEFINITION_ID,
DESCRIPTION,
FULL_DESCRIPTION,
EXTENDED_DESCRIPTION)
SELECT "Reference Range",
NS.NEW_SURROGATE,
NS.NEW_SURROGATE,
CONCAT(S.LAB_TEST_ID, "(", S.REFERENCE_LOW, "-", S.REFERENCE_HIGH, ")"),
CONCAT("Lab Test: ", S.LAB_TEST_ID,
      ", Range: ", S.REFERENCE_LOW, "-", S.REFERENCE_HIGH),
CONCAT("Lab Test: ", S.LAB_TEST_ID,
      ", Range: ", S.REFERENCE_LOW, "-", S.REFERENCE_HIGH)
FROM SOURCE.LAB_RESULT S
INNER JOIN NEW_SURROGATES NS
ON (NS.PROCESS = "Alpha Ref Range"
    AND NS.COMPOSITE
    = CONCAT(S.LAB_TEST_ID, "|", S.REFERENCE_LOW, "-", S.REFERENCE_HIGH))
GROUP BY NS.NEW_SURROGATE, S.LAB_TEST_ID, S.REFERENCE_LOW, S.REFERENCE_HIGH;

```

Lastly, you can create the natural hierarchy entries that connect each Reference Range row up the hierarchy to its associated Lab Test:

```

INSERT INTO PROCEDURE_H
(ANCESTOR_MASTER_ID,
DESCENDENT_MASTER_ID,
PERSPECTIVE, DEPTH_FROM_ANCESTOR)
SELECT A.NEW_SURROGATE, D.NEW_SURROGATE, "~Natural~", 1
FROM NEW_SURROGATES A
INNER JOIN NEW_SURROGATES D
WHERE A.PROCESS = "Alpha Lab Test"
AND D.PROCESS = "Alpha Ref Range"
AND D.COMPOSITE LIKE CONCAT(A.COMPOSITE, "%");

```

There are other dimensional natural keys in the source table that will also have to be loaded to their respective dimensions, but I'll leave that to you rather than repeating all of that code here. Each is a fairly standard load of

a one-part natural key. The coding is no different than what you've been working through above. Those dimensional loads include:

<i>Source Natural Key</i>	<i>Dimension</i>	<i>Subdimension</i>
Medical record number	Subject	Person
Attending provider ID	Caregiver	Provider
Result date	Calendar	Day
Clinical UOM	UOM	Unit
Abnormalcy indicator	Quality	Abnormalcy

The dimension data for these entries will end up being based exclusively on the natural key from the source since the source wasn't a master table in the first place. Note in the Procedure INSERT statements above how the stub description columns were populated for the definition rows for Panels and Lab Tests, using some literal descriptive text followed by the concatenated reference composite of natural keys. While that approach leaves definitional and descriptive columns NULL—something you certainly won't want in a production warehouse—it gets the job done by creating the required dimension entries needed to be able to load your facts. That's sufficient for your Alpha loading. If you happen to have some master tables to load instead, then by all means load them; but your Alpha loading should be mostly about facts, not dimensions.

Remember that even if one of these natural key columns has already been loaded through another load, it's always safest to build the load code again unless you are sure all of the keys have been loaded. It only takes a few minutes, and it will prevent you from losing data for keys in this fact table that didn't happen to appear in the previous table you might have loaded. Remember, the code is designed to skip dimension data already on the dimensions, so there's no real downside, and the Alpha coding is good practice.

Once you've loaded all that dimension data (it should have taken an hour or less), you're ready to load your Lab Result facts. There are actually three kinds of facts in this source, so you'll have three sets of code. Both of the first two facts are the Clinical Result, except that one of those facts is for when you have a Reference Range, and the other is for when you don't. The third fact is the Clinical Comment. You won't load the Clinical Comment if it's null.

These facts are the first example we've dealt with where there will be two dimension entries in the Procedure dimension for our facts. Your previous fact loads didn't need to build any bridges and groups because we relied on the standard single-bridge groups that we created for all entries in the dimension. Now our new facts will each require a two-bridge group for the ordered and resulted procedures. Some of the facts will require a group that combines the lab panel with the lab test, and others will require a group that combines the lab panel with the lab reference range, when available. You'll need to generate surrogates for

your new Group IDs, and you'll use the New Surrogates table in the same way you used it for your new Master IDs when loading the dimensions. You start by clearing out your surrogate generation entries from any previous executions:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS IN ("Alpha Panel-Test", "Alpha Panel-Range");
```

Then you take the distinct combinations of Panels and Lab Tests (where the Reference Range data are NULL) from the source table into your surrogate generator:

```
INSERT INTO NEW_SURROGATES (PROCESS, COMPOSITE)
SELECT DISTINCT "Alpha Panel-Test",
    CONCAT(P1C.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID "+Resulted+1+1.0")
FROM SOURCE.LAB_RESULT S
INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
    AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Test")
INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
    AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID)
LEFT JOIN PROCEDURE_G G ON (G.GROUP_COMPOSITE =
    CONCAT(P1C.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID "+Resulted+1+1.0"))
WHERE G.GROUP_COMPOSITE IS NULL
    AND S.REFERENCE_LOW IS NULL
    AND S.REFERENCE_HIGH IS NULL;
```

You then Union the distinct combinations of Panels and Lab Tests with nonNULL Reference Range data:

```
UNION
SELECT DISTINCT "Alpha Panel-Range",
    CONCAT(P1C.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID "+Resulted+1+1.0")
FROM SOURCE.LAB_RESULT S
INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
    AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Ref Range")
INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
    AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID
    AND P2R.CONTEXT_KEY_02 = S.REFERENCE_LOW
    + "-" + S.REFERENCE_HIGH)
LEFT JOIN PROCEDURE_G G ON (G.GROUP_COMPOSITE =
    CONCAT(P1C.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID "+Resulted+1+1.0"))
WHERE G.GROUP_COMPOSITE IS NULL
    AND (S.REFERENCE_LOW IS NOT NULL OR S.REFERENCE_HIGH IS NOT NULL);
```

Note that these two steps are mutually exclusive of each other, meaning that each different source combination will end up in only one of the two insert paths. The first set filtered out NULL high and low reference range values in the WHERE clause. The second set required that one of the high or low values not be NULL. As a result, every legitimate combination ended up establishing a new Group ID. There will be source combinations in some sources that don't exhibit this mutual exclusivity on the dimension inserts, but they will be rare. Double-check yourself whenever you feel that such exclusivity isn't needed. Having

assigned the new surrogate Group IDs, you can then pull the data back and insert the entries in the Group table:

```
INSERT INTO PROCEDURE_G
  (GROUP_ID, GROUP_COMPOSITE)
SELECT NS.NEW_SURROGATE, NS.COMPOSITE
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS IN ("Alpha Panel-Test", "Alpha Panel-Range");
```

Creating the needed Bridge table entries is a bit more complicated than inserting the Group table entries because the Bridge entries need the Master ID of the associated definition row, and those aren't in the New Surrogates table. You'll need to include the source table and connections to the Definition table in order to resolve the necessary keys.

First you can insert the Bridge entries for the ordered panels found in the source rows that didn't have the Reference Range data:

```
INSERT INTO PROCEDURE_B
  (GROUP_ID, MASTER_ID, ROLE, RANK, WEIGHT)
SELECT NS.NEW_SURROGATE, P2R.MASTER_ID, "Lab Test", 1, 1.0
  FROM SOURCE.LAB_RESULT S
INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
                                AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Test")
INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
                                AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID)
INNER JOIN NEW_SURROGATES NS ON (NS.COMPOSITE
                                  = CONCAT(P1R.MASTER_ID + "+Ordered+1+0.0|", P2R.MASTER_ID "+Resulted+1+1.0"))
WHERE NS.PROCESS = "Alpha Panel-Test";
```

Next, we can UNION in the Bridge entries for the lab tests into those same source rows that did have Reference Range data:

```
UNION
SELECT NS.NEW_SURROGATE, P1R.MASTER_ID, "Ordered", 1, 0.0
  FROM SOURCE.LAB_RESULT S
INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
                                AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Test")
INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
                                AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID)
INNER JOIN NEW_SURROGATES NS ON (NS.COMPOSITE
                                  = CONCAT(P1R.MASTER_ID + "+Ordered+1+0.0|", P2R.MASTER_ID "+Resulted+1+1.0"))
WHERE NS.PROCESS = "Alpha Panel-Test";
```

Recognize that it took two inserts to create the necessary bridges for these groups. No matter how many bridges are needed in a group, there will always be only a single row inserted in the Group table. A single Group entry represents a list of roles of arbitrary complexity, but each of those Bridges needs to be inserted individually. The larger and more complex the group, the more inserts required creating the bridges.

Next, you can Union in the Bridge entries for the ordered panels found in source rows that did have Reference Range data:

```
UNION
SELECT NS.NEW_SURROGATE, P2R.MASTER_ID, "Resulted", 1, 1.0
  FROM SOURCE.LAB_RESULT S
    INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
    INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
                                   AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
    INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Ref Range")
    INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
                                   AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID
                                   AND P2R.CONTEXT_KEY_02 = S.REFERENCE_LOW
                                   + "-" + S.REFERENCE_HIGH)
  INNER JOIN NEW_SURROGATES NS ON (NS.COMPOSITE
= P2R.MASTER_ID "+Resulted+1+1.0|" + P1C.MASTER_ID + "+Ordered+1+0.0")
WHERE NS.PROCESS = "Alpha Panel-Range";
```

Finally, we can Union in the bridge entries for the lab test reference ranges:

```
UNION
SELECT NS.NEW_SURROGATE, P1R.MASTER_ID, "Ordered", 1, 0.0
  FROM SOURCE.LAB_RESULT S
    INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
    INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
                                   AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
    INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Ref Range")
    INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
                                   AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID
                                   AND P2R.CONTEXT_KEY_02 = S.REFERENCE_LOW
                                   + "-" + S.REFERENCE_HIGH)
  INNER JOIN NEW_SURROGATES NS ON (NS.COMPOSITE
= P2R.MASTER_ID "+Resulted+1+1.0|" + P1R.MASTER_ID + "+Ordered+1+0.0")
WHERE NS.PROCESS = "Alpha Panel-Range";
```

With all of the Procedure dimension multiple-bridge groups now created, you can start inserting the desired Facts. These facts will continue to take advantage of the simpler logic that uses the Master ID of each Definition as the inferred Group ID of the single-bridge group in every case except the Procedure dimension. The code you'll use to insert your Lab Result facts will be structured in the same way as the earlier, and far simpler, facts you've already loaded. There will be *more* code than before because this new source table has *more* data columns. Don't be intimidated by the extra lines of SQL code. They simply repeat the very simple structure of joining a dimensionalized source data column to the Context and Reference tables of the associated dimension. That simple repetition is what will enable the generic system that you'll build in the Beta version, but for now you've got to provide all of that code. If you pay attention to what's being repeated in each dimension, the extra code isn't as scary. You'll start with the Clinical Result facts for those source rows that did not include a Reference Range:

```

INSERT INTO FACT
(SUBJECT_GROUP_ID,
CALENDAR_GROUP_ID,
CAREGIVER_GROUP_ID,
PROCEDURE_GROUP_ID,
UOM_GROUP_ID,
QUALITY_GROUP_ID,
METADATA_MASTER_ID,
VALUE)
SELECT SR.MASTER_ID,
CALR.MASTER_ID,
CR.MASTER_ID,
PG.GROUP_ID,
UR.MASTER_ID,
QR.MASTER_ID,
META.MASTER_ID,
CLINICAL_RESULT
FROM SOURCE.LAB_RESULT S
INNER JOIN METADATA_D META ON (META.SOURCE      = "EMR"
                                AND META.EVENT      = "Result"
                                AND META.ENTITY     = "Lab Test"
                                AND META.ATTRIBUTE  = "Value")
ON (SC.CONTEXT_NAME = "MRN")
ON (SC.CONTEXT_ID = SR.CONTEXT_ID
    AND SR.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
ON (CALC.CONTEXT_NAME = "Day")
ON (CALC.CONTEXT_ID = CALR.CONTEXT_ID
    AND CALR.CONTEXT_KEY_01 = S.RESULT_DATE)
INNER JOIN SUBJECT_C SC
INNER JOIN SUBJECT_R SR
INNER JOIN CALENDAR_C CALC
INNER JOIN CALENDAR_R CALR
INNER JOIN CAREGIVER_C CC
INNER JOIN CAREGIVER_R CR
INNER JOIN UOM_C UC
INNER JOIN UOM_R UR
INNER JOIN QUALITY_C QC
INNER JOIN QUALITY_R QR
INNER JOIN PROCEDURE_C P1C
INNER JOIN PROCEDURE_R P1R
INNER JOIN PROCEDURE_C P2C
INNER JOIN PROCEDURE_R P2R
INNER JOIN PROCEDURE_G PG
ON (P1C.CONTEXT_NAME = "Lab Panel")
ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
    AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
ON (P2C.CONTEXT_NAME = "Lab Test")
ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
    AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID)
ON (PG.GROUP_COMPOSITE =
    CONCAT(P1R.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID, "+Resulted+1+1.0"))
WHERE S.REFERENCE_LOW IS NULL
      AND S.REFERENCE_HIGH IS NULL;

```

Next you can insert the Clinical Result facts for those source rows that did include Reference Range data. The only difference in this code is that the WHERE clause filters the Reference Range columns differently, and the Join to the Procedure dimension for the Reference Range is correspondingly different:

```
INSERT INTO FACT  
    (SUBJECT_GROUP_ID,  
     CALENDAR_GROUP_ID,  
     CAREGIVER_GROUP_ID,  
     PROCEDURE_GROUP_ID,  
     UOM_GROUP_ID,  
     QUALITY_GROUP_ID,  
     METADATA_MASTER_ID,  
     VALUE)
```

```

SELECT SR.MASTER_ID,
       CALR.MASTER_ID,
       CR.MASTER_ID,
       PG.GROUP_ID,
       UR.MASTER_ID,
       QR.MASTER_ID,
       META.MASTER_ID,
       CLINICAL_RESULT
  FROM SOURCE.LAB_RESULT S
 INNER JOIN METADATA_D META ON (META.SOURCE      = "EMR"
                                 AND META.EVENT     = "Result"
                                 AND META ENTITY   = "Lab Test"
                                 AND META ATTRIBUTE = "Value")
                                 ON (SC.CONTEXT_NAME = "MRN")
                                 ON (SC.CONTEXT_ID  = SR.CONTEXT_ID
                                 AND SR.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
                                 ON (CALC.CONTEXT_NAME = "Day")
                                 ON (CALC.CONTEXT_ID  = CALR.CONTEXT_ID
                                 AND CALR.CONTEXT_KEY_01 = S.RESULT_DATE)
                                 ON (CC.CONTEXT_NAME = "Provider")
                                 ON (CC.CONTEXT_ID   = GR.CONTEXT_ID
                                 AND CR.CONTEXT_KEY_01 = S.ATTENDING_PROVIDER_ID)
                                 ON (UC.CONTEXT_NAME = "Unit")
                                 ON (UC.CONTEXT_ID   = UR.CONTEXT_ID
                                 AND CR.CONTEXT_KEY_01 = S.CLINICAL_UOM)
                                 ON (QC.CONTEXT_NAME = "Abnormalcy")
                                 ON (QC.CONTEXT_ID   = QR.CONTEXT_ID
                                 AND QR.CONTEXT_KEY_01 = S.ABNORMALCY_INDICATOR)
                                 ON (P1C.CONTEXT_NAME = "Lab Panel")
                                 ON (P1C.CONTEXT_ID  = P1R.CONTEXT_ID
                                 AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
                                 ON (P2C.CONTEXT_NAME = "Lab Ref Range")
                                 ON (P2C.CONTEXT_ID  = P2R.CONTEXT_ID
                                 AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID
                                 AND P2R.CONTEXT_KEY_02 = S.REFERENCE_LOW
                                 + "-" + S.REFERENCE_HIGH)
                                 INNER JOIN PROCEDURE_G PG ON (PG.GROUP_COMPOSITE =
                               CONCAT(P1R.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID, "+Resulted+1+1.0"))
 WHERE S.REFERENCE_LOW IS NOT NULL
   OR S.REFERENCE_HIGH IS NOT NULL;

```

Next, you can insert the Result Comment facts, again, starting with the source rows that didn't include Reference Range data:

```

INSERT INTO FACT
  (SUBJECT_GROUP_ID,
   CALENDAR_GROUP_ID,
   CAREGIVER_GROUP_ID,
   PROCEDURE_GROUP_ID,
   QUALITY_GROUP_ID,
   METADATA_ID,
   VALUE)
SELECT SR.MASTER_ID,
       CALR.MASTER_ID,
       CR.MASTER_ID,
       PG.GROUP_ID,
       QR.MASTER_ID,
       META.MASTER_ID,
       RESULT_COMMENT
  FROM SOURCE.LAB_RESULT S

```

```

INNER JOIN METADATA_D META ON (META.SOURCE      = "EMR"
                                AND META.EVENT      = "Result"
                                AND META ENTITY    = "Lab Test"
                                AND META.ATTRIBUTE = "Comment")
INNER JOIN SUBJECT_C SC   ON (SC.CONTEXT_NAME = "MRN")
INNER JOIN SUBJECT_R SR  ON (SC.CONTEXT_ID  = SR.CONTEXT_ID
                                AND SR.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
INNER JOIN CALENDAR_C CALC ON (CALC.CONTEXT_NAME = "Day")
INNER JOIN CALENDAR_R CALR ON (CALC.CONTEXT_ID  = CALR.CONTEXT_ID
                                AND CALR.CONTEXT_KEY_01 = S.RESULT_DATE)
INNER JOIN CAREGIVER_C CC  ON (CC.CONTEXT_NAME = "Provider")
INNER JOIN CAREGIVER_R CR  ON (CC.CONTEXT_ID  = GR.CONTEXT_ID
                                AND CR.CONTEXT_KEY_01 = S.ATTENDING_PROVIDER_ID)
INNER JOIN QUALITY_C QC   ON (QC.CONTEXT_NAME = "Abnormalcy")
INNER JOIN QUALITY_R QR   ON (QC.CONTEXT_ID  = QR.CONTEXT_ID
                                AND QR.CONTEXT_KEY_01 = S.ABNORMALCY_INDICATOR)
INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID  = P1R.CONTEXT_ID
                                AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Test")
INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID  = P2R.CONTEXT_ID
                                AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID)
INNER JOIN PROCEDURE_G PG  ON (PG.GROUP_COMPOSITE =
                                CONCAT(P1R.MASTER_ID, "+Ordered+1+0.0", P2C.MASTER_ID, "+Resulted+1+1.0"))
WHERE S.REFERENCE_LOW IS NULL
      AND S.REFERENCE_HIGH IS NULL
      AND S.RESULT_COMMENT IS NOT NULL;

```

Finally, we can insert the Result Comment facts for the source rows that did include Reference Range data:

```

INSERT INTO FACT
(SUBJECT_GROUP_ID,
CALENDAR_GROUP_ID,
CAREGIVER_GROUP_ID,
PROCEDURE_GROUP_ID,
QUALITY_GROUP_ID,
METADATA_ID,
VALUE)
SELECT SR.MASTER_ID,
       CALR.MASTER_ID,
       CR.MASTER_ID,
       PG.GROUP_ID,
       QR.MASTER_ID,
       META.MASTER_ID,
       RESULT_COMMENT
FROM SOURCE.LAB_RESULT S
INNER JOIN METADATA_D META ON (META.SOURCE      = "EMR"
                                AND META.EVENT      = "Result"
                                AND META ENTITY    = "Lab Test"
                                AND META.ATTRIBUTE = "Comment")
INNER JOIN SUBJECT_C SC   ON (SC.CONTEXT_NAME = "MRN")
INNER JOIN SUBJECT_R SR  ON (SC.CONTEXT_ID  = SR.CONTEXT_ID
                                AND SR.CONTEXT_KEY_01 = S.MEDICAL_RECORD_NUMBER)
INNER JOIN CALENDAR_C CALC ON (CALC.CONTEXT_NAME = "Day")
INNER JOIN CALENDAR_R CALR ON (CALC.CONTEXT_ID  = CALR.CONTEXT_ID
                                AND CALR.CONTEXT_KEY_01 = S.RESULT_DATE)
INNER JOIN CAREGIVER_C CC  ON (CC.CONTEXT_NAME = "Provider")
INNER JOIN CAREGIVER_R CR  ON (CC.CONTEXT_ID  = GR.CONTEXT_ID
                                AND CR.CONTEXT_KEY_01 = S.ATTENDING_PROVIDER_ID)
INNER JOIN QUALITY_C QC   ON (QC.CONTEXT_NAME = "Abnormalcy")

```

```

INNER JOIN QUALITY_R QR ON (QC.CONTEXT_ID = QR.CONTEXT_ID
    AND QR.CONTEXT_KEY_01 = S.ABNORMALCY_INDICATOR)
INNER JOIN PROCEDURE_C P1C ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
    AND P1R.CONTEXT_KEY_01 = S.ORDERED_PANEL_ID)
INNER JOIN PROCEDURE_C P2C ON (P2C.CONTEXT_NAME = "Lab Ref Range")
INNER JOIN PROCEDURE_R P2R ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
    AND P2R.CONTEXT_KEY_01 = S.LAB_TEST_ID
    AND P2R.CONTEXT_KEY_02 = S.REFERENCE_LOW
    + "-" + S.REFERENCE_HIGH)
INNER JOIN PROCEDURE_G PG ON (PG.GROUP_COMPOSITE =
    CONCAT(P1R.MASTER_ID, "+Ordered+1+0.0|", P2C.MASTER_ID, "+Resulted+1+1.0"))
WHERE (S.REFERENCE_LOW IS NOT NULL
    OR S.REFERENCE_HIGH IS NOT NULL)
    AND S.RESULT_COMMENT IS NOT NULL;

```

Note the changes in the code between the inserts of the Clinical Result facts versus the Result Comment facts. You have to add an additional WHERE clause to prevent the Comment fact from loading when the value is NULL. You also must change the Attribute name in the Metadata from “Value” to “Comment” to differentiate the facts in the Fact table. Lastly, you omitted the Clinical UOM from the Comment fact since the comment is text regardless of the value of that UOM column.

You can now finalize your thinking about how you’ve loaded facts into the Alpha version of the warehouse. In [Figure 6.3](#), we saw the logical flow of an Alpha load in which all of the natural keys that needed to be dimensionalized were used in the context of only single-bridge groups. This allowed those earliest loads to simply join those keys to the dimension Reference tables in order to use the Master ID as the proxy for the known Group ID. The subsequent lab result loading was still able to take advantage of those single-bridge groups for all but the Procedure dimension. As a result of having to build those new two-bridge groups, we can now generalize the fact loading problem for this Alpha version. [Figure 6.4](#) illustrates this more extended fact loading flow by including the side-step necessary to create any needed multiple-bridge groups.

The building of facts for insertion is always a combination of Reference table lookups for single-bridge groups, and Group table lookups for multiple-bridge Groups. The ratio between these two pathways will vary, with more complex data requiring more multi-bridge Groups, but the general logic stays standard. Note, in fact, that if you coded all of the dimensions for multi-bridge Group lookups, it would still work because the single-bridge Groups are structurally consistent. They just require additional code that we know isn’t actually needed when the Master ID equals the Group ID.

The four facts you’ve now loaded for your lab test results example are

1. Clinical Result with Reference Range
2. Clinical Result without Reference Range
3. Result Comment with Reference Range
4. Result Comment without Reference Range

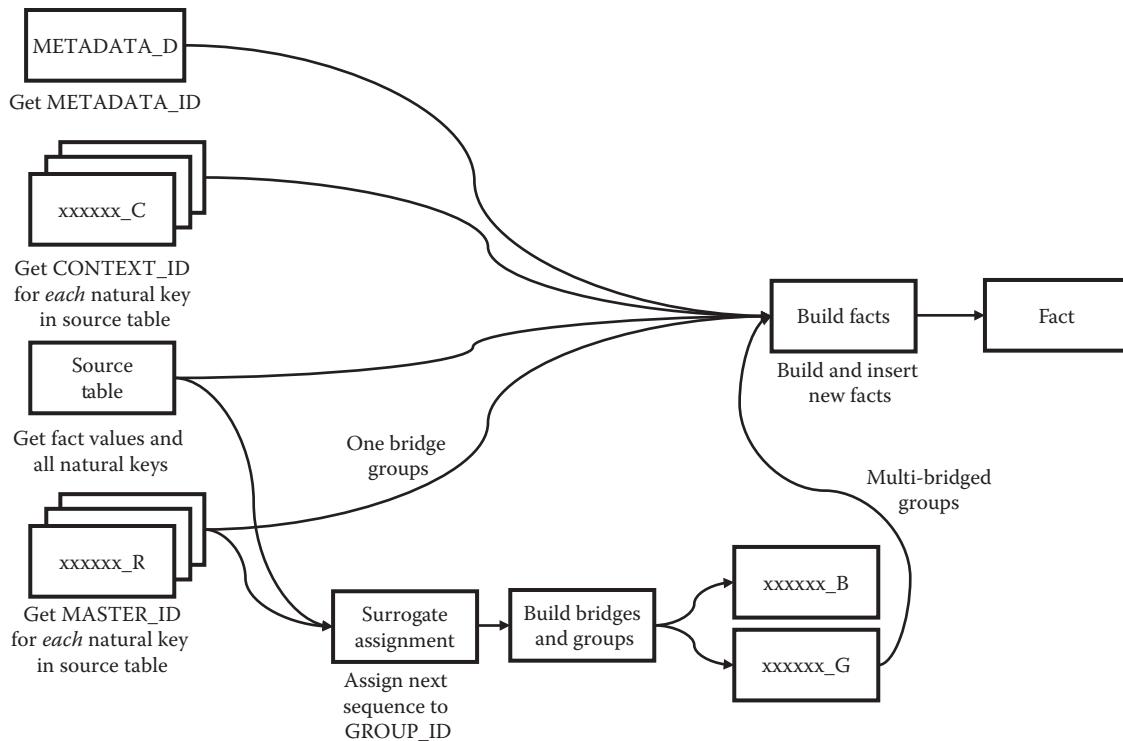


Figure 6.4 Brute-force fact load with multiple-bridge groups (Alpha).

These examples included all of the variations in coding structures needed to build the Alpha version of the warehouse:

- Definition loading from master vs. fact data sources
- Reference loading using one-part vs. multiple-part natural keys
- Hierarchy loading for self-referencing and natural key-driven dependencies
- Group loading for single-bridge vs. multiple-bridge groups
- Fact loading with variable group and filtering permutations

Using combinations of each of these coding constructs, the Alpha version of the warehouse can be loaded very quickly with a wide variety of data. The biggest limitation is usually gaining physical access to any source data of interest. Once access is available, these coding structures only take minutes to put together. You'll find that it isn't hard to load tens of millions of fact rows, involving dozens of fact types, within a few days. You'll learn as you are doing it. Typically, I completely wipe out the warehouse and reload it every few days during the early Alpha loading, because I've learned something that allows me to expand the dimensionality of the earliest facts I loaded using something I learned while loading later facts. Because this Alpha version is considered a throw-away, dynamic reloading shouldn't pose any problems up to a point. Once you hit 50–100 million facts in the Fact table, you probably should stabilize it all because people are writing testing and learning queries, and that process breaks down if you keep flushing the data.

Finalizing Alpha Structures

The nature of your throw-away code has left some holes in the data you've loaded that need to be cleaned-up before you can consider your loading complete. The main issue is that your Fact table has a lot of NULL foreign keys corresponding to dimensions you didn't use for those early facts. You're not particularly concerned here with dimensions that you never used, since you're unlikely to join to them in your testing or querying. Dimensions that were only used in some of the facts will give different results depending upon how you use them. It's best if you clean up some of these anomalies so you can query the warehouse consistently without regard to what data were loaded.

Two examples of this problem are in the Procedure and Geopolitics dimensions. You've loaded demographic address facts against Geography that didn't use the Procedure dimension, and you've loaded lab results against Procedure that didn't use Geopolitics. You'll get varying query results if you join to these dimensions with the Fact table in different scenarios, so you'd like to clean them up to avoid any problems. In all of these cases, the reason we didn't use the dimension on any particular fact was that the dimension was not applicable to our dimensionalization of the facts. To solve this problem, every dimension has the System subdimension that includes the Not Applicable row with a Master ID equal to 1. You can simply update the Fact table to set any unused dimensional keys to that value:

```
UPDATE FACT F
  SET PROCEDURE_GROUP_ID = 1
 WHERE PROCEDURE_GROUP_ID IS NULL;

UPDATE FACT F
  SET GEOPOLITICS_GROUP_ID = 1
 WHERE GEOPOLITICS_GROUP_ID IS NULL;
```

This code can be defined for any dimension, but should always be defined for any dimension that has been used in a variable fashion. If a dimension hasn't yet been used at all, and there will be quite a few of these depending upon the range of data that have been selected for the Alpha version, I recommend *not* filling out these keys. The NULL values should remain in the Fact table to signal that the dimension has yet not been used.

There are many other aspects of your Alpha load that aren't complete when viewed from the overall design of the warehouse, but other anomalies don't affect your ability to define, load, test, and use the Alpha version. The Beta version will fill in those details by the time they will have any impact on your loading and testing.

V&V of Alpha Version

Verifying and validating (V&V) the Alpha version data involves writing queries that will accomplish two mutually supporting objectives:

1. Verify that all the data loaded correctly fit the design model of six-table dimension entries, and fully dimensionalized facts, and
2. Validate that all definitions and facts correctly represent the data that were received in sources.

Metadata-Fact Counts

One of the most important, and common, queries used to test the results of warehouse loads is the Metadata-Fact Count query, which joins the Fact table to the Metadata dimension to count the number of facts in the Fact table by Metadata definition:

```
SELECT META.SOURCE, META.EVENT, META.ENTITY, META.ATTRIBUTE, COUNT(*)
FROM WHSE.FACT F
INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID)
GROUP BY META.SOURCE, META.EVENT, META.ENTITY, META.ATTRIBUTE
ORDER BY META.SOURCE, META.EVENT, META.ENTITY, META.ATTRIBUTE;
```

Based on the coding examples above, your result might look something like this:

Source	Event	Entity	Attribute	Count(*)
EMR	Demographic	Patient	Address	1,154,223
EMR	Result	Lab test	Comment	186,323
EMR	Result	Lab test	Value	421,883

This result shows that the warehouse contains three types of facts, with a large majority being the patient address facts. From the counts of the lab result values versus comments, we can also infer that 235,560 of the sourced lab results were extracted with a NULL value in the Comment column since that's the only reason there would be fewer Comment facts than Value facts. This basic metadata-fact count query will become a central staple of your ETL test scripts.

Dimension–Subdimension Counts

Next, it's quite common to take a row count in each dimension, by subdimension as a way to see what's been loaded into the dimensions:

```
SELECT "Procedure" AS DIMENSION, D.SUBDIMENSION, COUNT(*)
  FROM PROCEDURE_D D
 GROUP BY "Procedure", D.SUBDIMENSION
UNION
SELECT "Geopolitics" AS DIMENSION, D.SUBDIMENSION, COUNT(*)
  FROM GEOPOLITICS_D D
 GROUP BY "Geopolitics", D.SUBDIMENSION;
```

Based on the coding examples above, your result might look something like this:

<i>Dimension</i>	<i>Subdimension</i>	<i>Count(*)</i>
Procedure	Lab panel	3,225
Procedure	Lab test	2,854
Procedure	Reference range	2,140
Procedure	System	6
Geopolitics	City	14,326
Geopolitics	Country	4
Geopolitics	State	115
Geopolitics	System	6

This result shows all of the dimension data that have been loaded into the Procedure and Geopolitics dimensions. The rest of the dimensions can be added to the query with the appropriate UNION clause to extend this result to include other dimensions. This dimension–subdimension count query will also serve as a central component of many of your warehouse test plans. As the list of subdimensions grows large, I sometimes execute a version of this query that omits the System subdimension from each dimension; but I only do this after my testing has instilled high confidence that the required six rows are always present.

Dimension–Fact Counts

You can also combine these two initial verification queries into a single combined Fact Count by Dimension–Subdimension:

```
SELECT "Procedure" AS DIMENSION, D.SUBDIMENSION, B.ROLE, META.SOURCE,
       META.EVENT, META ENTITY, META.ATTRIBUTE, COUNT(*)
  FROM WHSE.FACT F
 INNER JOIN METADATA_D META ON (META.MASTER_ID = F.METADATA_ID)
 INNER JOIN PROCEDURE_B B ON (B.GROUP_ID = F.PROCEDURE_GROUP_ID)
```

```
INNER JOIN PROCEDURE_D D ON (D.MASTER_ID = B.MASTER_ID)
GROUP BY "Procedure", D.SUBDIMENSION, B.ROLE,
        META.SOURCE, META.EVENT, META ENTITY, META.ATTRIBUTE
ORDER BY D.SUBDIMENSION, B.ROLE,
        META.SOURCE, META.EVENT, META ENTITY, META.ATTRIBUTE
```

Your result might look something like this:

<i>Dimension</i>	<i>Subdi-mension</i>	<i>Role</i>	<i>Source</i>	<i>Event</i>	<i>Entity</i>	<i>Attribute</i>	<i>Count(*)</i>
Procedure	Lab panel	Ordered	EMR	Result	Lab test	Comment	186,323
Procedure	Lab panel	Ordered	EMR	Result	Lab test	Value	421,883
Procedure	Lab test	Resulted	EMR	Result	Lab test	Comment	12,310
Procedure	Lab test	Resulted	EMR	Result	Lab test	Value	86,413
Procedure	Reference range	Resulted	EMR	Result	Lab test	Comment	174,013
Procedure	Reference range	Resulted	EMR	Result	Lab test	Value	335,470
Procedure	System	Procedure	EMR	Demo-graphic	Patient	Address	1,154,223

The grand total in this result will *not* match the previous metadata-fact count result. Some differences are the result of the increases in subdimension detail, such as the total 186,323 Comment facts now being split between Lab Test and Reference Range entries in the Procedure dimension. Additionally, since those facts included a two-bridge group to Procedure, this latter query results in a Cartesian join that double counts some facts against the Procedure dimension. As a result, this query reports the opportunity for detail splits or group-based Cartesian joins that must be considered in any test query that counts facts by dimension.

Recreating Sources

A central validation strategy in a data warehouse is to be able to faithfully reproduce all of the source data from the warehouse data after they have been loaded. If all source data can be reproduced, without any exceptions, the loading is considered validated. The query structure that accomplishes this can be standardized:

```
SELECT S.<<< source columns >>>,
       W.<<< warehouse columns >>>
FROM
  (
    <<< Source Table Query Here >>>
  ) S
```

```

FULL OUTER JOIN
(
    <<< Warehouse Query Here >>>
) W
ON( S.<<< source columns >>> = W.<<< warehouse columns >>> )
WHERE (S.<<< any source column >>>           IS NULL
      OR W.<<< any warehouse column >>>       IS NULL)
  
```

This query structure selects all source columns and all warehouse columns from the results of a FULL OUTER JOIN of two inner queries: a selection of all source columns from the source tables, and a selection of all warehouse columns that should have been loaded using that source data. Each query might include some filtering to account for what might have been distinctive load strategies, as well as formatting of columns that might have been involved in transformations during the load. The ON clause for the FULL OUTER JOIN tries to equate each source column to each warehouse column. The WHERE clause excludes matches by looking for a NULL value on either side of the OUTER JOIN. If the loads were valid, this query structure will not return any rows. Rows returned represent mismatches in the load and will typically be present as paired rows. Using the results created by this query, you'll backtrack to where something was mapped incorrectly during the loading queries, and you'll repeat the load with the corrected logic. The cycle repeats until this query produces no result because everything in the warehouse exactly matches the source.

The validation query for our lab result Comment facts from source rows that included Reference Range data would be:

```

SELECT S.MEDICAL_RECORD_NUMBER,
       S.RESULT_DATE,
       S.ATTENDING_PROVIDER_ID,
       S.ABNORMALCY_INDICATOR,
       S.ORDERED_PANEL_ID,
       S.LAB_TEST_ID,
       S.REFERENCE_RANGE,
       S.RESULT_COMMENT
     ,W.MEDICAL_RECORD_NUMBER,
       W.RESULT_DATE,
       W.ATTENDING_PROVIDER_ID,
       W.ABNORMALCY_INDICATOR,
       W.ORDERED_PANEL_ID,
       W.LAB_TEST_ID,
       W.REFERENCE_RANGE,
       W.RESULT_COMMENT
  FROM
  (
      <<< Source Table Query Here >>>
  ) S
  FULL OUTER JOIN
  
```

```

(
    <<< Warehouse Query Here >>>
) W
ON (
    S.MEDICAL_RECORD_NUMBER = W.MEDICAL_RECORD_NUMBER
    AND S.RESULT_DATE = W.RESULT_DATE
    AND S.ATTENDING_PROVIDER_ID = W.ATTENDING_PROVIDER_ID
    AND S.ABNORMALCY_INDICATOR = W.ABNORMALCY_INDICATOR
    AND S.ORDERED_PANEL_ID = W.ORDERED_PANEL_ID
    AND S.LAB_TEST_ID = W.LAB_TEST_ID
    AND S.REFERENCE_RANGE = W.REFERENCE_RANGE
    AND S.RESULT_COMMENT = W.RESULT_COMMENT
)
WHERE (S.MEDICAL_RECORD_NUMBER IS NULL
    OR W.MEDICAL_RECORD_NUMBER IS NULL);

```

The source table inner query in the above code would be:

```

SELECT MEDICAL_RECORD_NUMBER,
       RESULT_DATE,
       ATTENDING_PROVIDER_ID,
       ABNORMALCY_INDICATOR,
       ORDERED_PANEL_ID,
       LAB_TEST_ID,
       CONCAT(REFERENCE_LOW, "-", S.REFERENCE_HIGH) AS REFERENCE_RANGE,
       RESULT_COMMENT
  FROM SOURCE.LAB_RESULT
 WHERE (REFERENCE_LOW IS NOT NULL
       OR REFERENCE_HIGH IS NOT NULL)
   AND RESULT_COMMENT IS NOT NULL;

```

The warehouse inner query would be:

```

SELECT SR.CONTEXT_KEY_01      AS MEDICAL_RECORD_NUMBER,
       CALR.CONTEXT_KEY_01     AS RESULT_DATE,
       CR.CONTEXT_KEY_01       AS ATTENDING_PROVIDER_ID,
       QR.CONTEXT_KEY_01       AS ABNORMALCY_INDICATOR,
       P1R.CONTEXT_KEY_01      AS ORDERED_PANEL_ID,
       P2R.CONTEXT_KEY_01      AS LAB_TEST_ID,
       P2R.CONTEXT_KEY_02      AS REFERENCE_RANGE,
       F.VALUE                 AS RESULT_COMMENT
  FROM FACT F
 INNER JOIN METADATA_D META ON (META.SOURCE      = "EMR"
                                 AND META.EVENT      = "Result"
                                 AND META ENTITY    = "Lab Test"
                                 AND META.ATTRIBUTE = "Comment")
 INNER JOIN SUBJECT_C SC   ON (SC.CONTEXT_NAME = "MRN")
 INNER JOIN SUBJECT_R SR   ON (SC.CONTEXT_ID   = SR.CONTEXT_ID
                                 AND SR.MASTER_ID  = F.SUBJECT_GROUP_ID)
 INNER JOIN CALENDAR_C CALC ON (CALC.CONTEXT_NAME = "Day")
 INNER JOIN CALENDAR_R CALR ON (CALC.CONTEXT_ID = CALR.CONTEXT_ID
                                 AND CALR.MASTER_ID = F.CALENDAR_GROUP_ID)

```

```

INNER JOIN CAREGIVER_C CC      ON (CC.CONTEXT_NAME = "Provider")
INNER JOIN CAREGIVER_R CR      ON (CC.CONTEXT_ID = GR.CONTEXT_ID
                                  AND CR.MASTER_ID = F.CAREGIVER_GROUP_ID)
INNER JOIN QUALITY_C QC        ON (QC.CONTEXT_NAME = "Abnormalcy")
INNER JOIN QUALITY_R QR        ON (QC.CONTEXT_ID = QR.CONTEXT_ID
                                  AND QR.MASTER_ID = F.QUALITY_GROUP_ID)
INNER JOIN PROCEDURE_B P1B     ON (P1B.GROUP_ID = F.PROCEDURE_GROUP_ID)
INNER JOIN PROCEDURE_C P1C     ON (P1C.CONTEXT_NAME = "Lab Panel")
INNER JOIN PROCEDURE_R P1R     ON (P1C.CONTEXT_ID = P1R.CONTEXT_ID
                                  AND P1R.MASTER_ID = P1B.MASTER_ID)
INNER JOIN PROCEDURE_B P2B     ON (P2B.GROUP_ID = F.PROCEDURE_GROUP_ID)
INNER JOIN PROCEDURE_C P2C     ON (P2C.CONTEXT_NAME = "Lab Ref Range")
INNER JOIN PROCEDURE_R P2R     ON (P2C.CONTEXT_ID = P2R.CONTEXT_ID
                                  AND P2R.MASTER_ID = P2B.MASTER_ID)

```

Conducting these source-recreating queries can be tedious, but they are worth doing in order to give users confidence in the load processes, and the analysis that went into their design. That confidence will be essential as you move into the Beta stage of development, where much of your analysis will become abstracted into the Metadata dimension.

BETA VERSION



Building the Alpha version of the warehouse allowed us to quickly have a functioning dimensional warehouse with some basic data available. The Beta version of the warehouse builds on the Alpha version by adding most of the required functionality and a lot more data. It provides a full functioned, but not production-ready, warehouse that can be loaded and queried in the broadest sense. The emphasis of the Beta version is to get the fullest possible warehouse up and running in order to facilitate longer-term planning, user and support training, and operational environment setup. It's a show-and-tell tool for the team, not a self-service tool for users. The Beta version can usually be ready about 2–3 months after the Alpha version is complete.

Chapter 7 provides details about completing the warehouse design beyond those capabilities that were required for the Alpha version. It completes some of the data controls not needed in the Alpha version and guides the initial development of the metadata that allows the Beta version to be developed generically. Chapter 8 then describes how to analyze, define, and extract source data for loading using the generic metadata concepts. The generalized ETL workflow architecture is then described in Chapter 8. Chapters 9 through 11 describe the specific logic required to load reference, definition, and fact data, respectively. Chapter 12 provides strategies for verifying and validating the data loaded into the Beta version of the warehouse.

Chapter 7

Completing the Design

To move into the Beta version, we need to back up and fill in some of the components of this general warehouse design that we were able to skip while building the Alpha version. We want the entire design in place for the Beta version so it will remain stable right through to production. To accomplish this, we need to add more capability to some of our dimensions beyond what the Alpha version required as well as establish a capability to initialize the warehouse in such a way that we'll be able to quickly reinitialize all data structures repeatedly during Beta development and testing. As we move toward a more generalized ETL architecture, we need to explore how capturing specifications in the metadata dimension will determine how data are processed in the warehouse, and we need to understand what that metadata is and how it gets defined by the source data analyst. Specific components to be completed here include:

- Expanding the domain of the Unit of Measure (UOM) dimension to include value-specific entries that allow for more control over querying common values that might appear in the Fact table but that can be more efficiently queried as details of the UOM.
- Fully implementing the Metadata dimension beyond the simple fact-identifying role that the dimension served in the Alpha version. The dimension will now contain a row for every distinct source-to-target value mapping from any of the data sources to any target in the warehouse, enabling the Beta version to construct complete source-independent generic ETL data-handling processes.
- Expanding the range of dimensions implemented in the model to include those that support the internal control system for the warehouse, most of which were likely to remain unused in the Alpha version because the hard-coded approach to getting that version built omitted most data controls.

While the first production release that we implement might not include all possible functionality, we want to ensure that we have the *capability* to implement all expected functionality. Functionality can always be added later in subsequent releases as long as we've laid the foundations—mostly in the implemented data structures—in the first release. The final scoping discussion for what gets included soon in Release 1.0 versus later in Release 1.1 or even later in Release 2.0 will vary in every organization. Implementing the rest of the design now will assure full forward compatibility into whatever time frames are chosen for the phasing of future implementations.

Unit of Measure

The UOM dimension is a vital component of the system of internal controls in the data warehouse. None of those control aspects of the dimension was needed for the Alpha version, so you might not have loaded anything into it in the Alpha version if you didn't need it to dimensionalize your early Alpha facts. Here in the Beta version, we want to add the use of this dimension to all of our source analysis and data mapping thought process, making it a common element of every ETL load.

The UOM dimension defines the *syntactic* elements that are used to place data facts into the warehouse, helping to ensure that stored facts are of the right type and range. The dimension is comprised of a six-level natural hierarchy of control data (Figure 7.1), comprised of scale, class, unit, measure, language, and value.

Because the dimension includes portions of the internal control mechanisms for the warehouse, sometimes it will be treated as any other dimension that users of the warehouse will source data into and query by. At other times, the dimension will be treated as an internal control that users might access but won't modify directly.

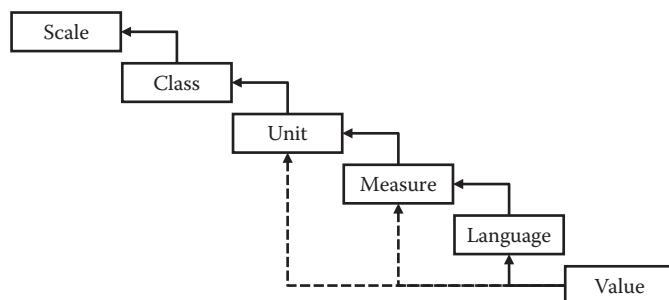


Figure 7.1 Unit-of-measure dimension conceptual hierarchy.

UOM Scale

The UOM scale classifies data with the warehouse's broadest classification. The list of scales is relatively small and is largely predetermined by convention. They are as follows:

- *Quantitative*: Numbers, counts, ratios, or amounts
- *Ordinal*: Ordered nonquantitative values (e.g., Likert scales)
- *Nominal*: Named unordered values (e.g., colors)
- *Narrative*: Free form or nondiscriminated text (e.g., pathology report, nursing note)

A scale designation is part of the warehouse's system of internal controls, so it is not a part of the reference identifier for a row in the dimension. The single most important control derived from the scale property is the ability to avoid trying to perform quantitative calculations on fact values that are not in a quantitative scale.

UOM Class

The UOM class indicates what is being measured by the associated facts. In theory, two units in the same class should be convertible to each other, although the warehouse might not contain the data necessary to carry out the conversion. The list of UOM classes is relatively small and is largely predetermined by convention. The base unit classes are as follows:

- *Length*: A standard measure of the distance between two points
- *Mass*: A standard measure of the amount of matter/energy of a physical object
- *Substance*: A standardized quantity of an element or compound with uniform composition
- *Temperature*: A standard measure of the average kinetic energy of the particles in a sample (a.k.a. how hot or cold something is)
- *Time*: A standard measure of the temporal direction in which events occur in sequence
- *Logic*: A standard measure of logical circumstance, including assertion or denial and presence or absence (often a flag or switch)
- *Quality*: A standard measure of some characteristic or property of interest, including the degree of quality (often an indicator or scale value)
- *Narrative*: A unit for which there is no predefined meaning, with the actual units usually embedded in the semantic content of the associated values
- *Quantity*: A standard measure of a physical or logical count to a variable grain of detail

Additional unit classes that can be derived from some of these base classes include the following:

- *Pressure*: A standard measure of the force applied to a given area
- *Volume*: A standard measure of the amount of space occupied by any substance
- *Frequency*: A standard measure of the number of repetitive actions in a particular time
- *Currency*: A standard measure of monetary value

UOM Unit

Specific units occur within the aforementioned classes and represent the actual thing being measured without any semantic context. Some examples are provided in Table 7.1. One particular unit that must be present in the dimension is the Unexpected Text unit (Table 7.2) that is used by the Fact pipe in Chapter 12 to suspend facts that have been defined as quantitative but that actually arrive from their sources as nonquantitative data. The Unexpected Text unit allows the

Table 7.1 Unit-of-Measure Scale–Class–Unit Examples

Scale	Class	Unit
Quantitative	Temperature	Centigrade (°C)
Quantitative	Pressure	Millimeters of mercury (mmHg)
Nominal	Logic	Yes/no flag
Quantitative	Length	Inches (in.)
Quantitative	Length	Centimeters (cm)
Quantitative	Quantity	Count
Ordinal	Quality	FLACC scale
Quantitative	Density	Units/liter (U/L)
Quantitative	Frequency	Units/minute (U/min)
Quantitative	Currency	U.S. dollars (\$)
Quantitative	Currency	Euros (€)

Table 7.2 In-Place Suspense Unit-of-Measure Entry

Scale	Class	Unit
Narrative	Narrative	Unexpected Text

erroneous facts to be loaded into the warehouse without corrupting the numeric data that some users might be using for aggregations or statistical analyses that would fail if a nonnumeric value were encountered.

UOM Measure

Measures begin to introduce *semantic* meaning (in this case, *healthcare* meaning) to the various units in the dimension. Some examples are shown in Table 7.3. Care should be exercised in establishing units and measures to assure that they will accurately reflect the semantic meaning that will be interpreted by the users of the warehouse. Typically, the combination of names should be distinct if the unit and measure names are combined (e.g., “Oral Temperature [°C]” vs. “Oral Temperature [°F]”). The scale and class are not included in most displays of UOM data, so unit–measure should be unique in the dimension.

UOM Language

Language allows narrative units, with or without measures, to be qualified according to the language of the narrative value (Table 7.4). The ETL Fact pipe does not make any effort to *enforce* the fact value actually being written in the specified language. The language in the UOM serves more to document

Table 7.3 Unit-of-Measure Scale–Class–Unit–Measure Examples

<i>Scale</i>	<i>Class</i>	<i>Unit</i>	<i>Measure</i>
Quantitative	Temperature	Centigrade (°C)	Oral temperature
Quantitative	Temperature	Fahrenheit (°F)	Oral temperature
Quantitative	Temperature	Fahrenheit (°F)	Tympanic temperature
Quantitative	Pressure	Millimeters of mercury (mmHg)	Systolic blood pressure
Quantitative	Pressure	Millimeters of mercury (mmHg)	Diastolic blood pressure
Nominal	Logic	Yes/no flag	Deceased
Quantitative	Frequency	Units/minute (U/min)	Breaths per minute
Quantitative	Frequency	Units/minute (U/min)	Beats per minute
Quantitative	Quantity	Count	Refills
Ordinal	Quality	FLACC scale	Pain level
Ordinal	Quality	Wong–Baker scale	Pain level
Quantitative	Currency	U.S. dollars (\$)	Revenue

Table 7.4 Unit-of-Measure Scale–Class–Unit–Measure–Language Examples

<i>Scale</i>	<i>Class</i>	<i>Unit</i>	<i>Measure</i>	<i>Language</i>
Narrative	Narrative	Text		
Narrative	Narrative	Text	Comment	
Narrative	Narrative	Text		English
Narrative	Narrative	Text	Comment	English

the *expectation* of the value’s language, an expectation that can be checked postload for consistency. Initially, the language for a value will be identified by your source data analyst so it can be set by the ETL during loading. It isn’t unreasonable for a narrative text in any particular language to contain terms or phrases in other languages, so this language indication is taken to be the *primary* language of the narrative. Here in the Beta version, you’ll rely most heavily on ASCII character set discrimination to estimate the primary language of a text when the analyst is unable to determine a primary language for a source in advance. In the Gamma version, you’ll supplement that logic through the language encoding available in many of the publically available semantic ontologies.

Language is playing an increasingly important role in informatics and analytics as the sources of data become more global and less formal. Global health data exchange increases the likelihood that certain textual facts will be received in different languages from biomedical professionals around the world. Textual facts received from less formal sources (e.g., personal health exchanges, Twitter) are increasingly likely to be in different languages based on the nationality of the creators of that data. As semantic ontologies are added to the warehouse design in the Gamma version, we’ll be able to use their sensitivity to language differences to identify language differences in our facts in very robust ways, making the UOM language feature a useful tool as our data become more diverse and mature.

UOM Value

UOM values provide a mechanism for identifying the specific values that a fact might take on and is ideal for providing fast query access to specific values. Since a lot of what it means to query a large-scale dimensional data warehouse involved looking at the values in the Fact table, it makes sense to add features to our warehouse design to make it easier and more efficient to see what’s in the Fact table. Querying by a dimension property value is always faster than by a fact value, and this will make a big difference when you have billions of rows in your Fact table. [Table 7.5](#) provides UOM scale–class–unit–measure–value examples.

Table 7.5 Unit-of-Measure Scale–Class–Unit–Measure–Value Examples

Scale	Class	Unit	Measure	Language	Value
Narrative	Logic	Yes/no flag		English	Yes
Narrative	Logic	Yes/no flag		English	No
Narrative	Logic	Yes/no flag	Deceased	English	Yes
Narrative	Logic	Yes/no flag	Deceased	English	No
Quantitative	Temperature	Fahrenheit (°F)	Oral Temperature		
Quantitative	Temperature	Fahrenheit (°F)	Oral Temperature		98.6
Quantitative	Currency	U.S. Dollars (\$)	Revenue		0.00
Quantitative	Currency	U.S. Dollars (\$)	Revenue		1.00
Narrative	Narrative	Text	Comment	English	NEGATIVE
Narrative	Narrative	Text	Comment	English	ERROR

Note in the examples that Value entries can be defined with or without measures and languages. In fact, only the first three entries are required by this design: A fact value must be able to be defined as being in a unit. Adding additional details related to the measure, language, and value to the dimension allows for a more specific linkage between the facts in that unit and the UOM dimension. More specific linkages generally provide better performance when executing queries against the data in those units.

To illustrate the difference, let's take an overly simplified example: Count the number of times a "Yes" or "No" value shows up in the Fact table entry that has been defined as being in the "Yes/No Flag" unit. A simple query can join the Fact table to the UOM dimension in order to isolate the correct facts and then count those facts grouped by the value in the Fact table.

```
SELECT F.VALUE, COUNT(*)
  FROM FACT F
 INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
 INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                        AND D.UNIT = 'Yes/No Flag' )
 GROUP BY F.VALUE;
```

The result of this query might look something like the following:

VALUE	COUNT (*)
Maybe	1,678
No	763,201
Other	8
Yes	352,298

Alternatively, if we've established the two desired value rows of interest in the UOM dimension, the same query can be built using the value in the UOM definition rather than the value in the Fact table, thus eliminating the need for the query to actually obtain values from the Fact table.

```
SELECT D.VALUE, COUNT(*)
  FROM FACT F
 INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
 INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                        AND D.UNIT = 'Yes/No Flag')
 GROUP BY D.VALUE;
```

The result of this query might look something like the following:

VALUE	COUNT(*)
	1,686
No	763,201
Yes	352,298

Note the differences in the results in the two versions of the aforementioned query. The first query provided accurate counts for other values in the Fact table, while the second result set simply showed that count as associated with a NULL entry. The difference is because we only speculated that the two value rows for "Yes" and "No" were added to the UOM dimension to support our requirement. The 1686 rows containing with "Maybe" or "Other" were still defined as being in the unit we're interested in, but no value rows were provided. Those facts are pointing to the actual unit subdimension row in the dimension, and the value column of that definition row is NULL. If our intent was to actually only count the "Yes" and "No" entries, the differences in the result sets aren't significant. Our requirement would be met either way. If our intent was really to know the fact counts for all of the values in the unit, the second query fails to meet the requirement.

To broaden the second example to correctly give all four counts, we would typically want to add the "Maybe" and "Other" value rows to the UOM dimension. That change would cause the second example query to provide the same results as the first. Since there were only two additional values and only 1,686 facts would be affected by the change, that's probably what I'd recommend in this example. Suppose there were dozens or hundreds of other values, and we really didn't want to put them as values in the UOM dimension. If that were the case, we could either revert to the first query example and accept the performance consequences of scanning the Fact table or we could combine the two approaches in a single unioned query that would use the two value rows for "Yes" and "No" values, while looking to the Fact table for any other values.

```

SELECT D.VALUE, COUNT(*)
  FROM FACT F
 INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
 INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                           AND D.SUBDIMENSION = 'Value' AND D.UNIT = 'Yes/No Flag')
 GROUP BY D.VALUE
UNION
SELECT F.VALUE, COUNT(*)
  FROM FACT F
 INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
 INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                           AND D.SUMDIMENSION = 'Unit' AND D.UNIT = 'Yes/No Flag')
 GROUP BY F.VALUE;

```

The difference in performance among these variations of these queries will vary depending upon several factors, so there's no specific number I can provide for how much better performance one version gives over another. Generally, I find that database optimizers are pretty smart these days, so I expect the portion of the queries I build between facts and dimensions to take advantage of good indexing to avoid looking at the Fact table. I also expect that when the cardinality of the need to look at the Fact table value is low (e.g., only 1600 facts with only two distinct values), the same indexes will provide for fast access to the Fact table without needing to revert to a table scan to obtain my desired values. In case of other factors being equal, I typically expect the UOM value-based query to be at least an order of magnitude faster than a similar fact value-based version of the query.

To illustrate the power of the Value entries in the UOM dimension, imagine a more typical scenario where you want to calculate an aggregate total for some fact type or group of facts. Suppose you want to aggregate all of the financial charges in the Fact table. Rather than writing a simple query to sum the fact value over all facts, an operation that will often result in a full table scan of the Fact table, you can use the value rows in the UOM dimension as a means of creating a virtual PRODSUM of the desired values. For facts that are defined using UOM Value entries, the desired sum is the product of the value and the count of those values; something that is obtained much faster in the indexes than in the Fact table itself. For facts defined using UOM unit entries, the traditional SUM query is still needed. These two subqueries can then be aggregated as follows:

```

SELECT SUM(TOTAL_CHARGES)  FROM
(
    SELECT D.VALUE * COUNT(*) AS TOTAL_CHARGES
      FROM FACT F
 INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
 INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                           AND D.SUBDIMENSION = 'Value'
                           AND D.UNIT = 'Dollars' AND D.MEASURE = 'Charge')
 GROUP BY D.VALUE
)
```

```

    UNION
    SELECT SUM(F.VALUE) AS TOTAL_CHARGES
    FROM FACT F
    INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
    INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                           AND D.SUMDIMENSION = 'Unit'
                           AND D.UNIT = 'Dollars' AND D.MEASURE = 'Charge')
    GROUP BY F.VALUE
);

```

The level of improvement using this approach over the original straightforward SUM query will vary depending upon the number of distinct charge amounts present in the Fact table and the ratio of those values that have been defined in the UOM dimension as Value entries. It also depends upon the cardinality of these facts relative to all of the facts in the Fact table. If the data volume is high (i.e., hundreds of millions of facts) and the distinct values follow a reasonable Pareto distribution (e.g., 80% of facts use only 20% of values and that are be placed in the UOM dimension), the performance improvement of this approach will be considerable. I've seen many queries executed in 5–10 seconds this way that might have required 5–10 minutes as full table scans.

To take full advantage of the UOM dimension controls against facts, there's one more query you will want to always include near any aggregating query in these examples, and that is a query to check for invalid values. In this case, what if some of the charges arrived from the sources with nonnumeric values? A simple standard query can check for the following occurrences:

```

SELECT COUNT(*) AS COUNT_OF_ERRONEOUS_CHARGES
FROM FACT F
INNER JOIN UOM_B B1 ON (F.UOM_GROUP_ID = B1.GROUP_ID AND B1.ROLE = 'Value')
INNER JOIN UOM_D D1 ON (B1.MASTER_ID = D1.MASTER_ID AND D1.STATUS_INDICATOR = 'A'
                        AND D1.UNIT = 'Unexpected Text')
INNER JOIN UOM_B B2 ON (F.UOM_GROUP_ID = B2.GROUP_ID AND B2.ROLE = 'Expected')
INNER JOIN UOM_D D2 ON (B2.MASTER_ID = D2.MASTER_ID AND D2.STATUS_INDICATOR = 'A'
                        AND D2.UNIT = 'Dollars' AND D2.MEASURE = 'Charge');

```

This query takes advantage of the ETL control that will check quantitative data for valid numeric entries in Chapter 14. If sourced facts are invalid, they will be moved into the Unexpected Text UOM, and the value will be marked as having “expected” the Dollar Charge UOM. Because they are in this controlled UOM, the invalid entries will not interfere with the aggregating summation query. To the extent that the calculated count of bad values is small relative to the total number of entries being aggregated, you might ignore them for business purposes, but that is a user choice only. Always including this extra query gives your users the information they need to make a more informed choice in each case.

Finally, and only to illustrate the flexibility of this design, there are a two more changes we can make to the aforementioned example that might improve its performance: (1) there's no need to calculate an aggregate of zero-amount facts,

and (2) there's no need to do the multiplication when the charge amount is one. These two changes alter our query to the following:

```

SELECT SUM(TOTAL_CHARGES) FROM
(
    SELECT COUNT(*) AS TOTAL_CHARGES
        FROM FACT F
        INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
        INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                                AND D.SUBDIMENSION = 'Value' AND D.VALUE = '1'
                                AND D.UNIT = 'Dollars' AND D.MEASURE = 'Charge')
    UNION
    SELECT D.VALUE * COUNT(*) AS TOTAL_CHARGES
        FROM FACT F
        INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
        INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                                AND D.SUBDIMENSION = 'Value'
                                AND D.VALUE <> '1' AND D.VALUE <> '0'
                                AND D.UNIT = 'Dollars' AND D.MEASURE = 'Charge')
    GROUP BY D.VALUE
    UNION
    SELECT SUM(F.VALUE) AS TOTAL_CHARGES
        FROM FACT F
        INNER JOIN UOM_B B ON (F.UOM_GROUP_ID = B.GROUP_ID AND ROLE = 'Value')
        INNER JOIN UOM_D D ON (B.MASTER_ID = D.MASTER_ID AND STATUS_INDICATOR = 'A'
                                AND D.SUBDIMENSION = 'Unit' AND F.VALUE <> '0'
                                AND D.UNIT = 'Dollars' AND D.MEASURE = 'Charge')
    GROUP BY F.VALUE
);

```

Whether these two changes actually improve performance or not is a function of the indexing of the data and the cardinality of the values in each table and index, but my experience is that they typically do help since many financial systems process zero and one dollar transactions quite a bit. The good news is that altering these query strategies doesn't involve making any changes to the data in the warehouse. Because some of these options involve extra code in the related queries, I typically only focus on creating the code when initial simpler versions of these kinds of queries begin to run slower than desired. Also, many of these more complicated versions of these queries are typically hidden from user view inside query objects created at the semantic layer, making them fairly easy to add, alter, or remove as experience unfolds in the use of the data.

There will be many factors that you'll want to consider when determining whether or not you want to use the value feature of the UOM dimension in different circumstances. Its intent is to provide fast access to facts that might otherwise depend upon undesirable Fact table scans when queried, so those are the circumstances where it should be considered by the data analyst. You'll also want to consider the volatility of the data values and the number of possible values and their relative distributions. For now, the Beta version implements this construct in a fairly rigid manner: The Fact pipe will check for value rows when inserting facts so the appropriate Value entries are used when generating dimension bridges, but no retroactive updating of any of those relationships will be supported.

The Gamma version (in Chapter 14) will add the maintenance capability needed to seamlessly change value rows in the UOM dimension. Once implemented in a dynamic way, the use of the value feature of UOM will be more flexible, and decisions as to whether or not to use the feature for certain facts can be separated from the original modeling and mapping of source. Value entries will be added or subtracted by data analysts late in the process, usually based on actual observation of query performance against data of interest. You'll tend to add Value entries for slower performing data, resulting in performance improvements without needing to change data mappings.

Metadata Mappings

The metadata dimension is a cornerstone of the data warehouse, providing all of the source-to-target mappings necessary for the creation of generic ETL jobs that can process arriving source data. In the Alpha version, you used the metadata dimension only to define each discrete fact that you were loading into the Alpha version Fact table. Since there was no generic ETL code in the Alpha version, you didn't need to use the metadata dimension for anything else. Here in the Beta version, the metadata dimension becomes the driver of all ETL processing and a center for developing user queries.

Here in the Beta version, we want a full implementation of the metadata dimension. This will entail implementing data that fit the standard design pattern for a dimension, but that was very likely to be omitted as it is unnecessary in the Alpha version.

Metadata Contexts

There are two contexts into which metadata references are mapped: Bridged and Unbridged. Both place the same logical values in the first six context keys, with the Bridged context also using the next three context keys to provide bridge values for fact-to-dimension connections in dimensions that use the group and bridge constructs in the dimension design pattern ([Table 7.6](#) for parameter names; [Table 7.7](#) for their definitions).

Since the Metadata dimension is not really used for querying in the same way as any other dimension, there will be a tendency to relax some of the design pattern heuristics for this dimension. I recommend against making exceptions, regardless of how inconsequential they might appear to be. Using two contexts to differentiate metadata using, and not using, bridges is one requirement that some people don't bother with; but I recommend keeping with the full model at all times. If you take shortcuts with the design, someone over time will make a mistake because he or she is counting on compliance to the design pattern.

Table 7.6 Metadata Context Columns

<i>Context Column</i>	<i>Values</i>	
CONTEXT_NAME	Bridged	Unbridged
LABEL_01	Physical Dataset	Physical Dataset
LABEL_02	Logical Dataset	Logical Dataset
LABEL_03	Fact Break	Fact Break
LABEL_04	Source Column	Source Column
LABEL_05	Target Table	Target Table
LABEL_06	Target Column	Target Column
LABEL_07	Target Role	NULL
LABEL_08	Target Rank	NULL
LABEL_09	Target Weight	NULL
LABEL_10	NULL	NULL

Metadata Reference

An entry of metadata always maps a single value in a source dataset to a single spot in the warehouse. Therefore, each metadata reference entry must uniquely identify only one mapping. If a distinct source value will be placed in multiple places in the warehouse, then there will be a different metadata entry for every placement. If a warehouse spot is sourced from multiple sources, there will be a different metadata entry for each of those sources. Metadata puts source to target into one-to-one discrete relationships.

In the next chapter, the generic source intake jobs will produce a standardized source dataset that includes the first four context keys defined here; so when the time to develop generic ETL logic in the following chapter comes, the source datasets will be internalized to the ETL process by joining on these first four contexts keys. In this way, the ETL will *know* exactly how to map any source column to any Target Column without advanced knowledge and with all processing carried out by the same ETL code regardless of what source provided the data. The detailed parameters that will drive that generic processing are provided in the Metadata Definition table.

Metadata Definition

I like to describe the metadata definitions as the set of parameters that I've externalized from my ETL code over the years. If you can imagine having a library of hundreds or thousands of ETL jobs or flows that have been developed over the years for a warehouse using traditional development techniques, you

Table 7.7 Metadata Reference Columns

<i>Reference Column</i>	<i>Purpose/Definition</i>
CONTEXT_KEY_01 (Physical Dataset)	An arbitrary name assigned to the dataset being sourced
CONTEXT_KEY_02 (Logical Dataset)	An arbitrary name assigned to a specific subset of the dataset being sourced; often a record or transaction type, or a distinct state of data in the source dataset
CONTEXT_KEY_03 (Fact Break)	An arbitrary name assigned to a specific fact within the dataset that requires different dimensionalization than other facts in the dataset
CONTEXT_KEY_04 (Source Column)	The name of the sourced column, typically populated as a result of an SQL UNPIVOT operation; although the name can be manipulated concretely in the source data intake as needed for any special mapping requirements
CONTEXT_KEY_05 (Target Table)	The warehouse table against which the ETL will map the sourced value
CONTEXT_KEY_06 (Target Column)	The column of the warehouse table against which the ETL will map the sourced value
CONTEXT_KEY_07 (Target Role)	(Bridged only) The role that will be assigned in the Bridge entry that connects the fact to the target dimension
CONTEXT_KEY_08 (Target Rank)	(Bridged only) The rank that will be assigned in the Bridge entry that connects the fact to the target dimension
CONTEXT_KEY_09 (Target Weight)	(Bridged only) The weight that will be assigned in the Bridge entry that connects the fact to the target dimension

will probably also imagine that most of those jobs or flows are almost the same. One flow puts the data in the patient dimension, while another puts it in the diagnosis dimension. One fact goes in the Clinical Fact table, while another goes in the Finance Fact table. One dimension is managed as a slowly-changing dimension, while another one is not. Beyond these predictable differences, the vast majority of the ETL code is virtually the same across most of the jobs and flows that serve a data warehouse.

Now imagine taking all of those hard-coded parameters out of those flows and externalizing them into a table so those thousands of flows can be reduced to a few generic flows that simply use the external parameters rather than their old hard-coded values. The Metadata Definition table is that external table of parameters. Its columns are used to drive the processing and flow of the generic ETL in terms of identifying targets in the warehouse database, naming the facts that result from that targeting, identifying units of measure that facts will be tied to implicitly, knowing when newly arriving facts should supersede previously received facts, mapping code values that need to be translated upon arrival, and identifying hierarchic relationships among different definitions in the dimensions.

Targeting Metadata

A metadata definition row includes properties for identifying the warehouse targeting of a source value (Table 7.8). The highest level distinction drawn in the targeting metadata is the load type that identifies whether the targeting of the source value is for the purpose of a dimension load or a fact load, or both (known as a hybrid load). This column is useful whenever you want to execute only a partial ETL run because it helps you limit execution to one kind of source feed. Although the generic ETL described here is intended to be used to execute against all data at the same time, it can be helpful at times to load dimensions before facts. I don't do this often, but when data volumes grow very large—particularly during historical loading or system testing—it is sometimes helpful to load the dimensions entirely before running fact loads. The load-type property provides a control for making that choice based on the data definitions.

The rest of the targeting metadata properties are specific to the actual Target Column for the received source value. The target-type property determines which of the ETL processing “pipes” the data will be processed in. Targeting also involves knowing whether a property is considered required or optional and what default value might be substituted if a NULL value is received.

Table 7.8 Targeting Metadata Definition Columns

Definition Column	Purpose/Definition
LOAD_TYPE	<i>Dimension, Fact, or Hybrid.</i> This value doesn't affect the ETL execution internally but is sometimes used at the job control level to decide what data should be included in any particular ETL execution
TARGET_TYPE	<i>Reference, Hierarchy, Alias, Definition, or Fact.</i> This value is the primary control for moving data through the generic ETL logic
REQUIRED_FLAG	Y or N. An indication of whether or not the sourced value should be considered required. Missing and required values will be set to “~Missing~” if not defaulted
DEFAULT_VALUE	The value that will be substituted for any null source value received
TARGET_DIMENSION	The target dimension name being targeted by the source value
TARGET_SUBDIMENSION	The target subdimension into which the data are being mapped. This value is only used when new definition constructs are added, including if an orphan is generated to support a fact
TARGET_CONTEXT	The dimension context within which the reference composite of the natural keys will be interpreted
CHANGE_TYPE	<i>Type 1 or Type 2.</i> An indication as to whether the target definition column should be managed as a slowly-changing dimension value (Type 2) or not (Type 1)

Other targeting columns determine the final target dimension, table, and column into which the source value will be placed. Note that the target dimension and target subdimension are in the Definition table, while the Target Table and Target Column ended up in the Reference table because they were part of the unique identifier for the mapping. Finally, the targeting metadata identifies whether a dimension target property will be managed as a slowly-changing dimension value.

Property Metadata

In defining your discrete facts in the Alpha version, your metadata entries defined the value that was being placed in the value column of the Fact table. Knowing the metadata row for a fact is analogous to knowing which table and column you might query in a more traditional relational database. The four property metadata columns that you used in the Alpha version will continue to be used throughout the rest of your implementation. Those four specific values in the Definition table of the Metadata dimension allowed you to identify the source from which a fact was received (e.g., CPOE), the type of event that generated the property (e.g., New Order), the entity that was being acted upon by the event (e.g., medication), and the correct interpretation of the fact attribute (e.g., dosage) (Table 7.9).

These four properties continue to be used as we move to the Beta version in the same manner, except that there will be metadata entries for all source-to-target mappings, not just the fact value targets. The four columns will be required when defining metadata for a fact in the Fact table and will be optional otherwise. However, if full auditing is turned on in the ETL processing, even

Table 7.9 Property Metadata Definition Columns

<i>Definition Column</i>	<i>Purpose/Definition</i>
SOURCE	The defining source system or transmission responsible for originating the data and to which a user would turn with source-related questions or concerns (e.g., CPOE)
EVENT	The operational or clinical event or transaction that resulted in the creation or setting of this attribute value. For fact values, the timing of this event is dimensionalized in the Calendar and Clock dimensions (e.g., new order)
ENTITY	The conceptual entity that serves as the object or target of the event. For fact values, this entity is typically dimensionalized within one of the dimensions of the warehouse star. For dimension properties, this entity is typically the subdimension for the related property (e.g., medication)
ATTRIBUTE	The actual valued attribute being captured for storage in the Fact or Dimension table. For fact values, the attribute should not try to capture the units of measure for the associated Fact value (e.g., dosage)

dimension property metadata ends up being factual metadata for the generated audit facts, so eventually all metadata rows will populate these four columns. Generally, though, we usually speak about these four columns only in the context of identifying facts in the Fact table.

Implicit UOM Metadata

Many source datasets don't explicitly source a unit of measure (UOM) for most of their columns. The implicit UOM metadata columns will be used to connect facts to the UOM dimension in those circumstances where no explicit UOM has been sourced ([Table 7.10](#)). These values are ignored if there is an explicit UOM in the source data. If these metadata columns are NULL, then no implicit UOM is determined.

While these UOM metadata columns are optional, it isn't possible to load a fact without any UOM. If no explicit UOM is provided by the source, and these implicit UOM columns are NULL, the final assignment of a default UOM takes place within the ETL Fact pipe processing based simply on whether the fact value is a valid numeric or not.

Superseding Metadata

For facts only, previous facts along defined dimensions will be marked superseded when newer active facts are created. The superseding metadata columns determine the dimensionality within which the superseding of facts should be carried out ([Table 7.11](#)). They should be left NULL if no superseding is intended.

Of the features and functions of this general warehouse design, the superseding of facts is among the subset of features that can't be completely automated in a generic way. The details of the superseding logic will be covered in Chapter 12, but I'll tell you in advance that it won't be perfect. The metadata columns defined here are pretty good at automating 75%–90% of the superseding requirements that I've run into over the years, but there is always some distinct subset of business requirements that I haven't been able to completely generalize. Those cases can be

Table 7.10 Implicit Unit-of-Measure Metadata Definition Columns

<i>Definition Column</i>	<i>Purpose/Definition</i>
IMPLICIT_UNIT	The unit in which the value should be assigned
IMPLICIT_MEASURE	The measure to which the value should be assigned. If not NULL, it must exist within the unit
IMPLICIT_LANGUAGE	The language presumed to be in the text value. If not NULL, it must exist in the unit or unit-measure

Table 7.11 Superseding Metadata Definition Columns

<i>Definition Column</i>	<i>Purpose/Definition</i>
SUPERSEDING_ORIENTATION	Subject, Interaction, or Organization
SUPERSEDING_DIMENSION	The warehouse dimension, within the specified orientation, against which superseding logic will identify matching facts
SUPERSEDING_ROLE	The warehouse bridge role, within the specified dimension, against which superseding logic will identify matching facts
SUPERSEDING_RANK	The warehouse bridge rank, within the specified bridge role, against which superseding logic will identify matching facts
SUPERSEDING_PERIOD	Ever, Year, Quarter, Month, or Day

implemented through manual coding in the Fact pipe and will simply be a small component of the architecture that remains less generic than the rest.

Codeset Translation Metadata

All values being processed by the ETL are subject to possible translation through the warehouse codeset dimension. The codeset translation metadata (Table 7.12), if populated, determines which codeset is used for that translation. The translation replaces the sourced value with a defined value in the codeset if the value exists in that codeset. These columns should be `NULL` if no translation is intended.

Table 7.12 Codeset Translation Metadata Definition Columns

<i>Definition Column</i>	<i>Purpose/Definition</i>
CODESET_CONTEXT	The context in the Codeset dimension within which value translation will take place
CODESET_CONTEXT_KEY_01	An identification of the controlling system for the codeset
CODESET_CONTEXT_KEY_02	An identification of the actual codeset in the controlling system
CODESET_CONTEXT_KEY_03	The codeset value against which the source value is compared for decoding
CODESET_COLUMN	The name of the warehouse Codeset dimension column that should be used to translate the sourced value if a match is found. This is typically one of the various definition stub description columns: DESCRIPTION, FULL_DESCRIPTION, EXTENDED_DESCRIPTION, ABBREVIATION, or VERNACULAR_CODE. If any other column is used, it must be added to the translation code in the ETL logic

Table 7.13 Fiat Hierarchy Metadata Definition Columns

<i>Definition Column</i>	<i>Purpose/Definition</i>
FIAT_ROLE	Ancestor or Descendent
FIAT_CLASS	Hierarchy, DAG, or Network
FIAT_PERSPECTIVE	An arbitrary name for the perspective within which the relation should be interpreted. It represents a logical grouping of entries, often derived from the ontology or ontology axis from which the relation was sourced
FIAT_RELATION	The specific name of the descendent-to-ancestor edge (e.g., “member of”). The ancestor-to-descendent edge is presumed to be the inverse of this relation (e.g., “has member”)
FIAT_ORIGIN_FLAG	Y, N, or U
FIAT_DEPTH_FROM_PARENT	1, 2, 3...
FIAT_TERMINUS_FLAG	Y, N, or U

Fiat Hierarchy Metadata

Reference data for fiat hierarchy loads must specify values for the resulting hierarchic edge between the two references. The fiat hierarchy metadata columns are required when the target type is hierarchy, and they should be NULL otherwise (Table 7.13).

Note that fiat hierarchy metadata should always come in two set of rows, with one set of metadata for the ancestor references and one set of metadata for the descendent references. Each set will have different numbers of metadata rows depending upon differences in the number of natural key components used by the two references. Regardless of the number of rows, however, the fiat hierarchy data columns other than role will always have the same data values since these rows eventually pivot on these columns to finalize the single Hierarchy table entry.

Metadata Examples

While most of these metadata columns are actually quite simple, they aren't necessarily intuitively obvious with respect to their meaning, purpose, or implications. That learning will continue in the next chapter. Before moving on, though, there are some observations about the metadata that might make the next chapter a bit easier to assimilate because you'll know where some of these are going.

The first time you initialize the warehouse, you won't have any metadata because you haven't yet sourced any data. In the next chapter, you'll define metadata at the same time you implement the sourcing queries that actually extract the data. The only data you really have on the first pass are the data you loaded in the Alpha version, so we'll use that data here to illustrate the various elements of metadata that will be considered and designed for each source dataset.

First Alpha Source: Patient Master Data

The first dataset you loaded into the Alpha version was your patient master data, consisting of five columns of data: medical record number (MRN), first name, last name, date of birth, and gender code. The metadata for this source will be fairly simple because the data are a simple dimension load of your Subject dimension, with each source row representing one row in our person subdimension.

Always begin metadata definition by defining the elements of the Metadata Reference table. The metadata references for this source will be defined in the Unbridged context, so the first six context keys will need to be populated. The data are simple, so three of the values will be constants across all source rows generated for these data.

<i>Reference Column</i>	<i>Value</i>
CONTEXT_KEY_01 (Physical Dataset)	Patient
CONTEXT_KEY_02 (Logical Dataset)	~All~
CONTEXT_KEY_03 (Fact Break)	~All~

The choice of a physical dataset name is arbitrary as long as it is unique among all source datasets that will be loaded, so we'll simply use the value "Patient" for this example. Because the dataset is very simple, we don't need a logical dataset distinction or fact break distinction within these data, so we use the customary "~All~" value to indicate that the metadata being defined should apply to all entries coming from the source within the physical dataset. To complete the reference entries, each source-to-target combination required for the patient load must be defined as an additional row of metadata.

<i>CONTEXT_KEY_04 (Source Column)</i>	<i>CONTEXT_KEY_05 (Target Table)</i>	<i>CONTEXT_KEY_06 (Target Column)</i>	<i>MASTER_ID</i>
MEDICAL_RECORD_NUMBER	SUBJECT_R	CONTEXT_KEY_01	101
MEDICAL_RECORD_NUMBER	SUBJECT_D	VERNACULAR_CODE	102
FIRST_NAME	SUBJECT_D	FIRST_NAME	103
LAST_NAME	SUBJECT_D	LAST_NAME	104
DATE_OF_BIRTH	SUBJECT_D	DATE_OF_BIRTH	105
GENDER_CODE	SUBJECT_D	GENDER	106
FULL_NAME	SUBJECT_D	DESCRIPTION	107

Source columns that are destined for more than one place in the warehouse must be defined in the metadata multiple times, once for each target. In patient data, the MRN must be mapped to Context Key 01 to serve as the identifier for the patient and to the Vernacular Code in the Definition table since users of these data think of the MRN as a vernacular value that is readily used in discourse independent of its use as a data key.

Note, also, that this list of source columns defined in the metadata must include any columns being generated by the source extract query that might not actually be part of the original source data. In this example, the Full Name column didn't actually exist in the source. The code in the Alpha version created these data as a concatenation of the Last Name and First Name source columns. The sourcing query in the next chapter will be responsible for ensuring that these columns are generated during the extraction.

Other source columns are mapped one-to-one with their counterparts in the Subject Definition table. Note that the Gender Code source column is mapped to a more descriptive column called Gender. During master loads, it is customary to convert coded values to their appropriate descriptions. Since the Alpha version didn't decode that value, we'll start doing so automatically in this Beta version.

In building the definition metadata for these entries, four of the columns will carry the same values across the rows.

<i>Definition Column</i>	<i>Value</i>
LOAD_TYPE	Dimension
REQUIRED_FLAG	Y
TARGET_DIMENSION	Subject
TARGET_SUBDIMENSION	Person

We'll need three other definition columns populated as well.

<i>MASTER_ID</i>	<i>TARGET_TYPE</i>	<i>TARGET_CONTEXT</i>	<i>CHANGE_TYPE</i>
101	Reference	MRN	
102	Definition		1
103	Definition		2
104	Definition		2
105	Definition		1
106	Definition		2
107	Definition		2

The MRN (Master ID 101) will be sent through our generic Reference pipe in Chapter 10, while all of the other data will flow into our generic Definition pipe in Chapter 11. Using Target Context, we indicate that the Reference pipe will look up the MRN value in the Subject Context named MRN. Of the definition columns, all will be treated as slowly-changing data columns (Type 2) except for the Vernacular Code (102) and Date of Birth (105) columns, which will be treated as Type 1. The codeset translation requirement for Gender Code (106) needs the codeset columns of the Metadata Definition table to be populated.

MASTER_ID	CODESET_CONTEXT	CODESET_CONTEXT_KEY_01	CODESET_CONTEXT_KEY_02	CODESET_COLUMN
106	~Codeset~	EMR	Gender	DESCRIPTION

The source Gender code column will be used as a lookup in the Gender codeset that was loaded from the EMR. If found, the Description column of the codeset will substitute for the coded value received from the source.

With the metadata complete, it can now be used to describe the processing that will occur for the data.

There will be a source for Patient (Physical Dataset) data that will be used to populate the Subject (Target Dimension) dimension with Person (Target Subdimension) master dimension (Load Type) data. The patient MRN will be used as the one-part MRN key (Target Context) and will be loaded into the Vernacular Code in the resulting definition row. The definition row will also be populated with the patient's First Name, Last Name, Full Name, Gender, and Date of Birth. The Gender will be a translated description of the sourced Gender Code, if available. Versions of the names and gender will be maintained over time. All of this data are considered required and quality messages will be raised if any of it is missing in the source.

Second Alpha Source: Patient Address Facts

The second dataset you loaded into the Alpha version was your patient address data. We used that data to load master data for countries, states, and cities, and then you loaded fact data for patient addresses. Since the data in this source were put to multiple purposes in the Alpha version, you'll use multiple logical dataset definitions in your metadata to accomplish the same distinctions, leaving only two of the reference columns carrying the same values across all rows.

Reference Column	Value
CONTEXT_KEY_01 (Physical Dataset)	Address
CONTEXT_KEY_03 (Fact Break)	~All~

Also, since this source includes some facts, you'll have a mix of metadata in both the Bridged and Unbridged contexts. Since the Unbridged context is a subset of the Bridged context, you typically begin defining the metadata only for the reference columns common to both contexts.

<i>CONTEXT_KEY_02 (Logical Dataset)</i>	<i>CONTEXT_KEY_04 (Source Column)</i>	<i>CONTEXT_KEY_05 (Target Table)</i>	<i>CONTEXT_KEY_06 (Target Column)</i>	<i>MASTER_ID</i>
Country	COUNTRY_CODE	GEOPOLITICS_R	CONTEXT_KEY_01	201
Country	COUNTRY_NAME	GEOPOLITICS_D	DESCRIPTION	202
State	COUNTRY_CODE	GEOPOLITICS_R	CONTEXT_KEY_01	203
State	STATE_CODE	GEOPOLITICS_R	CONTEXT_KEY_02	204
State	STATE_NAME	GEOPOLITICS_D	DESCRIPTION	205
City	COUNTRY_CODE	GEOPOLITICS_R	CONTEXT_KEY_01	206
City	STATE_CODE	GEOPOLITICS_R	CONTEXT_KEY_02	207
City	CITY CODE	GEOPOLITICS_R	CONTEXT_KEY_03	208
City	CITY_NAME	GEOPOLITICS_D	DESCRIPTION	209
Address	MEDICAL_RECORD_NUMBER	SUBJECT_R	CONTEXT_KEY_01	210
Address	COUNTRY_CODE	GEOPOLITICS_R	CONTEXT_KEY_01	211
Address	STATE_CODE	GEOPOLITICS_R	CONTEXT_KEY_02	212
Address	CITY CODE	GEOPOLITICS_R	CONTEXT_KEY_03	213
Address	STREET_ADDRESS	FACT	VALUE	214

This single physical dataset is being used in four ways (i.e., as four logical datasets): to load countries, to load states, to load cities, and to load addresses. The source columns will only be sourced once (i.e., the logical dataset in the source data will be “~All~”), but they’ll join effectively to any of the four logical datasets that need them. Within these logical datasets, each load makes use of the code and name columns from the source. The mappings also treat the keys to the state and city as compound two- and three-part keys, respectively.

You finish up the reference metadata entries for these data by adding values for the remaining three columns in the Bridged context for only those metadata rows involving reference values for the facts to be loaded in the address logical dataset.

<i>CONTEXT_KEY_05 (Target Table)</i>	<i>CONTEXT_KEY_06 (Target Column)</i>	<i>CONTEXT_KEY_07 (Target Role)</i>	<i>CONTEXT_KEY_08 (Target Rank)</i>	<i>CONTEXT_KEY_09 (Target Weight)</i>	<i>MASTER_ID</i>
SUBJECT_R	CONTEXT_KEY_01	Subject	1	1.0	210
GEOPOLITICS_R	CONTEXT_KEY_01	Geopolitics	1	1.0	211
GEOPOLITICS_R	CONTEXT_KEY_02	Geopolitics	1	1.0	212
GEOPOLITICS_R	CONTEXT_KEY_03	Geopolitics	1	1.0	213

The bridge data provided in these entries are simple, taking advantage of the single-bridge groups that are generated by the ETL for every new definition in any dimension. It need not necessarily have been that way, and the next example will get more complicated. In this case, an address fact will connect to a single Subject row and a single Geopolitics row.

With 14 rows of metadata now defined in the Reference table, the rest of the parameters required by this generic ETL system will be encoded in the Metadata Definition table. In building the definition metadata for these entries, only two of the columns will carry the same values across the rows.

<i>Definition Column</i>	<i>Value</i>
LOAD_TYPE	Hybrid
REQUIRED_FLAG	Y

Alternatively, the first three logical datasets (e.g., country, state, city) could be assigned a load type of dimension, and the fourth logical dataset (e.g., Address) could be assigned a load type of fact. You'll need several other definition columns populated as well.

<i>MASTER_ID</i>	<i>TARGET_TYPE</i>	<i>TARGET_CONTEXT</i>	<i>TARGET_DIMENSION</i>	<i>TARGET_SUBDIMENSION</i>	<i>CHANGE_TYPE</i>
201	Reference	Country	Geopolitics	Country	
202	Definition		Geopolitics	Country	2
203	Reference	State	Geopolitics	State	
204	Reference	State	Geopolitics	State	
205	Definition		Geopolitics	State	2
206	Reference	City	Geopolitics	City	

(Continued)

<i>MASTER_ID</i>	<i>TARGET_TYPE</i>	<i>TARGET_CONTEXT</i>	<i>TARGET_DIMENSION</i>	<i>TARGET_SUBDIMENSION</i>	<i>CHANGE_TYPE</i>
207	Reference	City	Geopolitics	City	
208	Reference	City	Geopolitics	City	
209	Definition		Geopolitics	City	2
210	Reference	MRN	Subject	Person	
211	Reference	City	Geopolitics	City	
212	Reference	City	Geopolitics	City	
213	Reference	City	Geopolitics	City	
214	Fact				

You also need to define codeset translation parameters for the country and state natural key values in order to support variations in these values from the desired standard values.

<i>MASTER_ID</i>	<i>CODESET_CONTEXT</i>	<i>CODESET_CONTEXT_KEY_01</i>	<i>CODESET_CONTEXT_KEY_02</i>	<i>CODESET_COLUMN</i>
201	~Codeset~	EMR	Country	ABBREVIATION
203	~Codeset~	EMR	Country	ABBREVIATION
204	~Codeset~	EMR	State	ABBREVIATION
206	~Codeset~	EMR	Country	ABBREVIATION
207	~Codeset~	EMR	State	ABBREVIATION
211	~Codeset~	EMR	Country	ABBREVIATION
212	~Codeset~	EMR	State	ABBREVIATION

With the metadata complete, it can now be used to describe the processing that will occur for the data.

There will be a source for patient Address (Physical Dataset) data that will be used to populate the Fact table with Addresses (Logical Dataset) dimensionalized to the Person (Target Subdimension) in the Subject (Target Dimension) dimension and City (Target Subdimension) in the Geopolitics (Target Dimension). The same sourced data will be used to load Country (Logical Dataset), State (Logical Dataset),

and City (Logical Dataset) in the Geopolitics (Target Dimension) dimension. Country and State codes will be translated to their standard coded values where variation occurs, and the descriptions of countries, states, and cities will be versioned over time within the dimension.

If the aforementioned text seems to read like a specification, that's intentional. The actual specifications against which you develop your ETL might come in a variety of ways, often very informal based on your analysis or sampling of the data. However, once your metadata has been created, you should be able to produce a very specific statement of what that metadata will do when processed by the ETL. As you get used to the metadata and its structure in this warehouse design, you'll come to see that specification in your mind simply by reviewing the metadata values. Until then, I always recommend translating any metadata you've created into a simple English paragraph that users and others not as familiar with the warehouse design can read and understand. If they agree with your generated specification, then they can have confidence in the load design even if they remain unfamiliar with the workings of the metadata dimension.

Third Alpha Source: Basic Lab Results

The third dataset you loaded into the Alpha version was your basic lab result data. You used that data to load master data for lab panels, lab tests, and reference ranges; and then you loaded fact data for the lab results, including the clinical result and comment columns. In this third example, only the physical dataset value remains constant across all metadata rows.

<i>Reference Column</i>	<i>Value</i>
CONTEXT_KEY_01 (physical dataset)	Lab Results

These source data are more complicated than the two previous examples because there are different kinds of data elements in this source dataset that need to be dimensionalized differently. The logical dataset will be used to differentiate source rows that contain a reference range value from those that do not. The fact break will be used to keep the processing of the two result and comment facts in each source row separately.

CONTEXT_KEY_02 (Logical Dataset)	CONTEXT_KEY_03 (Fact Break)	CONTEXT_KEY_04 (Source Column)	CONTEXT_KEY_05 (Target Table)	CONTEXT_KEY_06 (Target Column)	MASTER_ID
~All~	~All~	MRN	SUBJECT_R	CONTEXT_KEY_01	301
~All~	~All~	ATTENDING_PROVIDER_ID	CARGIVER_R	CONTEXT_KEY_01	302
~All~	~All~	ORDERED_PANEL_ID	PROCEDURE_R	CONTEXT_KEY_01	303
NoRange	~All~	LAB_TEST_ID	PROCEDURE_R	CONTEXT_KEY_01	304
RefRange	~All~	LAB_TEST_ID	PROCEDURE_R	CONTEXT_KEY_01	305
RefRange	~All~	REFERENCE_RANGE	PROCEDURE_R	CONTEXT_KEY_02	306
~All~	~All~	RESULT_DATE	CALENDAR_R	CONTEXT_KEY_01	307
~All~	~All~	ABNORMLCY_INDICATOR	QUALITY_R	CONTEXT_KEY_01	308
~All~	Result	CLINICAL_UOM	UOM_R	CONTEXT_KEY_01	309
~All~	Result	CLINICAL_RESULT	FACT	VALUE	310
~All~	Comment	RESULT_COMMENT	FACT	VALUE	311

Note that most of the metadata need not use the logical dataset or fact break columns to differentiate anything, so they contain the conventional “~All~” value. Those rows will apply to all incoming data regardless of any other distinctions. When the source extraction job places “NoRange” in the logical dataset, the Lab Test ID will be used as a one-part key to identify the appropriate entry in the Procedure dimension. When the source extraction job places “RefRange” in the logical dataset, the Lab Test ID and Reference Range will be used as a two-part key to identify the appropriate entry in the Procedure dimension.

The source dataset includes both the Clinical Result and the Result Comment source columns, so the distinction being drawn by the fact break is not implemented in the source extraction job. Therefore, it will place “~All~” in the fact break. By placing “Result” in the fact break in the Clinical Result and Clinical UOM metadata, you’ll indicate that the UOM being sourced applies only to the result fact, and not the comment fact. We place

“Comment” in the fact break of the Result Comment row to make it distinct from the Clinical Result metadata in order to prevent the sourced UOM from being used for the comment.

Finally, the references are completed by providing the remaining three columns needed for fact dimensionalization.

CONTEXT_KEY_05 (Target Table)	CONTEXT_KEY_06 (Target Column)	CONTEXT_KEY_07 (Target Role)	CONTEXT_KEY_08 (Target Rank)	CONTEXT_KEY_09 (Target Weight)	MASTER_ID
SUBJECT_R	CONTEXT_KEY_01	Subject	1	1.0	301
CARGIVER_R	CONTEXT_KEY_01	Caregiver	1	1.0	302
PROCEDURE_R	CONTEXT_KEY_01	Ordered	1	0.0	303
PROCEDURE_R	CONTEXT_KEY_01	Resulted	1	1.0	304
PROCEDURE_R	CONTEXT_KEY_01	Resulted	1	1.0	305
PROCEDURE_R	CONTEXT_KEY_02	Resulted	1	1.0	306
CALENDAR_R	CONTEXT_KEY_01	Event	1	1.0	307
QUALITY_R	CONTEXT_KEY_01	Abnormalcy	1	1.0	308
UOM_R	CONTEXT_KEY_01	Value	1	1.0	309

In the cases of Subject and Caregiver, the role, rank, and weight are defined as the default single-bridge entries always available in any dimension. The Procedure entries represent a two-bridge group, with an Ordered role and a Resulted role. The Calendar entry includes the required Event role, and the UOM includes the required Value role. The role for the Abnormalcy indicator has been set as Abnormalcy. This is a single-bridge group; but there's no information available here as to what subdimension the final definition will be in, so it's unclear whether this fact should use the default single-bridge entry of Quality as the role. The analyst needs to check the design of the master load for the Abnormalcy indicators to see if the assigned subdimension would be descriptive enough to allow the Quality role to be used here. If the subdimension were to be Abnormalcy, then a role of Abnormalcy would be redundant here. When in doubt, I suggest using a more descriptive role.

Control Dimensions

Data State

The Data State dimension defines the system states in which relevant facts should be considered within the data warehouse. The vast majority of data is in an Active state. Some data are placed in a Superseded state when newer

Table 7.14 Core Data State Dimension Entries

<i>Data State</i>	<i>Definition</i>
Active	Facts are considered valid and available. The vast majority of the facts in the warehouse are in this state
Superseded	Facts are considered available but have been updated more recently by additional facts that make these facts less likely needed (e.g., final lab results supersede preliminary lab results)
Deleted	Facts are considered unavailable because they have been indicated as having been removed by the source application. The data remain in the warehouse for historical analysis and typically remain visible in any user-oriented universe
Inactive	Facts are considered unavailable because they have been processed in such a way within the warehouse to invalidate their use (e.g., old data for a patient record that have been merged into another patient record). These data remain in the warehouse for internal control and are typically not visible in any user-oriented universe
Quarantined	Facts are considered unavailable by fiat because one or more business rules preclude the data being made available in an active state until one or more data owners or users approve a change

versions of the same data are captured. A small subset of data might be in Inactive or Deleted state. Table 7.14 provides the core Data State dimension entries.

The Data State dimension is a critical component of the warehouse's system of internal controls. The dominant row in the dimension is the active row, and virtually all data facts enter the warehouse in this state. Facts that pass through a series of life-cycle stages in the clinical process (e.g., preliminary vs. final vs. corrected lab results) will move to the Superseded state as new facts further out the life cycle are loaded. Some facts will transition through to the Deleted state if the source intake process receives delete transactions in the interface stream.

Operation

The Operation dimension defines the system-level indicators, usually order numbers, that can be used to group facts related to the same clinical or business events. This dimension is populated so all of the facts related to multiple-fact events can be tied together, including relating orders to results. This dimension needs to be generic enough to tie together different kinds of events and facts, so it is designed as a simple generic cluster of general descriptors. Most facts can be tied together with only a few descriptors (e.g., placer order number, filler order number, line number) but up to 10 are provided for the more extreme cases.

Data Feed

The Datafeed dimension defines the actual source application feeds from which relevant facts and dimension properties were obtained. A top-tier entry exists for each source application system from which data are extracted, while the lower second-tier entries provide traceability to the specific extract execution that obtained the data.

Reinitializing the Warehouse

With the entire design pattern now included in the design, the data structures built in the Alpha version will need to be reinitialized. This involves a lot more than simply flushing the Alpha data and inserting new Beta data into the structures. We actually want to create a highly repeatable *process* for reinitializing the entire warehouse, drawing a heavy distinction between an empty warehouse and an *initialized* warehouse. Having the ability to easily and quickly reset the warehouse to its initial state will enable us to test and retest our emerging Beta version. Not having such a capability becomes an impediment to both testing and extensibility in the opening months of development. Investing in this step enables more effective iterative development, with frequent system builds, and allows a small development team to complete the Beta version much more effectively.

Empty Warehouse

During the development of the Alpha version, you often recreated the database in its empty state in order to facilitate continuous and iterative development and testing. After you've run your table and index CREATE scripts in your database management system, you have an empty warehouse. You also have an empty warehouse if you simply delete all of the data in all of the tables. In the Beta version of the warehouse, we want to take that notion a step further in order to standardize the data that need to be present in our "empty" data warehouse in order to facilitate ETL processing. Much of this initialization data weren't needed in the Alpha version, and what was needed could be hard coded for Alpha testing. Here in the Beta version, we want to institutionalize this warehouse initialization process. We'll refer to our warehouse as initialized when all user data have been deleted, but the configuration data described here are still present.

System Rows

To initially configure an empty data warehouse, the following necessary six System subdimension rows must be added to every dimension:

1. *Not Applicable*: Used to define facts for which the concepts represented by the dimension would not be applicable logically
2. *Unknown*: Used to define facts for which the dimensional concepts are required, and for which the ETL processes that load facts should have had the necessary data to provide the connection, but didn't
3. *Undetermined*: Used to define facts for which the concepts of the dimension apply, but the data source for the facts would not have had adequate access to the data needed to establish the necessary connections
4. *Ambiguous*: Used when the ETL process, or postload spider process, is unable to connect facts to the correct dimensional entry because the algorithm involved yields multiple candidates for connection
5. *Other*: Used exclusively as a parent node for connecting orphans to flat hierarchies where the orphan process is unable to determine correct parentage
6. *All*: Used to define facts that have been completely aggregated up a hierarchy perspective for the dimension

Every dimension gets these same six rows added, including the use of the same surrogate identifiers (e.g., 1, 2, 3, 4, and 5). This is an exception to the general mandate that every construct in the warehouse have its own distinct identifier, even across dimensions. While most of the code in the warehouse system can be written using standard dimension access logic even for system rows, it will often be expeditious to “hard code” certain system row access in order to simplify the logic (e.g., the most common being that we know the Not Applicable row always has an identifier of one, and the Unknown row always has an identifier of two). If you aren't comfortable with such a shortcut in your system, simply use the standard context-reference access to build a lookup. If so, you can generate an arbitrary next surrogate identifier for the system rows as you insert them. The convention of using standard identifiers will be followed here. The system rows require a context to be defined in each dimension's Context table.

CONTEXT_ID	CONTEXT_NAME	LABEL_01
0	System rows	System row

The six system rows are then inserted into the Reference and Definition tables for that context.

<i>CONTEXT_ID</i>	<i>MASTER_ID</i>	<i>System Row</i>
0	1	~Not Applicable~
0	2	~Unknown~
0	3	~Undetermined~
0	4	~Ambiguous~
0	5	~Other~
0	6	~All~

The standard single-bridge group must be defined in each dimension for all six rows. As with any new Definition rows, the self-referencing rows must also be added to the Hierarchy table.

<i>ANCESTOR_ID</i>	<i>DESCENDENT_ID</i>	<i>TYPE</i>	<i>PERSPECTIVE</i>	<i>ORIGIN</i>	<i>DFP</i>	<i>TERMINUS</i>
1	1	H	~Self~	Y	0	Y
2	2	H	~Self~	Y	0	Y
3	3	H	~Self~	Y	0	Y
4	4	H	~Self~	Y	0	Y
5	5	H	~Self~	Y	0	Y
6	6	H	~Self~	Y	0	Y

Once again, I treat the identifiers of these rows as fixed values: zero for the context and one through six for the definition rows. This is an obvious exception to my general rule that all identifiers in the warehouse should be meaningless surrogate integers. The exception is worth it. It's very helpful to look at the data in the tables and be able to instantly recognize references to system rows. Losing that capability in order to maintain some data purity simply isn't rational. If you are a purist, then go ahead and generate a new surrogate key for every system context and definition row but recognize that you'll need to query the tables each time you need to assign a system row value to a reference in the data.

Data States

The different data states required for implementation are also required during initialization. As with the system rows, the data states available are a system internal parameter that can be controlled using a fixed list of values. Even though

the user community can add new data states over time, the list will remain very small even in the long term. The initial states added to the Data State dimension must include the core expected values.

<i>Data State</i>
Active
Inactive
Deleted
Superseded
Quarantined

The list of data states can grow over time, but this list will serve fine for an initial implementation. As your data warehousing process matures, and your governance group gets more involved, you'll encounter situations where you want to define additional enterprise-specific data states in order to support your own unique or distinct processing requirements. The Quarantined data state actually started out that way several years ago as I encountered organizations that had distinctive requirements for being able to load isolated data into the warehouse. The requirement occurred often enough that I now include it in my standard list.

The standard data intake table definition in the next chapter includes a Target State column for sourcing, so a new data state can be added at any time. However, adding a data state to a source dataset will only result in that particular data being loaded into that particular state. Typically, a desire to create a new data state is also associated with new processing requirements for that data. If the processing is specific to that data state, the code for the processing would have to be added and tested within one or more ETL flows outlined in the next few chapters. I recommend against taking this step prior to the first production release because it introduces variability into the design prior to gaining a full understanding of, and experience with, all aspects of this generic model. Introducing a new data state should be rare and need not be done too early in your warehousing journey.

Units of Measure

UOM are typically considered as user data, but five particular units must be present in order for the ETL to function, so are considered initialization data ([Table 7.15](#)).

Configuration Data

In addition to the aforementioned system rows, data states, and default UOM that are standard in any implementation of the design described here, there is

Table 7.15 Required Initialization Unit-of-Measure Entries

<i>Subdimension</i>	<i>Scale</i>	<i>Class</i>	<i>Unit</i>	<i>Value</i>	<i>Unexpected</i>
Unit	Narrative	Narrative	Text		N
Unit	Quantitative	Numeric	Numeric		N
Unit	Ordinal	Time	Time		N
Value	Narrative	Logical	Factless	Factless	N
Unit	Narrative	Logical	Unexpected Text		Y

a broad category of other data that also needs to be loaded into the warehouse during initialization that is more discretionary and subject to the specific goals and scope of the implementation. In fact, any initialization of these data is highly speculative, as the values and scale of the data are likely to grow or expand over the life of the warehouse. For this reason, I describe these data as configuration data because it *configures* the warehouse to handle the types and circumstances of data desired by the warehouse owner. The initial configuration of the empty warehouse can be limited to only the configuration needed for the initial implementation.

Context Entries

Across all of the dimensions, configuration of an initial data warehouse requires that all known or expected contexts be defined and loaded.

Chances are that you will not know all of your needed contexts when you first initialize your warehouse database. You'll discover new source system identifiers, and thus a need for new dimensional contexts, continuously as you obtain and analyze new source datasets. As time passes, you'll find that you discover new contexts far less frequently as the range of data you've identified grows. In my experience, your warehouse will eventually contain 150–200 different contexts across the 26 dimensions of the warehouse.

Most contexts occur in only one dimension (e.g., MRN in Subject), while others occur in several (e.g., ICD-9 in diagnosis, pathology, procedure, and environment). A few contexts, mostly externally obtained, will exist in virtually every dimension (e.g., SNOMED codes) because of the diversity of data represented by the external source.

No two implementations of this warehouse design will have matching contexts, but collectively the same scale of contexts will be common across most implementations. When a new context is identified, load it into the necessary Context tables as early as possible. The Context Name in that table is used in the source extraction job and the mapping metadata, so having it available as early as possible helps keep the development process moving smoothly.

Default Dimensionality

There are dimensions in the data warehouse design that will typically always be relevant to some of the facts being loaded, but that will not necessarily be sourced as part of the data intake process. If a fact is loaded without any dimensional reference to the Organization dimension, the assignment to that dimension will be automatically defaulted by the ETL code to the Enterprise ID in the standard intake dataset definition. It is important that you ensure that those Enterprise IDs are loaded into the Organization dimension in order to assure that this default connection works correctly. Likewise, if a fact loads without a reference to the Calendar dimension, it will be assigned by default to the date in the Source Timestamp associated with the fact being created. As a result, there will never be a fact without a date or an organization.

For all of the other dimensions, by default, if data are loaded into a Fact table with one of its dimension foreign keys NULL, the dimension reference will automatically be changed to the Not Applicable row in the System subdimension. In some cases, that might not be desirable because, while not sourced, the dimension really is *not* Not Applicable. You might eventually want to substitute a different dimension reference for some or all of your facts, although this generic design doesn't yet automate the logic or processing of any changes. For now, if you want this generic ETL design to handle different default dimensionality for you, you'll need to either source an additional artificial column to represent the desired dimensional connection (recommended) or hard code the fault logic in the ETL Fact pipe (not recommended).

Since each case will be somewhat different, what I recommend in advance is that you and your team spend some time considering the types of default dimensional entries you might want to use in some of these cases and then ensure that those rows are actually loaded into the appropriate dimensions so source data analysts can go ahead and included them in their source extraction without worrying about having to do a separate dimensional load for their desired default values. If the dimensions contain the desired values, using them in source jobs as artificial columns poses no risk to the design or the project schedule.

The default dimension entries can be any concept your team identifies as possibly relevant to the various sources and facts you intend to load. [Table 7.16](#) includes some examples of common default concepts by dimension that I've often seen used. While default logic can improve the completeness and usefulness of facts, it need not be implemented early. It's also possible to write fairly simply spider update queries to go into the Fact table and change Not Applicable references to one of the desired defaults. For this reason, I suggest not devoting too much time to this issue during Beta analysis. By the time you are implementing your Gamma version, you have a much more informed sense of where and when this default logic is really needed.

Table 7.16 Common Default Dimension Entries

<i>Dimension</i>	<i>Subdimension</i>	<i>Entry</i>
System	Species	Homo sapiens
Structure	Anatomy	Organism
Structure	Anatomy	Extended organism
Environment	Setting	Hospital
Study	Epistemology	Provision of care

Metadata Loading

When reinitializing the data warehouse, the metadata dimension will be among the structures that must be loaded. On the first pass through this chapter, you don't have any metadata yet. The act of sourcing data in the next chapter will result in defining metadata. As you cycle your development activities through iterations, any metadata you've defined and loaded will be reinitialized along with the rest of the configuration data in the warehouse. The first time through is simply a special case where you'll reinitialize the Bridged and Unbridged metadata contexts but won't have any metadata reference or definition data yet (unless you initially define your metadata to match your previous Alpha version loads). In any event, you need to fully understand what the Metadata dimension is and how to use its various parameters, as you move to the next chapter and start actually analyzing and mapping your source data.

Chapter 8

Data Sourcing

Sourcing data into the warehouse is the critical touchpoint between the vast array of systems and data structures that you want in your warehouse and the generic design-patterned structure of the warehouse you are implementing. The secret to getting to production in less than a year is getting this step right. You want to source your data so they look exactly the same, regardless of where they came from, so they can be processed by the single generic ETL workflows that you'll see in the next chapter. Since all sourced data will look the same, work to source more data can be carried out in parallel to the development of subsequent ETL jobs. Sourcing one dataset is sufficient to enable further development. Breaking the dependency between sourcing and loading is the reason why the warehouse implementation can be accomplished so quickly. More data can be added at any time, and sequence dependency testing can be accomplished by using the reinitialization tools developed in the last chapter. It all comes down to removing any dependencies on the structure or semantics of the source system data.

Source Mapping Challenges

The most complex activity that needs to take place during the warehouse development project is the sourcing of data into the warehouse. None of the risks you'll face here is unique to this particular form of data warehouse design; but because everything else about the warehouse design is intended to be generic and source-independent, the risks are heavily *concentrated* in the analysis of the source data and the design of the metadata that will be used in the subsequent ETL processes. The process of mapping source data into the warehouse requires you to have a thorough understanding of what each data source contains and means (i.e., the *analysis* side), as well as a thorough understanding of where different kinds of data will need to be placed into the warehouse so it is available where and when needed to support user

requirements (i.e., the *design* side). The risks of missing something, or making an inappropriate decision, will abound, so I suggest placing a very heavy emphasis on quality control around these activities.

The team members who are assigned to conduct this analysis need strong analytical skills and preferably a lot of organizational experience with the data to be analyzed. There are certain aspects of data sourcing that can be treated as generic, but there's no getting around the fact that every organization implements its data differently and that even standards like HL7 are only a partial guide to what the data might be, what it might mean, and how it must be processed for sourcing.

Coverage and Seamlessness

The first challenge in sourcing data for the warehouse is the problem created because a new data source will start loading into the warehouse at some arbitrary point in time. That point in time represents the demarcation between data that are loaded retrospectively as history data and data that are loaded prospectively as ongoing new data. This cutover point might impact all of the data in the warehouse when the system is first placed into production, but it will also impact every new data source added to the warehouse in the future. If you have a new source that will begin loading prospectively as of first of July, then at some point at or around that time you will load all of your history data for that source up through 30th of June. It's important for your user community that the data from that source look the same before and after the arbitrary cutover. If it doesn't, users of the warehouse will have to know how to properly query and interpret the data in both configurations, and the resulting queries will be more difficult. In some scenarios, the differences could be extreme enough to prevent integration of that source data longitudinally.

Let's take as an example a situation that I've seen occur on every biomedical data warehouse project I've been involved in: encounter admission and discharge. Your analysis of prospective data might conclude that encounter admissions and discharges will be captured by monitoring HL7 message traffic from your electronic health record (EHR) system, resulting in three types of facts being loaded into the warehouse:

1. The arrival of any HL7 In-Patient Admission (A01) message will be treated as the prospective source of an in-patient encounter admission fact.
2. The arrival of any HL7 Ambulatory Admission (A04) message will be treated as the prospective source of an ambulatory encounter admission fact.
3. The arrival of any HL7 Discharge (A03) message will be treated as the prospective source of an encounter discharge fact.

In this scenario, each prospective encounter will end up with two of these three facts loaded into the fact table: one A01- or A04-based admission and one A03-based discharge. The retrospective encounters won't be available as HL7

messages because those messages are rarely archived for processing. Your analysis is likely to find an encounter table, somewhere in or near your EHR that can be used to source historical encounters. Because the source table has one row per encounter, your analysis could easily conclude that these encounters can be loaded as a single fact. This kind of single source row to single fact historical loading is what I call a *snapshot*. The single history fact takes a snapshot of everything available in the source row because the data are so easily available. At its simplest, your history fact for the encounter can dimensionalize to both the admission and discharge dates.

I've seen this scenario many times. It's easy and fast, but it doesn't work. It's true that every encounter has been captured in the warehouse, and every encounter is dimensionalized to both its discharge and admission dates; but a critical seam has appeared between the retrospective and prospective data. If a user wants to calculate a length of stay (LOS) for every encounter, the query to do so would have to be different before and after that seam. Before the seam, the query would find the snapshot fact, obtain the two desired dates from the Calendar dimension, and calculate the LOS. After the seam, the query would obtain the admission and discharge facts, obtain the Calendar entry for each, and calculate the LOS. These two subqueries are functionally similar but syntactically very different. An expert user could design a single query to obtain the desired results (by querying and using all three fact types, knowing all of them won't actually occur together for a single encounter), but you don't want your users to have to become experts just to obtain something as simple and common as LOS.

The solution to this problem is to avoid creating retrospective-prospective seams in the data. I recommend always identifying and analyzing prospective data sources before identifying and analyzing their retrospective counterparts. Conversely, don't define historical sources unless you understand how a prospective source will keep the data up to date. I'm not trying to suggest that you have to actually *load* the prospective data first. Usually historical data are loaded first because they offer the background and volume that are needed to get a warehouse started. I want to ensure that when you get around to starting your prospective data loads you won't find you've made some major mistakes in mapping that would be very difficult to fix at that point.

To get around the seam problem for encounter admission and discharge history, you want to define your historical load so it creates the same data in the warehouse as would have been created if the data had been loaded prospectively. In this case, you should load a separate admission and discharge fact for each historical encounter even though the information for each fact comes from the same row of the historical data source. The created admission fact should differentiate in-patient and ambulatory encounters in order to load the correct historical admission fact. Done correctly, a user will be unable to detect any seam in the data between data loaded retrospectively and data loaded prospectively.

The idea of a seam in the data doesn't just occur at the fact level. Even getting the right number and type of facts loaded doesn't completely eliminate the risk. There can also be additional seams that show up in the dimensionalization of the facts themselves. It often occurs when the historical table being used to load retrospective data includes information that wouldn't have actually been available when one or more of the retrospective facts occurred.

In the case of your historical encounter table, there are probably many columns in that table containing data that accumulated throughout the encounter and wouldn't have been available at the time of admission. Obvious examples include the discharge Diagnostic Related Group (DRG) and the Discharge Disposition. Because the prospective A01 or A04 messages that will be used to record future admissions won't have these data, you must resist the temptation to dimensionalize your historical admission facts to these values just because it's available. Doing so would create a seam in the data. A user could take advantage of its presence, but only for retrospective facts before the data seam.

There are also some data seams that are more logical in nature. The number and types of facts, as well as what facts are dimensionalized to, are physical seams that result in variations of query requirements on either side of the seam. In the case of logical seams, the query needed to access the data might not need to change, but the way users interpret the data might need to vary. This kind of seam occurs when your historical source contains data that would have been available for the facts you are interested in loading, but not necessarily with the same values as would have been present when the actual events represented by those facts occurred.

An example is your probable desire to dimensionalize your historical admission facts to the nursing unit or bed to which the patient was admitted. Your prospective facts include these data, so you want your retrospective data to include it. The encounter table you are sourcing for your historical facts includes the needed columns, so you consider dimensionalizing your admission fact using those columns. The problem with this approach is that you don't actually know what nursing unit or bed each patient was admitted to. This is reinforced by the idea that you probably are dimensionalizing your historical discharge facts to those columns in your encounter table. It's important to remember that your source encounter table is a set of snapshots of your historical encounters, the vast majority of which have reached a discharged status. The nursing unit or bed in that table is most likely the patient's discharge location. Dimensionalizing your admission fact to that location would create the false assertion that they were admitted to the same location from which they eventually discharged, and you don't know that.

The answer to this situation is not to avoid dimensionalizing your historical admission facts to any facility. Completely omitting the dimensionalization would introduce a new physical seam that would actually be more disruptive than the logical seam you're trying to avoid. Instead, unless your historical data source actually contains an admission location, dimensionalize them to the

Undetermined row in the System subdimension of the Facility dimension. That entry matches the actual results of your analysis that the location into which the patient was admitted cannot be determined from the source data. In the Gamma version, you'll develop algorithms that attempt to resolve and improve these Undetermined entries. Perhaps by then you will have loaded data from your Bed Management system, and you'll be able to update these admission facts with the location of the earliest bed move obtained from that system. If not, then at least the Undetermined entry aligns the facts with what is actually known, thereby avoiding misleading or incorrect results in queries.

Beyond dimensionalization, the last place to be concerned with data seams is in the values placed in the fact tables. An example of this is a commonly present Principle Complaint column in the historical encounter table. In your prospective loading, you will probably include this value in your admission facts as it arrives in the A01 or A04 admission messages. You probably won't store it in your discharge facts at all because it typically isn't included in the A03 messages upon which you've decided to base your discharge facts. This creates a problem for your retrospective load because you have the column, but the row in the table is a snapshot on discharge. You can't tell from the table whether the value of that column was updated after the patient was admitted, potentially overlaying the original value that the prospective facts for that encounter would have had. Even though the record is a snapshot of the discharge, you shouldn't load the value in your historical discharge fact because it isn't being stored in the prospective discharge facts.

In this situation, I would probably end up storing the Principle Complaint as the value of the retrospective admission fact. Upon analysis, you'll find that the Principle Complaint is rarely updated during an encounter and in some cases is never updated. In that case, the column is an accurate picture of that complaint even though the table row is a snapshot of the discharged encounter. Even if there were many exceptions to that conclusion, the historical column is what is available, and sometimes you just have to go with what's available.

There will be columns in your retrospective data that are far more cumbersome and problematic than these examples. You'll need to explore the data you have access to and work with your stakeholders to map your data into the warehouse in the best ways possible. Avoiding *physical* seams is critical to enabling users to query data without advanced knowledge. Most users can handle a few *logical* seam exceptions. It's not that tough to understand that the Principle Complaint on older encounters might not be the original value. It's also not difficult to grasp that when querying to find a set of encounters with different admission and discharge locations, you have to remember to exclude any encounters where the admitting location was Undetermined. There will be many little exceptions like these, and they can be described in a simple user guide or as part of user training. The point is to maximize the coverage of your data as much as you can while also minimizing the number of data seams that a user might have to be aware of or deal with in queries.

Functional Normalization

Most of the data you'll source into your warehouse—particularly historical data—will come from databases that have been designed and implemented by some previous information technology project or initiative. We often presume that the data in those databases have undergone the analytical process of *normalization*. Since normalization includes the reduction of functional dependencies within and across the data, you can fall into the trap of believing that your data source is highly normalized even if it's not. Mistakes of this type can result in warehouse data that are almost impossible to be queried naturally because the data end up having a technical feel that only makes sense to information technology staff, or to users who have a strong familiarity with the original source datasets. Since you want users without any background knowledge to be able to comfortably query your data warehouse, functional normalization becomes an important aspect of analyzing data sources. A few examples can illustrate the issue:

- A data source for medication administrations contains factual data about medications actually administered to patients. However, it also contains facts about medications *not given* to patients for a variety of reasons (e.g., patient sleeping, wrong dosage, patient off unit).
- A data source for allergy assessments contains factual data about patient allergies to different allergens, including the severity of their reactions when exposed to those allergens. However, it also contains negative findings of *No Known Allergies (NKA)* and *No Known Drug Allergies (NKDA)* in which no allergies have been found.
- A data source for laboratory results contains factual data about clinical results of lab tests, typically a quantitative or qualitative value accompanied by some form of comment or remark. However, it also contains entries where a result could not be reported for a variety of reasons (e.g., contamination, sample insufficiency, inconclusiveness).

In each of these scenarios, negative consequences arise for querying the data if these functional differences are not recognized and handled appropriately in loading the data into the warehouse. The different functional conditions that result in the same or similar entries in your source datasets must be separated or normalized. Because you will conduct most of your source analysis by looking at the datasets themselves, it can be difficult to discern the functional distinctions that went into populating those datasets. If you have an opportunity to actually observe the functional differences in clinical practice, you can often directly see the nuances that create the functional differences you are looking for. Without direct access, you need to adopt a fairly conservative analytical perspective to ensure that you properly normalize the functional view of the data you are analyzing.

Table 8.1 Data Source Analysis 2 × 2 Framework

<i>Finding</i>	<i>Assertion</i>	<i>Correction</i>
Positive	Assertion of a positive finding	Correction of a previous positive finding
Negative	Assertion of a negative finding	Correction of a previous negative finding

I recommend that you always presume that a data source does more than its name implies. If it nominally addresses positive findings, look for negative findings as well. If it presents correct data, look to see if it also presents incorrect data. Many datasets will not contain these distinctions, and it will usually take you just a few minutes to confirm that fact. The risk of wasting a few minutes taking a conservative position toward a new source is much lower than the risk you face if you don't recognize the functional distinctions present in your data. The analytical conservatism I'm recommending results in a 2×2 analysis construct (Table 8.1). Look for your source data to potentially support both positive and negative findings, as well as the corrective reversal of any those findings.

If you haven't already discovered what to look for, it can take considerable effort to identify and understand these functional distinctions by looking only at your source datasets. Every situation can be different based on the design and usage of the system from which you are sourcing the data. In the case of medication administrations, your source data might have a status column that explicitly indicates which of your four possible scenarios is represented by each row. You can generate appropriate facts when you source the data based on that column. Alternatively, you might see a Reason Not Given column in the source dataset and will be able to infer that the medication wasn't given when that column isn't NULL. There might be a Correction Flag in the row to indicate that the functional transaction represented by the data was being removed, or undone. Your job as analyst is to ensure that the way you load the data into the warehouse has a look and feel that matches what was really happening when the data were created.

I've seen scenarios where a medication administration fact was recorded for a patient, followed a few minutes later by a correction of that fact (i.e., sometimes referred to as "erroring out" the previous fact), with a final transaction a few minutes later recording that the medication was not given because the patient was off the unit. I've also seen the reverse: a medication not given transaction because the patient is asleep, followed by a correction of that fact, and a final medication administration fact. You can imagine what must be happening on the nursing unit that drives these transactions in your systems. To save time, staff sometimes enter events into the EHR in anticipation of doing something only to find they can't actually perform the tasks. Those strings of transactions that assert–correct–deny or deny–correct–assert are actually very common in source datasets, not just in medication administrations, but throughout healthcare. The advent of bar-code technology at the point of care is reducing the frequency

of these transaction strings; but they aren't going away completely, particularly among historical datasets that will be the focus of much of your data analysis.

The result of this level of conservative analysis is that sometimes you'll end up sourcing a table of data that nominally contains one type of fact (e.g., medication administrations) into four different fact types in the data warehouse. Each can be included in user queries independently because they are defined as distinct facts. Some user training will be required on some of these facts. A user counting medication administrations will need to know to count the positive assertion facts and then subtract a count of the assertion corrections in order to get an accurate count of administrations made.

In the Gamma version, you will add the capability for the loading of the assertion correction fact to automatically change the data state of the original assertion fact, so it doesn't show up in queries. The reason I don't recommend trying to do that here in the Beta version is that data state controls are a data governance issue, and at this stage of your implementation you probably don't have a sufficiently mature data governance body to begin making the long-term decisions necessary to correctly implement these controls. Attempting to do so now is more likely to cause significant delays in your Beta version than to successfully implement the controls. If this becomes a pressing issue among your user community for some reason, I recommend temporarily adjusting the specific data of concern using an Alpha-version-style update logic. That extra logic can be discarded when the generic Gamma version controls are implemented.

The functional differences among the four medication administration facts were driven by the need to recognize real-world distinctions in clinical practice that sometimes aren't obvious in source datasets. In other circumstances, the distinctions might be a result of how software systems have evolved in support of changing clinical practice. Allergy assessment data sources often fall into this analytic category.

An allergy fact generally needs an allergen, a reaction, and sometimes a severity. Early EHRs provided a system function for recording these allergies in a patient's computerized chart. As clinical practice has evolved, allergy management has become more involved. Today's standard of care is that we record the results of every allergy assessment, and those results can be positive or negative. The positive allergy assessment is the counterpart to the earlier allergy transactions that your EHR systems were made to support. In many cases, and in my experience in *every* case, the evolution of the software capabilities of the EHRs hasn't kept pace with the evolution of your clinical practice regarding allergy management.

Judging from the source datasets I've analyzed in recent years, providers are entering the results of allergy assessments, both positive and negative, into EHR functionality that still expects only positive results. This has implications for both clinical practice and the sourcing of allergy data into your data warehouse. In computer systems like EHRs, it can be difficult to record a negative statement. In many cases, it forces the assertion being made in the fact to be more categorical.

If you add an entry into the allergen table in the EHR that reads “No Known Allergies (NKA),” you end up indicating that a patient has no allergies by actually asserting that she or he *is* allergic to that entry. No provider will misinterpret that information when it shows up on a screen in the EHR, but it can wreak havoc in a data warehouse implementation if you don’t recognize the issue and source the data to handle the situation.

The NKA problem actually gets worse. In most EHRs, having entered an allergy, the system next requires the provider to enter a reaction and sometimes a severity. The provider is now in a position of having to specify a clinical reaction for a patient who actually isn’t allergic to anything. Many systems have gotten around this problem by adding a “No Reaction” entry to the reaction list used by the system and training providers to use that entry as the reaction for an NKA allergy transaction. If severity is optional in the EHR, it is usually omitted by the provider. If required, a new zero severity score is typically added to the system.

In theory, recognizing these data distinctions should allow you to source the data into very functionally specific sourced facts that will accurately reflect the clinical picture that providers are entering for their patients’ allergy management. A few more conditions sometimes apply as additional nonallergy categories are added to the allergen list (e.g., “No Known Drug Allergies or NKDA”) or allergy types are introduced (e.g., drug allergy, food allergies, environmental allergy). As long as you recognize them as functional distinctions to be normalized, they don’t present any new or unique challenges. Positive allergies can be recorded using allergen, reaction, and severity. Negative (non)allergies can be recorded using only the categorical allergen.

In practice, this approach to normalizing allergy data doesn’t work; at least it hasn’t worked in any implementation I’ve seen. The reason is that the EHR system often doesn’t actually enforce the normalizing constraints upon which you’ve based analysis. You’ll see NKA assessments that also show an actual reaction (e.g., hives) and a nonzero severity. I don’t know exactly what those transactions mean, but I know I need to represent them in the warehouse since they exist in the data source. This means that, while you’ll keep the facts distinct in the warehouse, even the negative assertion facts will need to dimensionalize to the severity and reaction. Leave the interpretation of the data received to the users of the warehouse, and eventually to the governance body, if these issues need to be corrected. Your job is to faithfully reflect the source data, making sure that the distinctions you’re seeing can be seen in the warehouse. Having positive and negative allergy assessments as separate facts allows users to query one or the other without confusion. Users can decide how they want to interpret reactions and severities on negative assessments.

Another instance where you should not try to enforce some of what you learn about allergy data is in recognizing differences in allergy types. You’ll be tempted to try to ensure that the allergen on a sourced drug allergy is, in fact, a drug. Unfortunately, this is another area where the source system might not have enforced that constraint. Drug allergies to foods, food allergies to environmental

factors, or pet allergies to drugs are all too common in the actual data I've seen on projects in recent years. You'll also see allergy facts that represent allergies to what you can easily see are multiple allergens (e.g., "Bananas and Apples"). Is the patient allergic to bananas and apples separately, or only to the two when exposed to them together? Should the patient be included in a count of patients allergic to apples? Even though some selective transformation seems obvious upon inspection, you must resist the temptations to break these facts apart. The capabilities to support these transformations will be added to the Gamma version at a point where your governance body has matured sufficiently to make the decisions necessary to implement these types of transformations.

A long as you consistently record both the allergen and allergy type on each fact, it will be up to the users of the data to decide exactly how the data should be interpreted. By loading the data as they actually occur in the source, you also create an excellent opportunity to use the warehouse to diagnose procedural problems in the field that might remain hidden if you attempted to clean up the data prematurely. An excellent way to convince EHR vendors that they need to improve their data controls is to actually show them the bad data their systems allow users to enter. In the meantime, you will have isolated and normalized that data so it can be queried consistently, and interpreted correctly, in the data warehouse.

The normalization challenges of scenarios like the medication administration or allergy management facts are fairly routine compared to even more challenging situations in which the data in a source dataset require much more explicit analysis and interpretation in order to decide what they really mean in terms of facts to be sourced into the warehouse. A source for laboratory results often falls into this more complex category.

Lab results can be positive or negative assertions of activity, and they can be corrected over time, providing an opportunity to differentiate facts among these variations. The idea of a negative assertion, in the sense of your data analysis, includes situations where the lab result is somehow signaling that no real result was possible (e.g., contaminated sample). It does not include the situation where the results being reported were "Negative." Reporting a result of "Negative" (which turns out to be the most common fact value in each of the biomedical warehouses that I've been involved with, "Yellow" being second) is a positive assertion of a lab result for analytical purposes. Your analytical challenge is to establish how negative findings are being communicated by your lab result source. The older the source system, the more diverse I have found these mechanisms to be. Often, I find that the word "Error" is placed in either the Clinical Result column of the source or the beginning characters of the Result Comment column. Finding that word, or something functionally similar to it, in the data row can usually be confirmed as an indication that the lab result is a negative transaction. Lab results representing positive versus negative assertions can then be represented as separate distinct facts when you source the data into the warehouse.

Recognizing when a lab result source row represents a corrective assertion can be more difficult. Again, the older the lab system from which these data are sourced, the more diverse the representations will be. Many systems offer a status column that indicates whether a result entry is preliminary, final, or corrected. If so, then you can dimensionalize your two facts to the result status to indicate its temporal progression, or you can actually define separate facts to represent data in each of these three states. Either way, you'll also be able to indicate that the progression of preliminary, final, and corrected results should be treated as temporal. If you keep the status as a dimension of each fact, you'll be able to have the Beta version ETL automatically keep track of which fact is the most recent for a lab result. If you choose to make the distinction one of loading separate facts for each status, the tracking of the most recent lab result won't be implemented until you implement cross-fact superseding logic in the Gamma version.

Functionally normalizing your data is an important aspect of understanding your sources. While the choices you make in defining different types of facts, or in selecting exactly what dimensionality each type of fact requires, might leave a few nuances and variations in the data that users will need to be trained to handle. Many of those will be eliminated by processing that you'll add in the Gamma version, but the alternative of not normalizing these functional issues presents even greater risks. Unnormalized, the data in the warehouse will have a data processing feel to it, with facts looking more like system transactions than representations of real-world clinical activity. Users who don't understand the transactional nature of the data might misinterpret query results. Imagine seeing three medication administrations where there was only one genuine administration after a negative–correction–positive series of transactions, or none after a positive–correction–negative series. Users who make decisions using misleading query results quickly lose confidence in the warehouse itself. Imagine being told that the most common allergen is “NKA” or that the most common lab result is “Error.” It's possible to train users to correctly query unnormalized functional data, but with proper source analysis and fact definition, that training isn't necessary.

Qualities of Facts

One of the more difficult parts of designing a star schema data warehouse is deciding what to put in the fact tables. This might not seem intuitive since the main reason many organizations would give for building a data warehouse is the desire to store and query the facts contained in the fact tables. Outside of biomedicine, the choice of what to put in the fact table is clearer because most traditional data warehouses are intended to be used to aggregate quantitative data. If something isn't a quantitative value that a user would want to aggregate up the various hierarchies in the dimensions, it wouldn't need to be placed in a fact table. There are certainly exceptions in any warehouse outside of

biomedicine, but those exceptions are relatively rare, and the volume of data represented by those exceptions is typically small. Presenting the data as a series of analytical “cubes” that can be navigated and aggregated is a central warehousing purpose that makes the choice of what to store as fact values fairly predictable and stable.

The typical data profile in biomedicine is reversed from those nonbiomedicine scenarios. Most data in healthcare are textual, not quantitative. Even the data that are quantitative are not of a type that a user would typically want to aggregate on a regular basis. No user needs to calculate the average blood pressure of a cohort of patients in the way a sales manager might want to calculate an average order value for a group of customers. Biomedical data are simply not handled or used in the same way as traditional business data. Healthcare fact data are much more likely to be counted than aggregated. Data can be counted whether quantitative or qualitative, so the notion that quantitative data to be aggregated is your primary indication that something should be treated as a fact doesn’t always apply with the diversity of biomedical data. Some other criteria are needed for deciding upon what to put in your fact tables versus what you put in your dimensions.

The first criterion I recommend is to store something in a fact table only if it’s something a user would want to count. This is an admittedly weak recommendation because anything can be counted, but the distinction between something that can be counted versus something somebody might actually count can be helpful as a starting point. The second criterion I recommend is to avoid storing millions of facts that would only have a handful of possible values. At the lower end of my “handful” might be facts that can only have one to a dozen possible values. At the high end, a fact that can only take on a few hundred distinct values is suspect, although possible. I’m not trying to recommend a specific cutoff point; rather, I’m trying to set the stage for your analysis. My third recommended criterion is to avoid storing facts that are trying to describe other facts. Descriptive facts are best seen as providing qualities of other facts, and there’s a dimension of the warehouse design devoted to defining qualities: the Quality dimension.

Lab Result Facts

An example will illustrate my recommendations. You might be analyzing a source dataset of laboratory results. Many of the columns of the table represent concepts that you easily determine are dimensional data: patient, encounter, lab test, attending provider, lab technician, order date, result date, and unit of measure. These properties are readily recognizable as corresponding to dimension entries, so you know you’ll dimensionalize whatever facts you choose to store from this dataset using those dimensional values. There is also a group of data

columns that you decide might need to be stored as facts in your fact table because they seem to represent the actual content of the lab result:

- Clinical Result (e.g., a quantitative value like 1.2 or 210; or a qualitative value like Negative or Yellow)
- Abnormalcy (e.g., Normal, High, Low, Critical High, and Critical Low)
- Low Reference Value (optional)
- High Reference Value (optional)
- Result Comment (optional)

All five of these data properties appear to represent the result being reported. I've seen all five stored as fact values in fact tables. Using my analysis recommendations, I would only define the Clinical Result and Result Comment as facts, and I'd look for ways not to store the comment as a fact if I could.

Of the five columns, the Clinical Result is the easiest to recognize as a legitimate fact. It has an unlimited range of possible values, and it is something users will want to count (e.g., How many A1Cs above a certain value last month?) When quantitative, the value can also be subject to basic aggregation in some form (e.g., median glucose over the past 12 months). The Clinical Result also says nothing about the values in the other four columns. Clinical Result is a solid fact that can be sourced into the fact table.

The Abnormalcy column fails my fact criterion. It is certainly something that can be counted, but that's not really enough to justify making a fact by itself. It is suspect because there are only five values associated with the column. Imagine storing hundreds of millions of rows in your fact table, each containing one of those five values. The value in the Abnormalcy column is also describing a quality of the Clinical Result column: its abnormalcy. This means that for every one of those hundreds of millions of rows in the fact table, there is already another set of hundreds of millions of rows containing the actual Clinical Result value that the Abnormalcy data describe. Instead of double-storing all of those fact rows, I would move the Abnormalcy property to the Quality dimension in order to dimensionalize the original Clinical Result to its Abnormalcy value. This completely eliminates the need to store hundreds of millions of additional facts while only loading five new rows to the Quality dimension (in a new Abnormalcy subdimension).

One way to recognize that one source column is describing a quality of another column is to look at how the two columns are likely to be queried by users of the warehouse. If one property will typically be used to qualify a search for the other property, then it is very likely to be something that should have been used to dimensionalize that fact. In this case, the Abnormalcy property ends up in the Quality dimension, being used to dimensionalize the Clinical Result fact. However, if these two properties were instead loaded as

two separate facts, user queries would still be possible, just more complicated. Typically, the Abnormalcy facts would be queried as a subquery in order to better filter the Clinical Result facts. A common query might look for all lab results that were Critical High as a subquery, subsequently returning the actual Clinical Result values for those lab results in the broader query. That complexity is avoided by pushing the Abnormalcy property into a dimension. The new query would simply look for Clinical Results for any lab result dimensionalized to Critical High. Simpler queries require less user training and always perform better than the subquery-oriented approach.

The Low and High Reference value columns are very poor candidates for facts when my recommended criteria are applied to their analysis. They do not represent data that a user would need to count (e.g., How many results were received with a low reference value of 12?) because such a count carries no significant biomedical meaning. Also, while the range of possible values in these columns might be unlimited, the actual variation in their values will be fairly small, especially compared to the number of lab results against which they might be recorded. A particular combination of low and high reference values is likely to be repeated in the fact table for tens of millions of rows. Lastly, the purpose of the low and high values is to describe why the Clinical Result was assigned the Abnormalcy value with which it is associated. These properties are actually qualities of the Clinical Result. More accurately, they are properties of the lab test procedure against which the results are being recorded. Since the result is already being dimensionalized to the lab test in the Procedure dimension, the low and high reference values most likely belong over in that dimension as part of the definition of the lab test instead of being placed in the fact table for each result in which they are referenced.

You might have recognized this scenario as exactly what you dealt with in Chapter 6 when loading Lab Results into the Alpha version. The low and high values were combined to form a Reference Range, which was then used to create a Lab Test Reference Range subdimension in the Procedure dimension, the separate new entry being required because many lab tests exhibit more than one legitimate reference range.

The last candidate fact considered in the lab result source was the Result Comment column. Comment data columns that can contain almost any text are typically loaded into the warehouse as facts. You usually don't count these facts, but because they have an unlimited range of values, you can't avoid placing them in a fact table. However, it's important to recognize that these comments are qualities of the other facts. The comment says something about the Clinical Result, just not in a way that you can easily see that connection and push the comment into a dimension as you did with Abnormalcy. Recognizing it as a quality of Clinical Result in your analysis will open up other analytical opportunities for consideration.

I always check whether comment fields are truly free form text. You typically consider a comment field to be something into which a user could type any text.

It's this freedom to contain anything that results in most comments being stored as facts. In many computerized systems, comment fields contain some short block of text that a user has actually selected from a pick list of some sort. There might be only a few dozen possible comments that are actually received in a dataset. If so, then the value of the comment can be dimensionalized so the core fact can reference the dimension entry, and the comment fact need not be stored at all. In the case of a lab result source with hundreds of millions of results in it, even if only a third of those results contained the optional comment, tens or even hundreds of millions of unnecessary facts could be avoided in the fact table. Even if there were tens of thousands of possible comment texts, it would still be a significant savings to store those comments in a dimension rather than having each occurrence of each value stored as a fact value. If the comments semantically involve one of the dimensions against which data are being collected, the comment would be dimensionalized into a subdimension in that dimension. If the comment is dimensionally neutral, meaning it only describes the other fact values, the comments would be dimensionalized into the Annotation dimension.

Comment data are undesirable in the fact table because it almost always describes other data somewhere in the warehouse that would be better served if that data were dimensionalized to the comment data directly. Comments are also problematic because, as free text values, they are typically considered HIPAA-protected data that can't be viewed during any deidentified access to the warehouse. The free form nature of many comments forces us to accept these data as facts for now. In the Gamma version, you'll add curation tools that extract semantically useful information from these comments to create annotations in the warehouse dimensions that are then used to extend the dimensionality of related facts. At that time, if the lab result comments contain information that can be identified as meaningful, you'll extract it and use it to annotate the Clinical Result facts that are of most interest to users. You'll need an effective data governance body in place by then because many of the annotation and curation options available will entail making choices about data ownership and protection that can only be made by your governing body in cooperation with your HIPAA control committee. I don't recommend trying to make those choices in this Beta version because many of those issues would cause extensive delays in your implementation schedule. Just concentrate on getting those comments into the fact table so they can be curated in the Gamma version.

Allergy Assessment Facts

Let's try another example of some source data that will use the same criteria, and make some of the same analytical choices, as the Lab Result facts. An allergy assessment source table will contain many of the common data properties that are quickly recognized as dimensional references: patient, encounter, provider,

allergen, assessment date, and onset date. A positive assessment will also contain two columns that might be candidates for facts: Severity and Reaction.

Upon analysis, the Severity of the allergic reaction will be dimensionalized into the Quality dimension for the same reason that the Abnormalcy column in the Lab Result was. The Severity column only has a few distinct values (e.g., possibly a 0–7 score) that would be repeated across many millions of allergy assessments. Moving the data to the Quality dimension allows allergies to very easily be queried by Severity.

Profiling of the Reaction column in the source might determine that a small list of distinct values accounts for the vast majority of entries, while the remaining values vary widely. You might learn that this column is populated in the EHR when providers pick a reaction value from a specific pick list in the system, and they have the option to enter a free text reaction instead. If it weren't for those free text entries, you would definitely choose to dimensionalize the Reaction column. In this case, you wouldn't use the Quality dimension. Reaction data are semantically closest to what the Pathology dimension represents, so you would place the new Reaction subdimension into the Pathology dimension and dimensionalize the positive allergy assessments to one of those entries.

The presence of free form Reaction values complicates your source analysis. You need to profile how often a pick-list value appears in the column versus how often a free text variant is present. How many distinct free text Reaction values are there, and over what period of time? Are there distinct free text entries that have actually been used more than once, perhaps being reused by the same providers over time? Through your analysis, you are trying to decide if Reaction can be moved into the Pathology dimension despite the presence of some free text entries. Sources where pick-list data that can be overridden with free text values actually contain far fewer free text variations than you might have anticipated. Moving the data into a dimension is a viable alternative to keeping data in a fact table where the dominant pick-list values might need to be repeated tens of millions of times. It is very likely that users will want to query allergy data by Reaction and Severity, so having these data properties dimensionalized will simplify queries and improve database performance.

As a technical data processor, you'll probably be tempted to reach a design compromise. You might decide to store the Reaction in the Pathology dimension when it contains one of the pick-list values and keep Reaction as a fact value when it contains a free form text value. Don't do it! First, it requires a user to look for the same information in two different places in the warehouse. Second, it requires the source intake ETL to know what the possible pick-list values are and to keep track of them as they change. What if a new pick-list value matches a previously entered free text value? And what does the free text reaction fact point to in Pathology? Do you have to introduce some form of "Other" reaction into Pathology to account for those facts that used free text? The added complexities create more problems than the compromise is intended to solve.

Each data element you map into your warehouse should have one—and only one—normalized place where users can be confident in looking for it. The reaction is either a Pathology entry or a fact. Don't complicate queries by trying to create a hybrid approach.

You might want to argue that the free text values in the Pathology dimension could overwhelm the subdimension for Reactions. If there are 400 valid reactions in the available pick list, that's 400 entries in the Pathology dimension. If upon initial load, there are 2000 distinct free text comments in the retrospective data, that's 2400 reactions in the dimension. Should you be afraid of this ratio? A subdimension with 2400 entries in it is still very small compared to most of the subdimensions in the warehouse. Suppose that after the first few years of prospective data being loaded into the warehouse, an additional 15,000 distinct free text reactions have been added. Should you be concerned now? No, you shouldn't. Your warehouse dimensions will routinely control tens of thousands, hundreds of thousands, sometimes millions of entries in each subdimension. A little 2400–17,400 entry Reaction subdimension isn't going to impact warehouse performance in the slightest. Only an IT person would ever be concerned about this distinction, and that's not a basis for changing the way data maps into your warehouse.

Unless you find that there are millions of distinct reactions in your source table, I strongly urge you to dimensionalize the data into Pathology. When you add semantic tools and annotation capability to the warehouse in the Gamma version, you'll probably learn that many of those free text Reaction values are simply variations on the values you have from the pick list. The most common variant I see is when a provider wants to list two of the reactions from the pick list in a single entry. Your Gamma tools will be able to split these apart to reuse the individual pick-list entries, eliminating the impact of the free text row in your Pathology dimension. Loading the data as dimensional in your Beta version enables that processing to add more value when you get to your Gamma version.

Note that moving both the Severity and Reaction candidate facts into the dimensions leaves nothing to put into the fact table for each entry in the source table. When this scenario occurs, insert a “Factless” fact into the fact table. A Factless fact doesn't contain any sourced data in the value column of the fact but still allows for the dimensionalization of the fact against the numerous dimensions that define the fact. Factless facts are actually very common when analyzing sources. When all of your candidate facts end up in the dimensions as a result of your analysis, you are left with a Factless fact. This means that, while reducing the number of candidate facts reduces the volume of data stored in the fact table, the minimum number of facts you can get down to is one. Resist the temptation to store one of your candidate facts in the fact table by rationalizing the choice as being appropriate since at least one fact needs to be stored anyway. The fact table is the most expensive place to store something, so don't use it to store something that could have been dimensionalized.

Financial Charge Facts

Most of the criteria that are needed to analyze source data for mapping into the warehouse have been covered by the lab result and allergy examples. Let's take a look at another example that introduces a slightly more complicated mapping challenge, precisely because there are many ways to resolve the mappings: the *degree of freedom* problem. The degree of freedom problem arises whenever you have a group of facts, one or more of which can be derived from the others, so it seems redundant to store all of those facts. Common examples of this problem are source datasets that contain financial charges. These datasets usually contain columns for a Unit Charge, and Number of Units, and an Extended Charge.

In this example, you have a situation with three data points where any two can be used to derive the third (i.e., three data points contain only two degrees of freedom). This should immediately tell you that you want a maximum of two facts per entry. Ideally, you expect to use your analysis process to reduce three facts to only two by finding a way to dimensionalize one of the data points. Whatever two you choose to store can be used during a query to derive the third, so it would be redundant to store all three. You begin by noting that users are most likely to want to query or aggregate the Extended Charge amount. That's the data point you'll most likely decide to place in your fact table. The challenge is whether and where to place the Unit Charge or Number of Units values.

There are a number of options for functionally normalizing these data, but let's start with an approach that is guaranteed to get the data into the warehouse even if it isn't necessarily the best option. If you profile the Number of Units data in your source, you are likely to discover that there aren't as many distinct values as you might expect. My experience is that the vast majority of charges involve only a single unit. Another large percentage of entries will typically involve ten or fewer units. Of the remaining entries, the number of distinct quantities is manageable; perhaps 1000–5000 distinct values over a very long time period. The number of distinct entries in this column is sufficiently small and manageable to be able to shift the data into a new Count subdimension in the Quality dimension. By placing the Extended Charge in the fact table and dimensionalizing that fact to the appropriate Count in the Quality dimension, the Unit Charge becomes derivable in a query. By storing a single fact in the fact table, all three data elements are then available. Since some warehouses store hundreds of millions of charge facts, the savings from not storing a second or third fact is significant.

While this analytic approach is guaranteed to get a set of degree of freedom data into the warehouse, it isn't really optimal from a functional normalization perspective. Additional analysis is warranted to try to get a more functionally specific mapping in place. The problem is in dimensionalizing the Number of Units and requiring that the Unit Charge be derived. That seems functionally backward and will bother your financial warehouse users. It is the Number of

Units that seems transactional of the two, so it should have been thought of as the second fact. While you still would not want to store a second fact, it seems like it would be the Number of Units that you'd want to derive in this situation. It seems odd to load the Number of Units as a quality of the Extended Charge. The Unit Charge seems like the master data that would be found in one of the dimensions.

An alternative might be to place the Unit Charge into a dimension, and not store the Number of Units, at all. The approach is the same as you would have used to place the Number of Units into the Quality dimension; only this time you'd be profiling the number of distinct values in the Unit Charge column of your source. The number of values will often be manageable enough to be placed in a dimension. In this case, instead of creating a Count subdimension into the Quality dimension, I'd recommend adding an Amount subdimension to the Accounting dimension. Even if the column contains every one-cent value from 0 to 10,000, there will still only be a million entries in the new subdimension. If you limit your load of the new subdimension to only those values that actually occur in the data, you'll find that the number of entries is significantly smaller than that, perhaps a few tens of thousands.

This solution to the degree of freedom challenge should be more acceptable to your financial users. The derived value at query time will be the Number of Units, making it look more like a fact and allowing the Unit Charge to be treated as dimensional. This will better align with user expectations and provide a higher level of user satisfaction.

You might want to consider creating both of the new subdimensions. If you dimensionalize the Number of Units to the Quality dimension and dimensionalize the Unit Charge to the Accounting dimension, you will have closed the loop on your degree of freedom problem. All three elements of the calculation would be available, and your users could decide which one to derive from the other two at query time. The choice could be based on optimizing any particular query strategy that might be used.

One query choice might be to consider the Extended Charge in the fact table as the value to be derived. Since the fact table is the most expensive place to store and retrieve data in a star schema warehouse, it might help the performance for certain queries to treat the Extended Charge fact as though it were factless and derive its value from the two dimensional entries in the Quality and Accounting dimensions. Imagine a case where you might want to write a query to aggregate (e.g., Sum) the charges for an encounter. At its simplest, your query would filter on the encounter in the Interaction dimension and the Metadata dimension entry for your Extended Charge fact and then aggregate the value in the fact table as a sum. The result is the total charges for the encounter. Alternatively, your query could join those facts to your entries in the Accounting and Quality dimensions and instead aggregate a derived product of the Number of Units in Quality and the Unit Charge in Accounting. The result would be the same total charges for the encounter.

At first glance, the first query seems like it would be much simpler. In fact, it is much simpler to write, if your users are actually writing SQL queries; but, since users tend to use a semantic layer to access the warehouse through a business intelligence tool, any extra effort required to write a slightly more complicated SQL query is embedded in the tools. It's not something that users need to worry about. Your users will probably drag an Extended Charge object into their query regardless of which of these two options is coded behind that object. Therefore, simplicity of code is not an adequate criterion for making this choice.

The reason you might want to consider deriving your Extended Charge fact from the dimensions is performance. The alternative query doesn't actually select any columns from the fact table. The fact table is only referenced in the query as a means to establish the connections among the dimensions. This means that your database optimizer will likely choose to make those connections using the various indexes that are defined in your database rather than physically accessing the actual fact table. Index accesses are much faster than table accesses and the Number of Units and Unit Charge dimensional connections can be established through the intersection of the appropriate indexes. Rather than obtaining a few thousand fact values from a fact table containing potentially billions of rows, the optimizer can obtain the few thousand overlapping index entries from dimensions having only tens or hundreds of thousands of rows. The index-based queries will always be much faster than the value-based queries from the fact table. The user doesn't need to see the distinction between these two query options if your business intelligence layer is well designed and implemented.

This analytic mapping strategy will work for any degree of freedom scenario that presents itself in your source data. Other common examples include the triads of Unit Dosage, Quantity, and Total Dosage in medication data sources; Height, Weight, and Body Mass Index in vital signs data sources; and Incision Time, Close Time, and Procedure Duration in surgical data sources. Each triad presents opportunities to apply degree of freedom analysis to the mapping of the apparent multiple facts in the data warehouse.

Before we leave this example, let's conduct some extra analysis of what we've placed in the Accounting dimension. While having a subdimension for Amounts solves your degree of freedom challenge, it is unsatisfactory as a method for normalizing data into the warehouse. Its use has a data processing feel to it that we've been trying to avoid in your analysis. Ideally, we'd like to have a solution that accomplishes the same separation of the Unit Amount from the facts, but doesn't require the creation of an arbitrary and contextless new subdimension.

I recommend that you look for a perspective in your dimensionalization of the Extended Charge that would suggest a more functionally appropriate context for storing the Unit Charge. In my experience, charge facts will be dimensionalized to an entry in the Accounting dimension that represents data you will be sourcing from the organization's Charge Master. Without some form of Charge Master, users will have a difficult time querying the various charges in the fact table because there will be no context for knowing what each charge represents.

This creates an opportunity to analyze the Charge Master to see if it can serve as a repository for the Unit Charge values you are analyzing.

Don't rush your analysis in this area. Initially, it might seem that the Charge Master would be an obvious place to put Unit Charges. While it might be legitimate for some charge types, you would need to be able to confirm that it is appropriate for all charge types in order to avoid building your Amount subdimension. The criterion of interest in your analysis is whether or not the only thing needed to connect a Charge Master entry to a distinct Unit Charge value is time. If at any one time there is only one Unit Charge for an entry in the Charge Master, Unit Charge can be moved into that subdimension as a property of the Definition table. Unfortunately, this condition is rarely met for all entries in the Charge Master.

An example where the condition is not met in many environments involves medication charges. Every medication might have its own Unit Charge, but there might not be a Charge Master code that is unique for every medication. If there's a generalized medication charge account in the Charge Master, the Unit Charge for that item will vary based on which medication is being charged. Knowing the charge code alone doesn't determine the Unit Charge, so you struggle to eliminate the need for your separate Amount subdimension. The same problem can arise in other dimensions, such as room charges in Facility, lab test charges in Procedure, or professional service charges in Caregiver. If your Charge Master uses generalized charge codes, the Charge Master can't serve as your repository for Unit Charges.

With some creative source analysis, you still have an option for avoiding your generic Amount subdimension. You could instead analyze the data source and create your own Charge Master Detail subdimension in which you can embed your Unit Charge values. Take your generic medication charge code as an example. If you recognize that this charge code is generic, you could choose to source the data in such a way that you concatenate your Medication ID (which you might already be using to dimensionalize your charges to the Medication subdimension) in the Material dimension, to the Charge Code to create a more detailed charge number that would be a hierachic child of the original charge code in the Charge Master subdimension in the Accounting dimension. For this to work, you still have to show that the only thing needed to establish a Unit Charge for one of these detail charge codes would be time. If not, this approach doesn't solve the problem, and you revert to your Amount subdimension.

In my experience, you'll eventually end up with some form of hybrid of these different approaches. Your Accounting dimension will include the Charge Master and your arbitrary Amount subdimension. You'll also have some form of Charge Master Detail subdimension, although you might discover legitimate data sources for some of its contents rather than arbitrarily generating it from the fact sources you receive. All of the subdimensions to which you add a Unit Charge column available in the Definition table (e.g., Facility, Procedure, Caregiver) will be populated in some rows and NULL in others. For this Beta version, I recommend

mapping your Extended Charge facts into the warehouse in such a way as to have two connections to the Accounting dimension: one for the Account role in the Charge Master or Charge Master Detail and one for the Amount role in the Amount subdimension. The Amount role guarantees there is always a Unit Charge even when the Charge Master doesn't have one.

In your Gamma version and beyond, you can continue your analysis of these data to confirm the data patterns you analyze actually continue to occur in practice. You can query those cases where the Account role yields a non-NULL Unit Charge and compare that value to the Unit Amount obtained following the Amount role. If they aren't the same, you can investigate why. Even when they are the same, they might not stay the same if late-arriving changes in the Charge Master subdimension alter the Unit Charge values for the time periods in which there were charges that used to match. The data are very dynamic, which is why I recommend taking a straightforward approach to mapping your facts into the two distinct perspectives. You aren't trying to reproduce the transactional logic of your financial systems. You need only represent the data as you receive it, in order to give your users access to investigating these dynamics.

Using your analysis to clarify that some of the properties in your data sources that might initially be viewed as facts are actually qualities of other facts will drastically reduce the number of facts you map into your fact tables while improving the ability to query the facts you do define through more enriched dimensionality. Many of the properties I move through this analysis end up in the Quality dimension because they represent actual qualities of the remaining facts (e.g., Severity, Abnormalcy). Quite a few properties end up in the Annotation dimension because they represent comments or remarks that annotate the remaining facts. Finally, many properties end up in other reference dimensions because they fit the ontological purpose of those dimensions (e.g., Reactions as Pathology) or they add detail to something already placed in that dimension (e.g., Reference Range under Lab Test in Procedure).

The differences driven by this analysis can be very dramatic. I've applied this set of techniques *post hoc* to existing fact table mappings in ways that have caused clients to delete hundreds of millions of facts from existing warehouses. If it were just data volume, I probably wouldn't be as concerned. What you really gain is the ability to query facts using simpler queries that run much faster because you're filtering is now on the dimensions rather than through orthogonal subqueries to other related facts.

Fact Superseding

Many source datasets that will be brought into the warehouse include data that can be updated in the source system without leaving a trace of previous data values. If the data values involve updates to a single dimension in the data warehouse, your standard processing for slowly-changing dimensions will manage the change effectively. If the changing data involves facts in your

warehouse, you'll need to allow for the management of the changes. Let's take a look at a few examples:

- The source patient record maps primarily to the Subject dimension but also results in a fact to connect the patient to his or her primary care provider (PCP) in the Caregiver dimension.
- The source patient record maps primarily to the Subject dimension but also results in a fact to connect the patient to include his or her mailing address in the Geopolitics dimension.
- The source encounter record maps primarily to the Interaction dimension but also results in a fact to connect the encounter to the patient's bed assignment in the Facility dimension.

Each of these examples involves a single fact to associate the two dimensional entries mentioned to reflect the relationship described. Updating any of these relationships in the source system will result in the record being sourced for processing and in the creation of new facts. While the source system might not identify previous PCPs for patients, or patient mailing addresses, or bed assignments for encounters, those earlier facts still remain within the warehouse. Over time, the warehouse accumulates many generations of data values that are no longer available in the source systems because those systems were designed to overlay new data values onto old values.

Since the source system doesn't include a longitudinal view of these facts, you want to reflect that source-based limitation in order to avoid confusing users who might query the warehouse expecting to see only what they would see if they looked in the source system. The source system doesn't keep track of past values of the fields in question, so you can't source an update or a delete transaction to modify or eliminate the earlier facts. You need to be able to make any necessary changes through the standard processing of the new value arriving from the data source. You'll do this by indicating a superseding profile for each fact in the metadata defined for mapping the new facts.

Within the metadata for any particular fact, you have the option to specify a superseding profile, two of which would be relevant to our three examples:

1. One fact per Subject (overall or per year, month, or day)
2. One fact per Interaction (overall or per year, month, or day)

Both the PCP and mailing address examples from the source patient record would be declared as occurring once per Subject. The bed assignment from the source encounter record example would be declared as once per Interaction. In all three examples, the ETL logic that adds new facts to the warehouse will automatically change the data state of previous versions of the fact from Active to Superseded, making them available for queries that look for them but are otherwise out of the way for standard queries that expect only active facts to be returned.

Note that the newly added fact won't necessarily be the active version of that fact after processing. The final active fact will be the one with the most recent event date in the Calendar dimension. This will usually be the most recently added fact, but not necessarily if historical data that are older than the currently active data are being loaded. An example is sourcing a dataset that has historical patient contact information. In the source, each patient has a mailing address to be loaded into the warehouse. Whether the patient addresses being loaded will end up as Active or Superseded facts in the warehouse depends upon what addresses are already known for each patient and the date on which each of those addresses was known. If an address coming from the new source has a date earlier than the current active address for the patient, the new fact will end up as Superseded. If an address in the source has a date later than the current active address, the new fact will be Active and the fact already in the warehouse will be changed to Superseded.

By making sure that only one of the patient address facts is Active at any one time, the general user query that filters or segments patients by geopolitical entities is simple to write because most queries look only at Active data. Building a query that returns all known addresses for a patient simply requires expanding the data state filter to include Superseded facts. Looking for a patient address as of some particular date requires selecting both Active and Superseded facts, where the effective date of the fact for that patient is the maximum value that is equal to or less than the date of interest.

The ability to have Active and Superseded facts supports a generic capability to keep longitudinal data in the warehouse even though the capability is often lacking in the source systems from which the data were received. I use this capability to align the facts in the fact table with user expectations that are often determined by their knowledge or awareness of source systems. I recommend that you don't store multiple active facts if the source system no longer contains those facts. When the source keeps only a most recent version of these data, try to mark all previous versions as being superseded by the current active version.

The main benefit of this approach is that it aligns the active data with the available source data without actually having to discard previous data values. Users who are familiar with the sources, but might not be familiar with the warehouse, can query the warehouse and see the results they would expect based on their knowledge of the data. Users can always expand their queries to include Superseded data if they want to see all of the earlier values, something they don't have the option of doing back in the source system.

If the superseding profile is not defined as necessary, it will result in multiple facts being stored over time that no longer match current data in the source system. In these situations, you can still query the facts to find current data, but the queries are more complicated and require the writer of the queries to know the superseded profile that would have been used. Typically, the effect of this scenario can be managed in a query by returning results that maximize the

calendar date; but those queries are much more difficult to write and exceed the skills of many typical warehouse users. Superseding the older facts eliminates that need.

Dimensionalizing Facts

The scenarios under which data will need to be sourced for any particular data warehouse will vary widely. While many sources will not be in forms that can be accessed immediately using SQL, there is a host of access tools and converters on the market that can be used to transfer data from one form to another. For this discussion, I'm assuming that any source data available can be presented to the ETL subsystem as a relational table.

The simplest data sources will be highly normalized structures where a single row in a source table represents either a single-dimension construct or fact at a point in time. The idea of a point in time is very important. The warehouse design includes an ability to maintain different values of certain slowly-changing properties over time. It is important to always know the appropriate timeframe for all data loaded into the warehouse since new or old data can be loaded in any order. In fact, out of sequence loads are the norm, not the exception. Each new source loaded into an operational warehouse will typically go back and load the history of that source. That data will have effective dates that will intersperse throughout any data already loaded into the warehouse prior to that source being loaded. As long as each source data fact or dimension construct has an appropriate timeframe associated with it, the warehouse design will keep everything in sync and well ordered.

The level of data normalization can affect your view of the data that need to be sourced. Foreign keys that point to master data that would be loaded separately as dimensional data need not be traversed when sourcing new facts; but when a foreign key points to a data aggregate, it might need to be traversed during sourcing. Examples of this include data tables in parent-child or header-detail relationships. The parent or header data might represent common data that need to be present to correctly interpret the data in the child or detail table that is being sourced. Sourcing a lab result table might require joining to the aggregating lab order table to pick up the order status of the total order as part of the dimensionalization of the individual result. While your source design is nominally focused on lab results, your code will actually access the joined order-result combination.

Once you've joined those two tables, you'll have to decide what other information you might want to source from that table as part of your sourcing of the result facts. You might be tempted to also source the attending physician from the header so it's available on each result fact. Presumably you would have sourced the attending physician when you sourced the lab orders themselves, but it probably seems like a good idea to have the attending physician on each

result fact, as well. I refer to this addition of more source data to the fact as *updimensionalizing* of the source. The attending physician wasn't really part of the data-normalized result; but it will be useful when writing queries against results because you won't have to access the header facts if you want to include or filter on the attending physician. You've increased the dimensionalization of the lower-level facts in order to avoid more complicated queries. In the Gamma version, we'll cover how to updimensionalize facts after they are loaded, but doing it during data sourcing is an inexpensive way to accomplish it early, and it provides significant extra value to users, if included at appropriate points.

The risk of updimensionalizing a fact arises from whether or not these facts would need to be redimensionalized if there is a change in the entries upon which the updimensionalizing was based. What if the patient's attending physician changes after the load of the orders or results? If, at the time of the order, the attending physician in the order was meant to always be the attending, then a change won't impact the order and the updimensionalization of the order is low risk. I only recommend updimensionalizing data during sourcing if the resulting extra dimensionality of your facts will remain stable over time. You don't want to create a situation where you become responsible for transaction processing to keep your dimensionality on certain facts up to date. When there is doubt, don't updimensionalize during your source loading. There will be many opportunities to updimensionalize the data later in the Gamma version, so avoid risks in the Beta version that might slow down your loading of data.

If the source dataset for which you are dimensionalizing your facts contains only one type of fact, it's probable that all of the keys in that source represent dimensionalization of that particular fact. Unfortunately, though, the one row—one fact source is relatively rare. Most source tables that are destined for your fact tables will contain more than one fact, and these distinct facts need to be differentiated during source design. A source as simple as your lab result table will probably have at least two facts: the actual clinical result and a comment or remark column. In analyzing these facts, beware of differences in how they might be dimensionalized. Differences need to be recognized and handled. The lab result table usually has a unit of measure column that is tied to the Clinical Result value. However, that unit of measure typically has nothing to do with the comment or remark column. One fact will dimensionalize explicitly to the Unit of Measure dimension using the value in the source, while the other fact will dimensionalize to Unit of Measure as a narrative text default. Other fact sources will be even more complex. A vital signs observation table might have 6–10 distinct facts in it. Often none of these will have an explicit unit of measure in the source table. Your analysis will need to determine implicit units of measure in the Metadata dimension based on profiling of the source values themselves, and the fact that the units of measure for vital sign observations are well known to healthcare professionals even though they are almost never noted in the relevant datasets.

Beyond dealing with multiple column-to-column fact types within your source data, another common variation within a source occurs at the row level. Different rows in a source table might be dimensionalized very differently, requiring distinct processing during loading. Your lab result table might have to treat the clinical result and comment columns differently depending upon other aspects of data occurring in the row.

One common variation at the row level for lab results is when some lab tests for which results are being reported have a reference range included in the source and others do not. Rows exhibiting these differences will map to the Procedure dimension based on the presence or absence of values in those reference range columns. Some will map only to the lab test, while others will map to the lab test with reference range. Another common variation in lab result rows is whether they actually report results (the nominal intent of the source) or nonresults, as explored earlier in this chapter.

Row-to-row variations are often independent of each other. Whether a lab result is clinically erroneous is typically independent of whether the lab test had a reference range defined for it. As a result, row-to-row variation will drive more complexity in your sourcing than column-to-column differences will. Your lab results will actually occur in four fact-metadata mappings, based on the presence or absence of reference ranges and the validity or erroneous nature of the results. The sourcing of the data will need to take all of that into account, making source data analysis even more important for sources with column-to-column and row-to-row variation.

An extreme example of row-to-row variation might possibly be an ICD-9 master table, each row of which represents a single ICD-9 code. ICD-9 master data actually map to more than one dimension in the star schema you are building. The source will typically include a column that indicates whether the relevant ICD-9 code represents a diagnosis or a procedure, each of which needs to be mapped to a different dimension in your star model. In addition, some of the ICD-9 codes are “E” codes that represent environmental factors. The “V” codes in ICD-9 represent propensities or histories of pathologic conditions. As a result, a source table for ICD-9 might require mapping source data to four different dimensions in your designed star model: Diagnosis, Procedure, Pathology, and Environment.

Sourcing Your Data

As you analyze and develop strategies for loading data into your warehouse, you'll begin to create the queries that will pull data from your sources. Because each source is a little different, you'll ultimately end up with a job or workflow that is unique to each of your sources, even though all of the ETL workflows that receive the data will be generic in all subsequent steps. That generic ETL

architecture is described in the next five chapters, so the extraction query you write to pull data from each source is the first and last source-specific code that you'll create.

Chapter 9 begins with a description of the generic functions and features needed by every job that sources data from the warehouse, referred to as source data intake (SDI) jobs. Here we're interested in the innermost subqueries in those jobs, which actually pull data from the sources. Essentially, every SDI source query is focused on getting data out of its source:

```
SELECT * FROM SOURCE_TABLE;
```

Almost none of the source queries you will create will be this simple, but it's a great way of thinking about what you'll need to do in your queries. You should build on this simple query to include any details necessary to support the mappings you've developed for placing data in the warehouse, or to support your choices regarding how and when you want to load your data.

Selecting Columns

Selecting all of the columns in a source dataset is appropriate if you know you want all of them loaded into the warehouse, although it is quite rare to load all of the columns of an operational source dataset. Invariably, columns that you'll decide you wouldn't want represent internal or state-specific variables in the source application. If there are few columns to be omitted, selecting all columns is still appropriate in order to keep your query simple. If the number of columns to be omitted is more than trivial, especially if you'll be sourcing a dataset with many rows, you should make your selection of columns more specific:

```
SELECT CLINICAL_RESULT,
       ABNORMALCY,
       LOW_REFERENCE_VALUE,
       HIGH_REFERENCE_VALUE,
       RESULT_COMMENT
  FROM SOURCE_LAB_RESULTS;
```

Sourcing exactly the columns you want to load into the warehouse minimizes the resources needed for the ETL loads, but it also constrains your loading in small ways. Be careful to include not just the columns you know you want to load but also the columns you consider likely to be loaded eventually, or columns your analysis is still considering. The generic ETL architecture that begins in the next chapter will automatically discard sourced columns for which you don't supply target mappings in the Metadata dimension. Including extra columns won't waste resources within the various ETL jobs. The extra space is used only in the staging tables for your sourced data, so your approach to

omitting columns should be very conservative. Select columns you might load so you minimize both the number of times you need to update your selection code and the number of times you need to reextract your sources when you change your mind. If you ultimately find that you end up selecting columns that you choose not to map into the warehouse, you can remove those columns from your code toward the end of the Beta version. During your development effort, these queries should be as stable as possible.

As part of the analysis of your source data, you will often find a need to source multiple variations of the source data. These extra derived columns should be built into your selection queries, so the downstream ETL processes don't need to know whether a column was actually present in the source or was derived from data in the source. Often the derivations needed are very simple, so do them as part of the source selection to avoid needing any nongeneric code within the ETL to deal with those situations. For example, it's common to take the names of people in source tables and derive multiple variations using those names:

```
LAST_NAME,
FIRST_NAME,
CONCAT(LAST_NAME, ' ', FIRST_NAME) AS FULL_NAME,
CONCAT(FIRST_NAME, ' ', LAST_NAME) AS COMMON_NAME,
CONCAT(SUBSTR(FIRST_NAME, 1, 1), SUBSTR(LAST_NAME, 1, 1)) AS NAME_INITIALS
```

To the ETL processes, these five columns are equivalent. It doesn't matter to the warehouse loading processes which columns were derived from the others. It is exactly this kind of source-specific knowledge that you want embedded in your sourcing jobs in order to allow the rest of your ETL subsystem to be developed generically. Based on your analysis, the derivation of new source columns can be arbitrarily complex. Here's an example that creates a Reference Range column from a combination of the two Low and High Reference value columns:

```
LOW_REFERENCE_VALUE,
HIGH_REFERENCE_VALUE,
CONCAT(LOW_REFERENCE_VALUE,
       ISNULL(CONCAT('-', HIGH_REFERENCE_VALUE), '')) AS REFERENCE_RANGE
```

The complexity of this derivation arises because, while many reference ranges have a low and a high value, some do not. When there is no high value, the low value is considered the reference value. The derivation of Reference Range takes this distinction into consideration by omitting the high value (and the inserted separation hyphen) when it is not present. The resulting derived Reference Range has the correct value for any of the permutations that arise in the source.

Beyond derived columns, you'll also encounter situations where your analysis has focused on the intersection of more than one dataset, often viewed as a parent-child or header-detail relationship between tables. In these cases, you might have chosen to source the combination of tables as your source because

you want to select a column from the header entry as though it were present in every detail entry. For example, if you want to source Lab Results but want each result to include the provider and status from the order against which the result was being recorded:

```
SELECT O.ORDER_STATUS,
       O.PROVIDER_ID,
       R.CLINICAL_RESULT,
       R.ABNORMALCY,
       R.LOW_REFERENCE_VALUE,
       R.HIGH_REFERENCE_VALUE,
       R.RESULT_COMMENT
  FROM SOURCE_LAB_RESULTS R
 INNER JOIN SOURCE_LAB_ORDERS O ON (R.ORDER_ID = O.ORDER_ID)
```

Your selection is being made from the joining of the two tables, although you will continue to refer to the SDI job within which this selection will be made as the Lab Result sourcing job. Be careful not to complicate the situation by creating more confusing names. Calling this example a Lab Order and Result extract might create the impression that the orders are being sourced, and they are likely to be sourced elsewhere as a result of a separate analysis of the source for that purpose. The joining of the detail table up to the header table is for the purpose of properly extracting the detail table, so the source job should be identified accordingly.

Be careful about choosing to join source tables that are not in these header–detail relationships. Keep each source extraction as narrow and focused as possible in order to keep all of your workflows clean and independent of each other. Analysts often want to identify patients in the data using a natural key (e.g., MRN), but the actual source system (e.g., the EHR) doesn't use that value. This desire results in a source job where one element of data from that source almost always has to join to another table in the source to get the required natural key:

```
SELECT K.NATURAL_KEY AS MRN
  FROM SOURCE_PATIENT P
 INNER JOIN SOURCE_PERSON_KEYS K ON (P.PERSON_ID = K_PERSON_ID AND K.TYPE = 'MRN');
```

Identifying data by the patient's MRN, even when extracting data that only the ETL jobs will see, results in a more complicated selection process, and this extra complication becomes embedded in almost all of the source selection extracts from that source system. Avoiding redundancy is one of the objectives of the generic ETL subsystem architecture, so I recommend an alternative approach.

First, extract your source data using whatever keys your source systems uses. That might mean sourcing data using its internal system keys typically never seen by system users:

```
SELECT PERSON_ID
  FROM SOURCE_PATIENT;
```

Second, if one of your source systems has alternative keys for identifying the data you are extracting, treat those alternatives as new and independent sources that would be selected for extraction in only one SDI extraction:

```
SELECT PERSON_ID,
       K.NATURAL_KEY AS MRN
  FROM SOURCE_PATIENT P
 INNER JOIN SOURCE_PERSON_KEYS K ON (P.PERSON_ID = K_PERSON_ID AND K.TYPE = 'MRN');
```

The relationship between the source's Person ID and MRN is a piece of information that can be extracted and loaded into your warehouse and will result in a new alias in the Subject dimension. Having the alias defined to the warehouse means that users can reference all of a patient's data by MRN, even though none of the source extracts that selected patient data for the warehouse included, or even needed to know about, the MRN for the patient.

Selecting Rows

Beyond selecting which columns you want to select from a source, you should also include logic in your extraction query that considers which rows to select, as well. When the distinction is based on differences in your source on a row-by-row basis, it's possible that you will split the source extract query to accommodate needed differences independently. For example, if you want to source the allergy assessment columns you analyzed previously:

```
SELECT ALLERGEN,
       SEVERITY,
       REACTION
  FROM SOURCE_ALLERGY_ASSESSMENTS;
```

Based on your analysis, perhaps you concluded that negative assessments shouldn't include the Severity or Reaction. You might have learned that the vast majority of assessments are negative and feel that extracting the extra data would be prohibitive, given the large number of assessments that you'll be sourcing. Presuming that the analysis justifies splitting the extraction, you might end up with two selection queries:

```
SELECT ALLERGEN
  FROM SOURCE_ALLERGY_ASSESSMENTS
 WHERE FINDING = 'NEG';

SELECT ALLERGEN,
       SEVERITY,
       REACTION
  FROM SOURCE_ALLERGY_ASSESSMENTS
 WHERE FINDING = 'POS';
```

However, if you learned that the Severity and Reaction data are often present even for negative findings, you might decide to always extract the three columns regardless of finding the type. You could still treat the two source selections as separate, but you would no longer need to. Instead, you could leave the source selection as a single query and add whether the finding was positive or negative:

```
SELECT FINDING,
       ALLERGEN,
       SEVERITY,
       REACTION
  FROM SOURCE_ALLERGY_ASSESSMENTS
```

This final version is the type of source selection that I recommend: simple and clean. It represents a single source of allergy assessments regardless of whether any particular assessment was a positive or negative finding. This kind of selection logic allows you to take full advantage of the capabilities of the generic ETL subsystem described in the next chapter. This one data source results in the creation of two different types of facts: one for positive findings and one for negative findings. Each row of input will result in one or the other of those two facts. Users will be able to query these facts independently of each other, which will make it easier for them to work with the positive allergy assessments without getting lost in the cacophony of negative assessments.

When you build your Metadata dimension entries for these facts, you define different metadata values for each fact. For positive findings, you will likely define the Allergen, Severity, and Reaction data as required. If any of the three values is missing from the source, an error condition, warning of the omission, will be raised in the ETL. For negative findings, you'll likely treat the Allergen as required and the Severity and Reaction as optional. Empty data values in those columns will not raise any error conditions. In the Gamma version, you'll be able to declare the Severity and Reaction as unexpected or undesired, raising warning (for unexpected) or error (for undesired) conditions if values are received on the loads. Note that none of these error or warning conditions will prevent the data from loading into the warehouse. Data always load, even if a missing identifier column of input results in one of more of your facts pointing to a dimension's Unknown row.

Selecting Time

One of the requirements of a longitudinal data warehouse for biomedicine is that you always need to be able to establish where your data fall in time relative to all of the other data you might be collecting. This can ultimately affect your choice of which rows to pull from a data source as you move from retrospective processing of your data toward more prospective processing where you will need to know what you've already extracted so you don't repeat the same data. Much of the logic of controlling dates is built into the generic architecture for the ETL,

but that logic will be dependent upon always having time variables as part of every source. Every source extraction you build should include time as one of the variables being extracted. Here's the allergy assessment selection with a time perspective added:

```
SELECT FINDING,
       ALLERGEN,
       SEVERITY,
       REACTION,
       ONSET_DATE,
       ASSESSMENT_DATE
  FROM SOURCE_ALLERGY_ASSESSMENTS,
```

This addition of two columns is one of the reasons I discourage splitting fairly simply sources into separate extractions based on row type. If you hadn't kept the allergy assessment sourcing as a single process, these columns would need to be added and maintained in two places. Your objective should be to avoid this scenario. Try to have a single source selection for each physical dataset that you want to source, and let the generic metadata sort out any differences.

In this example, the metadata you build for both the positive and negative findings will likely treat the Assessment Date as required, since that's the date of the event that resulted in the sourcing of these facts. Having this date meets the minimum requirement for loading a fact into the warehouse: It must have an established time period for the event that the fact represents. That date dimensionalizes the facts to the Calendar dimension. Had the column been a timestamp, it would also dimensionalize each fact to the Clock dimension.

The presence of the Onset Date provides additional load options. At the very least, it can be used to enrich the Calendar dimensionalization for these facts by adding it to the Calendar group to which each fact is dimensionalized, when present. If desired, it could be added only to the facts that represent positive assessments, since a negative assessment shouldn't have an Onset Date. However, as with your analysis of Severity and Reaction sometimes being present on negative assessments, it might be wise to make Onset Date an optional dimensionalization to Calendar even on negative assessment facts.

As an additional enrichment of the clinical picture of your patients, you might also choose to add more metadata in order to create a new allergy onset fact whenever the Onset Date is present, although I'd limit this one to only positive findings to avoid confusion that might be created by the new fact. Since the new onset fact wouldn't be connected to the original assessment fact (other than that the audit trail would show that they were loaded at the same time from the same source), it might create confusion if you were to load onset facts for negative assessments. Depending upon your perspective in the data, the onset fact might seem redundant to the assessment fact. Indeed, their dimensionalization is the same except for the Calendar entry; but for users of the warehouse browsing records longitudinally, it might be very useful to see an

allergy onset appear in the patient's timeline, sometimes many years before the assessment against which they were recorded.

I recommend dimensionalizing the new onset to the Assessment Date as well. Doing so creates a set of cross-referencing facts that can be very useful in certain queries that are looking longitudinally at patients. The onset fact might say that an allergy that had a 2008 onset was recorded in an assessment in 2015, while the corresponding 2015 assessment fact indicates that the onset of the allergy was in 2008. If, over time, you see many onset facts for the same patient allergy, each with the same onset but different assessment dates, that can serve as an indication that new assessments are being entered into the patient's record by providers, rather than using the existing assessments that have already recorded those allergies in the record.

Note that all of this extra capability and power comes with the generic nature of the ETL subsystem. No extra data sourcing is required, just additional metadata. I encourage you to try as many permutations of these cross-referencing facts as possible during the Beta version. If you decide you don't want the extra facts, all you need to do is disable the associated metadata entries to stop them from loading. My experience is that users who see these extra facts show up in places they didn't expect them quickly begin to appreciate the longitudinal aspect of the data warehouse.

To take this experimental philosophy a step further, I recommend that you plan to experiment with your data as you move through the Beta phase of your implementation. As you create your sourcing queries, keep in mind the variability of rules that you encountered while analyzing the data. You've probably made many choices about how you plan to load your data into the warehouse, but none of those choices has to be considered locked in place at this point. As you move into the next chapters of generic logic, and in particular as you start defining your metadata entries for loading your data, feel free to experiment with multiple simultaneous or overlapping scenarios. Your extracted source data can be added to the warehouse in multiple ways simultaneously by adding additional metadata.

If you are still considering multiple possibilities for loading a source extract, define the metadata to load it every way you are considering. Querying the same data that have been loaded in multiple ways is usually the best path to making final decisions about loading that data. A best option tends to emerge quite quickly when users start querying and playing with the data. The only extra effort or cost with this experimentation is a little metadata. Remember, there won't be any extra coding required to load data in alternative configurations. In fact, it's entirely possible that you will end up with your data that you choose to permanently load into the warehouse in multiple ways precisely because the different users who want to use that data have their own preferences for how they want to see it. Rather than seeing it as a problem to be solved, view it as an additional service that you can offer. That's the power of a generic architecture. You are now ready to begin implementing your generic ETL architecture for loading all of these sourced configurations.

Chapter 9

Generalizing ETL Workflows

The ETL workflows that you developed for the Alpha version were completely dependent upon knowing the structure and content of the source datasets that you were loading. In Chapter 7, you finished filling in some of the pieces of the design that weren't included in the fast-moving Alpha version, including the definition and content of the Metadata dimension on which this Beta version will be based. The approach to analyzing your source data and modeling the needed metadata for data sourcing in Chapter 8 has removed that knowledge from our view by creating a standardized mapping for every piece of source data to its warehouse target location. Chapter 8 ended with the defining of the queries needed to actually extract data from the source, those queries being the last places in your ETL subsystem where knowledge of the sources will be needed in the workflows.

This chapter begins with the generalizing code that is wrapped around each of those source-specific selection queries to create a standard inbound data layout that is the same regardless of the structure, volume, or complexity of the sourced data. This allows the ETL workflows that will actually load the warehouse in production to be built completely generically. There will only be one set of ETL workflows built, and those flows will load all sourced data, now and into the future. This is possible because of the combination of the standardization of the inbound source data as well as the standardization of the warehouse database design pattern. You simply won't need the hundreds or thousands of ETL workflows associated with traditional data warehousing in the past.

Standardizing Source Data

In order to process all source data into the warehouse through a single set of generic workflows, you must convert your arriving source data into a standardized format that is independent of any source and can flow through the

generalized ETL processes without those processes needing to know anything about the original sources. The ETL jobs that accomplish this conversion are referred to as *Source Data Intake* (SDI) jobs, and the central core of each SDI job is one of the source selection queries you started defining at the end of Chapter 8. The conversion of source files and tables into a standardized intake structure is one of the key enablers of this generic warehouse architecture, and so understanding these columns is central to understanding how the ETL ultimately works in its generic form.

The standard intake dataset is made up of just 13 columns (see [Table 9.1](#)); the first 11 represent job and data control parameters that help drive the ETL process, while the last 2 represent the source columns and values from the data source table after having been unpivoted. In essence, a wide source table is converted into a deeper but standardized narrower table for processing (e.g., a source table with 1,000 rows of 30 columns will convert into an SDI table with 30,000 rows and 13 columns). Since the ETL process iteratively alternates between pivoted and unpivoted data, you'll often refer to snapshots of data in the pipeline as *wide* or *deep*. The actual source tables are *wide*, and the SDI tables are *deep*. The final dimension and fact results are *wide* but will go through several wide-to-deep unpivots, and deep-to-wide pivots, as they pass through this generic ETL architecture. This alternating between wide and deep staging views of the data is what allows for a high-level variation to occur within the data while still supporting a single generic processing architecture. These variations include the number of components needed to define natural keys, the number of bridge entries in the dimensionalization of a fact to a dimension, and even the number of dimensions against which a fact is dimensionalized. Because a pivot command works with whatever data it receives, the ETL workflows don't have to be programmed to allow for these expected variations in source data structure or content.

Dataset Controls

While the objective of standardized source data is to allow generic ETL to process all inbound data without any knowledge of the specifics of the source data, that doesn't mean nothing identifies where data came from is internalized. Ultimately, the definition of the source data determines how and where the data end up in the warehouse. You also still want to know where all of the data in your warehouse came from for tracing and audit purposes. The challenge is to provide this information to the ETL process so it can guide the processing while remaining generic enough to allow one set of ETL jobs to process data from any number of sources.

Four of the 13 standard intake columns serve this purpose: a job identifier, a physical dataset name, a logical dataset name, and a fact control break. The job identifier is a surrogate integer value, generated at run time, which serves as a unique value to identify the run of a source dataset into the ETL job stream.

Table 9.1 Standardized 13 SDI Source Columns

<i>SDI Column</i>	<i>Definition/Remarks</i>
SDI_JOB_ID	A surrogate integer sequence value that represents the instance of this source job running. It will eventually be used to populate the Job row in the Data feed dimension.
PHYSICAL_DATASET	Typically a single value shared by all of the data generated in the job. It identifies, in some way, the dataset that is being sourced (e.g., "Lab Results").
LOGICAL_DATASET	A logical dataset represents one way to view the sourced data as it will eventually map into the warehouse. All of the data in a logical dataset will map into the warehouse in the same way.
FACT_BREAK	A control for a specific fact within the physical-logical dataset, allowing for fact-specific final dimensionality (e.g., UOM).
ENTERPRISE_ID	A control value that is used to segment data in the data warehouse so completely independent subsets of warehouse data can be isolated. Its use supports multiple enterprises sharing the data warehouse and is typically set to zero for single-enterprise implementations.
DEFAULT_KEY_01	An optional column selected from among the source data, or else established by job parameter, that will serve as highest-order natural key for dimensional data for which a highest-order key has not otherwise been established in the metadata.
BREAK_ID	A surrogate sequence integer value (one per transaction) that identifies the logical transaction. As a surrogate, it is better if this value is absolutely unique; <i>but it is sufficient to have it be unique within the JOB_ID, PHYSICAL_DATASET, and LOGICAL_DATASET combination.</i> A row number from the sourcing query is often used to set this value.
SOURCE_TIMESTAMP	A date and time that will be used for this transaction to select appropriate dimensional rows, so each row selected will have been active at the time of the timestamp. The grain of this column need be no more than the grain selected for managing slowly-changing dimensional changes across the dimensions, typically the minute or second.
DATAFEED_BREAK	Source row column(s) that delineate a separate logical source to be tracked through the ETL. One value per row (requiring some concatenation of multiple source columns is used), usually resulting in small range of values across the intake table. This value results in more discrete Datafeed dimension entries to be created, but not more Facts.
ETL_TRANSACTION	The type of desired ETL processing to be carried out on this data, defaulting to <i>Add/Update</i> if not provided.
TARGET_STATE	The data state in which facts will be inserted into the Fact table, defaulting to <i>Active</i> if not provided.
SOURCE_COLUMN	The name of the source column in the source dataset, including any derived columns generated in the selection query.
SOURCE_VALUE	The value of the source column in the source dataset or derivation. This value is coalesced to " <i>~Null~</i> " if the value is null.

It eventually serves as the dimension identifier of the job row in the Datafeed dimension, anchoring all of the information loaded for the source data.

The other three columns serve to provide a three-tiered hierarchy of structure for defining source data to the warehouse. At the highest level, the physical dataset name identifies the source dataset to be loaded to the warehouse. It is a textual column that names the source and must be unique within the warehouse environment. No two source datasets should have the same physical dataset name. These control values are “invented” by the data analyst while analyzing the sources to define the needed metadata. While they are text columns, you should avoid the temptation to make these names too descriptive. These deep intake files will have tens, or hundreds, of millions of rows in them. A few extra characters on these control columns can result in an extra gigabyte of storage requirement that, in the worst case, prevent much of your processing from being carried out in your server’s memory. Keep the values here simple enough to be unique in the long term but don’t try to overdescribe your source databases and tables.

The Logical Dataset and Fact Break columns provide for the definition of specific aspects of the data in the physical dataset that don’t necessarily apply to all of the data coming from the physical dataset. These two levels of hierarchy can be used for any arbitrary distinction, but there are typical patterns. As described in the previous chapter, the logical dataset typically defines differences within the physical dataset that occur at the row level (e.g., transaction type), while the distinctions drawn by the Fact Break tend to occur at the column level within the logical datasets (e.g., transaction states). If no such distinctions exist or are relevant for a source dataset, these columns should be set to “~All~” indicating that any unique specification as to the handling of the data will be determined exclusively by the metadata.

Figure 9.1 illustrates a situation in which these three levels of control might be needed. A physical dataset named “Lab Results” is being defined as needing two logical datasets, one that includes just a lab test identifier, and the other that includes a lab test identifier with a reference range specified. This distinction likely occurs in the source dataset at the row level; with each row of the source table representing a different lab result, some of which are for lab tests that have a reference range and the others for lab tests that do not specify a reference

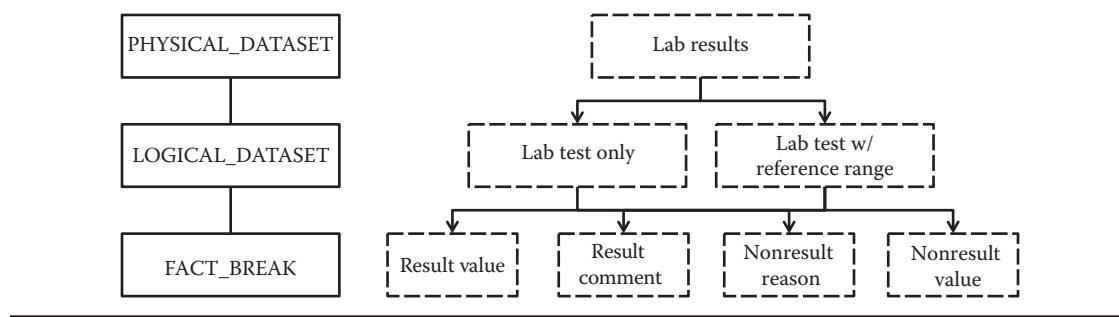


Figure 9.1 Example physical–logical–fact source controls.

range. Every row in the physical dataset will be processed in one of the two logical datasets or put another way; the physical dataset is said to have two logical datasets within it. In the parlance of traditional data processing, we might say that the file contains two different record types.

The distinction drawn at the level of the fact control break is different, and it applies to rows in both logical datasets. The main fact in this dataset is the Result Value, which will likely always be stored. There is also a Result Comment that will very likely only be stored when not NULL. The third fact is an example of something that might be uncovered by an analyst when analyzing the source data. In most lab result datasets in any clinical setting, there will often be a certain subset of the results that actually indicates that there are no results. An example might be where the specimen collected to do the test turned out to be contaminated. The reason for failure is usually present in the Result Comment column or some other explicit column provided in the source for that purpose. Even though these errors are typically received through the same dataset through which we receive the actual lab results, we typically choose to store them differently. If we didn't, we'd be leaving it to our users to learn to differentiate real from erroneous results, and undesired variability would be introduced into the perceptions and interpretations of the data by users approaching the data with different purposes.

This situation is handled by defining four different facts for the physical dataset, with all four being valid in either logical dataset. The intake metadata will be defined so only two facts will actually be stored for each arriving row. We'll either load a result value (required) with a comment (optional) or we'll store a nonresult reason (required) with a value (optional) if present. When users query the warehouse for test values, they'll see just the valid result values unless they specifically ask to include the nonresult entries in their queries. Likewise, users can also easily query just the nonresult entries without having to work with the valid results, unless they want the combination.

As a practical matter, these three levels of control allow us to specify where source data go in the warehouse, without having to define redundant entries in the multiple situations in which they occur. In our example, much of the data that can be defined at the physical dataset level because it remains true for all rows regardless of the distinctions drawn at the lower level. This is likely to include the definition of the patient, encounter, clinical service, ordering physician, performing laboratory technician, order date and time, result data and time, and nursing unit. These characteristics of the lab result don't change based on the conditions that are provided for at the Logical Dataset or Fact Break levels, so they need only be defined once for the physical dataset.

At the Logical Dataset level, we'll see the definition of the lab test as it needs to map to the warehouse's procedure dimension. One dataset will simply specify the lab test without a reference range, while the other will specify the test with a reference range. Once resolved to the dimension during ETL, it won't matter at the fact level which logical dataset was used for any particular result.

Final fact-specific metadata is defined at the level of the Fact Break. The Unit of Measure column in the source table is typically mapped to the two result value columns, while the two comment columns are typically defined implicitly in the narrative unit of measure since source tables only very rarely include an explicit unit of measure designation for comment or other text columns. Additionally, if your analysis determines that the various comment and reason values being loaded are in a particular language (e.g., English) that can be included explicitly in the unit of measure assignment.

Transaction Controls

Seven of the 13 standard intake columns serve the purpose of transaction controls: an Enterprise Identifier, a Break Identifier, a Source Timestamp, a Default (high-order) Key, a Datafeed Break, an ETL Transaction, and a Target State. These columns provide parameters needed by the generic ETL processes to properly control the completion of the transactions you determine needed during your source analysis. Not every control column is used in every transaction, but their presence provide you with the capability necessary to determine the processing flow of individual values through the ETL flows.

The Enterprise Identifier is a unique integer representing the enterprise loading data into the warehouse. This warehouse design supports consolidating the data for multiple enterprises into the database, with every value tied to an enterprise identifier. In this way, data across multiple identifiers never interacts, except for a few administrative users who are authorized to query the warehouse across enterprises. The typical warehouse user has authorization to see data in only one enterprise. If you are a single-enterprise organization with no intent to merge data from other institutions into your warehouse, then the enterprise identifier could either be eliminated from this design entirely or a default value of zero could be used as the only enterprise in the system.

The default high-order key, known as Default Key 01, is fairly simple to define, but more complicated to understand or appreciate. Whatever value is placed in this identifier will be used as the highest-order key in any multipart key (e.g., Context Key 01) for which a highest-order key source column is not otherwise provided. To illustrate, suppose you are sourcing data from a state-of-the-art EHR application that has been architected to support multiple hospitals in an aggregate hospital system. Each hospital using the application can configure its view of the system to include its own unique master data and metadata. The application uses a fairly standard Hospital Code in all of its data structures as a high-order key in order to keep the data for one hospital separate from the data of other hospitals that might otherwise use the same natural keys for their data. As a result, there is a Hospital Code source column in virtually every source dataset coming into the warehouse.

If you do *not* use the default identifier feature, you would simply source the Hospital Code column from the source into your source datasets like any other

source column. This will eventually create a performance problem that you might want to avoid. Imagine you're sourcing some historical lab test results, and there are 100,000,000 results to be loaded from a single hospital. That means that your deep intake file will have 100,000,000 rows for Hospital Code and they'll all have the same value. To make it worse, that file will eventually go through the metadata transformation step. The metadata will map the source data into the warehouse depending upon where each source column has been mapped. There might be a patient identifier, an encounter identifier, a lab test identifier, a unit of measure identifier, a lab technician identifier, a unit-room identifier, and an ordering physician identifier. That's seven natural keys in your dimensions, each of which needs the Hospital Code as the highest-order component. The 100,000,000 rows in the source table become 700,000,000 million rows after the metadata transformation join—all having the same value! If we instead place the Hospital Code in the default identifier of the sourced transaction, then NONE of those 100,000,000 source rows is needed, and they won't Cartesian into the 700,000,000 rows later. Instead, the standard ETL will pick up the Hospital Code (in the default high-order identifier) when the lookup of the dimension is not provided with a highest-order natural key. The use of the default high-order identifier feature is completely optional, but it saves a lot of memory, storage, and processing if used selectively.

The Break Identifier, or Break ID, uniquely identifies the transaction from the source dataset. Because the ETL takes the source data through a series of functional pipes as both wide pivoted and deep unpivoted datasets, you need to be able to rejoin the source transactional data that have gone through different pathways. The Break ID is the control value. It must be unique within the physical dataset value, and I recommend a surrogate integer that is unique across all datasets. If only unique within the dataset, the source table row number is often used for this purpose. It could be populated with a concatenation of source columns that serve as the natural key for the source transaction, but that requires a great deal more analysis, and lots of storage. The ETL never looks at the value of this column other than to join or pivot rows of reconverging source data.

The Source Timestamp identifies when the source data event or transaction occurred so the ETL will process the data as of that date. This is critical to the correct processing of slowly-changing dimensions in order to assure that dimension changes take place in the correct order, and that facts can always be pointed to the dimension definitions that would have been current on that date and time. Ideally, this column will be sourced from a functional source column taken directly from the clinical transaction being sourced. If no such functional column exists, then a system timestamp within the transaction might be used, although this introduces the risk of measurement error since that system timestamp might not exactly correspond to when the clinical event actually occurred.

The *grain* of the Source Timestamp ultimately determines the frequency with which slowly-changing dimensional variants are generated against the definitions in each dimension. For this reason, these timestamps are often truncated to the

minute for dimensional data since discrete seconds within system timestamps are not necessarily meaningful. For highly volatile dimensions, further truncation (e.g., 5, 10, or 15 minute; hourly or daily intervals) can further reduce dimension volatility. The deeper your grain, the more distinct versions of your dimension data will be created when volatility is high. The shallower your grain, the more your existing definition entries will be updated in place because new data arrive with the same timestamps against which the definitions were already loaded.

A shallow grain introduces some risk that you'll overlay some data with new versions that your analysis might have indicated should be multiversioned as a slowly-changing entry. To control this risk, use a deeper grain when in doubt and monitor actual volatility of the definitions in use. You'll include features in the Gamma version that will support undoing some of the high volatility stacks of definition versions that occur in some of your dimensions where you picked a deep grain that resulted in too many definition variants being generated during loads. This is a common problem that is easily fixed later; but loading data with too shallow a grain is extremely difficult to fix after loading because the source data often can't be reconstructed.

The Datafeed Break is optional and is useful in situations where some aspect of your source data that is otherwise not being loaded into the warehouse needs to be tracked for data administration purposes. An example might be a situation where a security-sensitive piece of information is being loaded, and it is desirable to know the computer terminal that was used to enter the data. Assuming the terminal identifier is available in the source and is not desired otherwise, it can be placed in the Datafeed Break during sourcing and a user of the warehouse will be able to report which data came from which terminal after the load. Using this feature increases the number of rows in the Datafeed dimension as each fact is tracked against each distinct value of the Datafeed break.

The ETL transaction determines what kind of processing is carried out by the generic ETL process. The options include the following:

1. *Add/update*: Data will be added to the warehouse if not already there and will be updated if found to be already present. This is the default if the ETL transaction column is left as Null. The vast majority of source data are taken into the warehouse using this transaction type.
2. *Update*: Data will be updated but will be discarded if not found. This type of transaction is useful for allowing less-trusted secondary sources to supplement data otherwise already in the warehouse without allowing that source to expand the scope or scale of the warehouse.
3. *Add*: Data will be added to the warehouse but will be discarded if already loaded. This type of transaction is useful for allowing a less-trusted secondary source to insert lesser-used information not otherwise available in the primary sources but only for as long as those primary sources don't supply the data.

4. *Initialize*: Data will be used to initialize Null columns of existing rows but will not result in new rows being inserted. This type of transaction is useful for allowing less-trusted secondary sources to provide values for data that might have been missing from primary sources.
5. *Delete*: The data will be logically deleted from the warehouse.

Other ETL transaction types will be added to the generic structure as needed to meet unique organizational requirements over time, and they would need to be coded into the ETL jobs in order to be implemented. Note that this column is not used by Beta version processing, instead presuming that the value is Add/Change for all data. Actual processing based on these values isn't included in ETL processing until the Gamma version.

The Target State control column provides the data state into which new facts will be inserted. Any valid entry from the Data State dimension is permitted. The default is the Active state if this column is left NULL.

Because the combined source dataset being fed into the generic ETL will typically have tens of millions of rows (possibly hundreds of millions of rows during history loads), to save storage, I prefer to leave the ETL Transaction and Target State columns as NULL when I want data added or updated into the active state. Only populating the exception values also makes them very easy to spot when using the staging tables during any testing or debugging activities since the vast majority of rows remain NULL in these columns, making the exceptions much more obvious. The actual performance improvement of leaving them NULL is dependent upon the efficiency with which your database management system handles NULL values.

Source Data Values

Finally, the SDI intake table needs to provide the actual data being received from the source. Two of the 13 standard intake columns serve this purpose: a Source Column and a Source Value. Each SDI row includes a single value for a single column of the source dataset. If the original source data was NULL, the Source value column will contain “~Null~” as a result of a standard coalesce in the source data selection logic.

Source Data Intake Jobs

The purpose of an SDI job is to convert one data source of arbitrary complexity into a single deep relational table of fixed columns that are completely independent of that source. Appending that generic dataset to the output of all other SDI jobs allows a single large data stream to be created that can be passed through all subsequent ETL jobs. Because the SDI job must be tailored to the specific data being sourced, there are many different SDI jobs needed in

the data warehouse architecture, one for each distinct source dataset needing to be sourced. The architecture of each SDI job is the same, but the specifics are different to the extent that the source datasets are different. An SDI job is the only job type in the ETL stream that is built uniquely for a source. All follow-on jobs are generic, being used to load any and all sources. The central core selection query for a source was introduced in Chapter 8. That source-specific query serves as the kernel of the fuller generic SDI job that drives the general extract of our generic datasets that are subsequently consolidated and driven through the ETL subsystem.

The source data to be processed by an SDI job is expected to be in some form of relational structure (Figure 9.2). If the actual arriving data are in some other form, such as a delimited text file, they must be converted into a relational table as input to this job. As introduced in Chapter 8, if data to be processed exist in a collection of interrelated relational tables, they must be joined to form a single intake table for this job. Unwanted columns in the source data may be excluded from the source relational view of the intake data. If you're unsure of a particular column, include it in the selection and the next job will drop it when it fails to join to any metadata entries. Only exclude columns if they are really unwanted in the long term.

Among all of the columns available in the intake table, some combination of columns will be identifiable as a combined identifier of a logical transaction within the source. This specification depends upon the combination of those columns being found in each row of the intake table, and those columns should collectively form a unique logical key to the transaction or event represented by the row. Typically, this combination of fields is unique at the row level of the inbound table. If not, adjustments will be needed to assure that your job selects the correct transaction break point for each sourced transaction. The Break ID is created in the output table to represent that break point. Less data space is used if a surrogate integer is generated instead of using the natural data. The most common Break ID assignment I've seen is the row number taken directly from the selection query. This value is typically unique within the physical dataset being selected, and so serves as an appropriate transactional break in the ETL.

The other control columns depicted in Table 9.1 are set by the SDI job as rows are selected from the source. Some of those columns are naturally extracted from the source data, while other columns need to be set in your code based on the coordination of the analysis of the source with the setup of the metadata that will

“CONTROL”	MRN	LNAME	FNAME	DOB	GENDER	ENTHNICITY				
2222222	009182719	Baker	Joanne	01/12/1968	F	C				
“CONTROL”	DEPT	PROFILE	TEST_ID	SPEC_TYPE	CAMPUS	BLDG	UNIT	O_DATE	P_DATE	RESULT
1111111	LB	0126	A546	Blood	001	0A4	East	2012-03-14	2012-03-16	1.2

Figure 9.2 Examples of two simple source data structures.



"CONTROL"	COLUMN	VALUE
1111111	DEPT	LB
1111111	PROFILE	0126
1111111	TEST_ID	A546
1111111	SPEC_TYPE	Blood
1111111	CAMPUS	001
1111111	BLDG	0A4
1111111	UNIT	East
1111111	O_DATE	2012-03-14
1111111	P_DATE	2012-03-16
1111111	RESULT	1.2
2222222	MRN	009182719
2222222	LNAME	Baker
2222222	FNAME	Joanne
2222222	DOB	01/12/1968
2222222	GENDER	F
2222222	ETHNICITY	C

"CONTROL"	DEPT	PROFILE	TEST_ID	SPEC_TYPE	CAMPUS	BLDG	UNIT	O_DATE	P_DATE	RESULT
1111111	LB	0126	A546	Blood	001	0A4	East	2012-03-14	2012-03-16	1.2
<hr/>										
"CONTROL"	MRN	LNAME	FNAME	DOB	GENDER	ETHNICITY				
2222222	009182719	Baker	Joenner	01/12/1968	F	C				

Figure 9.3 Unpivoted source data rows from two example sources.

be used to load data from the source. The widened intake source table, including the explicit generation of your needed control columns, is unpivoted, controlling on the 11 control columns just introduced, resulting in a deep table of the 11 control columns, plus Column Name and Column value columns (Figure 9.3). The depth of the table is the number of original rows times the number of available columns in the intake source table that have been selected for sourcing.

Note that the output of this job is the standard 13-column table that is common across all SDI jobs (see [Table 9.1](#)), so it can be appended to a common source dataset that will be passed into the metadata transformation job (see [Figure 9.4](#)). The amount of consolidated data that actually passes into the metadata transformation at any time is a performance issue to be addressed separately. For the purposes of your analysis and design, all of the data can be considered moving forward simultaneously from all sources into metadata transformation. It is not unusual for the consolidated dataset to include tens of millions of rows (possibly hundreds of millions during history loads).

SDI Design Pattern

Every SDI job will be somewhat unique to the extent that every source dataset accessed for the warehouse will be unique. The primary purpose of the SDI job architecture is to convert that source uniqueness into a standard set of columns so the rest of the ETL subsystem can be implemented generically and independent

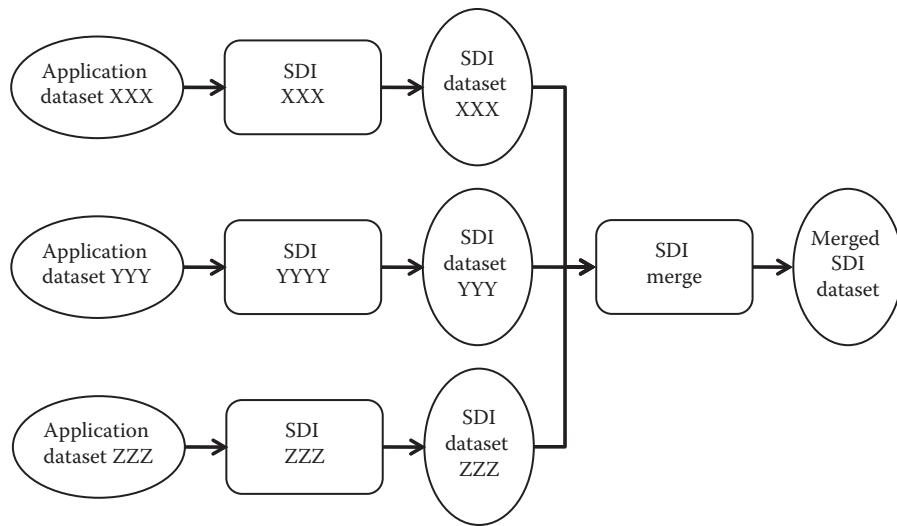


Figure 9.4 Source data intake control process.

of the structure or content of any particular data source. There is a standard design pattern to which all SDI jobs will adhere, and that makes their development and support much easier. The more complicated the sources, the more complicated the SDI jobs, but the overall structure and organization of every SDI job will remain quite stable within this design pattern. At its most basic, an SDI job is built around a query of the source table or dataset in order to obtain the desired source data:

```

SELECT PERSON_ID,
       ENCOUNTER_ID,
       ORDER_STATUS,
       PROVIDER_ID,
       ORDER_DATE
  FROM SOURCE_LAB_ORDERS;
  
```

Other than the most basic master data sources, most SDI jobs will obtain data from a view created by one or more join statements and filters:

```

SELECT O.PERSON_ID,
       O.ENCOUNTER_ID,
       O.ORDER_STATUS,
       O.PROVIDER_ID,
       O.ORDER_DATE,
       R.CLINICAL_RESULT,
       R.ABNORMALCY,
       R.LOW_REFERENCE_VALUE,
       R.HIGH_REFERENCE_VALUE,
       R.RESULT_COMMENT,
       R.RESULT_TIMESTAMP
  FROM SOURCE_LAB_RESULTS R
 INNER JOIN SOURCE_LAB_ORDERS O ON (R.ORDER_ID = O_ORDER_ID)
  
```

To set the stage for an SDI execution, a Master ID must be assigned for the source job. Any obsolete version of the SDI Master ID for the source is deleted, and a new single Master ID is assigned to the current execution:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS = 'SDI Lab Results'
INSERT INTO NEW_SURROGATES (PROCESS)
SELECT 'SDI Lab Results';
```

Together, the newly assigned Master ID for the source job and the query of the source data become the basis for the innermost subquery for the SDI job:

```
SELECT NS.NEW_SURROGATE AS SDI_JOB_ID,
       O.PERSON_ID,
       O.ENOUNTER_ID,
       O.ORDER_STATUS,
       O.PROVIDER_ID,
       O.ORDER_DATE,
       R.CLINICAL_RESULT,
       R.ABNORMALCY,
       R.LOW_REFERENCE_VALUE,
       R.HIGH_REFERENCE_VALUE,
       R.RESULT_COMMENT,
       R.RESULT_TIMESTAMP
  FROM SOURCE_LAB_RESULTS R
 INNER JOIN SOURCE_LAB_ORDERS O ON (R.ORDER_ID = O_ORDER_ID)
INNER JOIN NEW SURROGATE NS      ON (NS.PROCESS = 'SDI Lab Results')
```

To complete the sourcing subquery, a few more things need to be added: (1) each of the 13 common sourcing columns must be populated (even if NULL) based on the analysis of the source data that were used to define the metadata for loading the data, (2) any source column that might be NULL must be coalesced with a value (e.g., “~Null~”) to prevent the NULL value from moving into the ETL stream, and (3) data derived from the source data must be initialized:

```
SELECT NS.NEW_SURROGATE          AS SDI_JOB_ID,
       'Lab Results'           AS PHYSICAL_DATASET,
       CASE WHEN R.LOW_REFERENCE_VALUE IS NULL
             THEN 'No Range'
             ELSE 'Ref Range' END   AS LOGICAL_DATASET,
       CASE WHEN R.RESULT_VALUE = '**ERROR**'
             THEN 'Error'
             ELSE 'Result' END     AS FACT_BREAK,
       ROW_NUMBER()              AS BREAK_ID,
       NULL                      AS DATAFEED_BREAK,
       0                         AS ENTERPRISE_ID,
       NULL                      AS DEFAULT_KEY_01,
       R.RESULT_TIMESTAMP        AS SOURCE_TIMESTAMP,
       NULL                      AS ETL_TRANSACTION,
       NULL                      AS TARGET_STATE
```

```

COALESCE(O.PERSON_ID,                                '~Null~') AS PERSON_ID,
COALESCE(O.ENCOUNTER_ID,                            '~Null~') AS ENCOUNTER_ID,
COALESCE(O.ORDER_STATUS,                           '~Null~') AS ORDER_STATUS,
COALESCE(O.PROVIDER_ID,                            '~Null~') AS PROVIDER_ID,
COALESCE(O.ORDER_DATE,                             '~Null~') AS ORDER_DATE,
COALESCE(R.CLINICAL_RESULT,                        '~Null~') AS CLINICAL_RESULT,
COALESCE(R.ABNORMALCY,                            '~Null~') AS ABNORMALCY,
COALESCE(R.LOW_REFERENCE_VALUE,                   '~Null~') AS LOW_REF_VALUE,
COALESCE(R.HIGH_REFERENCE_VALUE,                  '~Null~') AS HIGH_REF_VALUE,
COALESCE(R.RESULT_COMMENT,                         '~Null~') AS RESULT_COMMENT,
COALESCE(R.RESULT_TIMESTAMP,                      '~Null~') AS RESULT_TIMESTAMP,
COALESCE(CONCAT(LOW_REFERENCE_VALUE,
    ISNULL(CONCAT(' - ',HIGH_REFERENCE_VALUE), '')),   '~Null~') AS REFERENCE_RANGE
FROM SOURCE_LAB_RESULTS R
INNER JOIN SOURCE_LAB_ORDERS O ON (R.ORDER_ID = O_ORDER_ID)
INNER JOIN NEW_SURROGATE NS      ON (NS.PROCESS = 'SDI Lab Results');

```

As like in many SDI jobs, the physical dataset is usually set to a constant that indicates the purpose of the overall SDI job. The logical dataset is often set conditionally on some aspect of the data being sourced. In this example, the logical dataset indicates whether the lab result included a reference range. This distinction allows the generic ETL to dimensionalize some of the data against a reference range, while allowing other data not to have such a range defined. The Fact Break is also set conditionally, indicating whether the source result was a valid or erroneous result. An erroneous result will not be dimensionalized against the unit of measure that would have been used for a valid result because the notation indicating that a result that has not been achieved will not be in the same unit of measure as a correct result value would have been.

This wide version of the source data must then be used as a subquery so the data can be unpivoted into the required 13-column format:

```

SELECT SDI_JOB_ID,
PHYSICAL_DATASET,
LOGICAL_DATASET,
BREAK_ID,
DATAFEED_BREAK,
ENTERPRISE_ID,
DEFAULT_KEY_01,
SOURCE_TIMESTAMP,
ETL_TRANSACTION,
TARGET_STATE,
SOURCE_COLUMN,
SOURCE_VALUE
FROM
(
SELECT
    NS.NEW_SURROGATE          AS SDI_JOB_ID,
    'Lab Results'              AS PHYSICAL_DATASET,
    CASE WHEN R.LOW_REFERENCE_VALUE IS NULL
        THEN 'No Range'
        ELSE 'Ref Range' END    AS LOGICAL_DATASET,
    CASE WHEN R.RESULT_VALUE = '*ERROR*' 

```

```

        THEN 'Error'
    ELSE 'Result' END
ROW_NUMBER()
NULL
0
NULL
R.RESULT_TIMESTAMP
NULL
NULL
COALESCE(O.PERSON_ID,
COALESCE(O.ENCOUNTER_ID,
COALESCE(O.ORDER_STATUS,
COALESCE(O.PROVIDER_ID,
COALESCE(O.ORDER_DATE,
COALESCE(R.CLINICAL_ESULT,
COALESCE(R.ABNORMALCY,
COALESCE(R.LOW_REFERENCE_VALUE,
COALESCE(R.HIGH_REFERENCE_VALUE,
COALESCE(R.RESULT_COMMENT,
COALESCE(R.RESULT_TIMESTAMP,
COALESCE(CONCAT(LOW_REFERENCE_VALUE,
ISNULL(CONCAT(' ',HIGH_REFERENCE_VALUE), '')),
'~Null~') AS REFERENCE_RANGE
FROM SOURCE_LAB_RESULTS R
INNER JOIN SOURCE_LAB_ORDERS O ON (R.ORDER_ID = O_ORDER_ID)
INNER JOIN NEW_SURROGATE NS ON (NS.PROCESS = 'SDI Lab Results')
) I
UNPIVOT
([SOURCE_VALUE] FOR SOURCE_COLUMN IN
(
PERSON_ID,
ENCOUNTER_ID,
ORDER_STATUS,
PROVIDER_ID,
ORDER_DATE,
CLINICAL_RESULT,
ABNORMACY
LOW_REF_VALUE,
HIGH_REF_VALUE,
RESULT_COMMENT,
RESULT_TIMESTAMP,
REFERENCE_RANGE
)
) U;

```

If desired, certain filters can be added to the sourcing query. In the case of the laboratory results, there is no value in extracting the reference range columns when they aren't used in the source, so those rows can be excluded at the time of sourcing:

```

WHERE LOGICAL_DATASET = 'Ref Range'
OR (LOGICAL_DATASET = 'No Range'
AND SOURCE_COLUMN NOT IN ('LOW_REF_VALUE','HIGH_REF_VALUE', 'REFERENCE_RANGE'));
```

This kind of filtering is completely optional since the data won't have any metadata defined for it; the sourced data rows will be dropped during the initial stage of ETL loading. Filtering them out is useful if their presence will cause

confusion or if the data volumes are great enough (e.g., tens or hundreds of millions of records) that performance improvements become predictable with their exclusion. Once the data have been selected and unpivoted, they can be inserted into a table with the standard 13-column configuration:

```
CREATE TABLE #SOURCE0 (
    SDI_JOB_ID           INT,
    PHYSICAL_DATASET     VARCHAR(64),
    LOGICAL_DATASET      VARCHAR(64),
    BREAK_ID              INT,
    DATAFEED_BREAK       VARCHAR(64),
    ENTERPRISE_ID         VARCHAR(64),
    DEFAULT_KEY_01        VARCHAR(64),
    SOURCE_TIMESTAMP      DATETIME,
    SOURCE_COLUMN          VARCHAR(64),
    SOURCE_VALUE           VARCHAR(2000)
);
```

All SDI jobs will follow this basic design pattern. Any complexity introduced into these jobs will be a result of the analysis of complex data sources. When the extractions become very complex, I recommend breaking them up into sourcing jobs with narrower scope. It takes some practice, but your team will quickly become comfortable with the basic design pattern, and adding new sources will end up taking only a few minutes each.

Source Data Consolidation

Once all of the SDI jobs have been built and tested, you can merge the outputs of all of their executions into a single large-scale dataset for sending that data into all subsequent ETL jobs (Figure 9.4). There will be many reasons why you'll ultimately choose to send a variety of subsets of your consolidated source data into the ETL jobs during any particular execution, but any logical distinctions among the data structures will not be among those reasons. You might choose to keep certain datafeeds separate for timing or availability issues, or because you perceive certain performance constraints as limiting your ability to load all data during a single ETL execution. Performance often becomes a concern during historical loading where extremely large datasets are sometimes handled requiring hours, or even days, to load. Under these conditions, you'll want control over which datasets are loading at any time. Over the long term, however, keep in mind that your daily loads to the warehouse will be small relative to those historical loads, and you'll typically get your best database performance if you send all of your data through at once in order to maximize your physical vs. logical performance of your joins and bulk loads.

External versus Internal Sourcing

This chapter has dealt with the SDI design pattern needed to source external data into the data warehouse. There will be situations where you'll also want to create source data that, at least in part, originates within the data warehouse itself. In these internally driven situations, you'll have access to internal surrogate keys that ordinarily would be hidden from views of external data sources. Figure 9.5 shows the five ways that data identifiers can be sourced for a dimension. The Natural Keys are the most traditional pathway and the only one that doesn't require knowledge of the internals and logic of the warehouse design. Each of the other four requires knowledge of the warehouse structure and content to be encoded in the SDI job. Ordinarily, I advise against using such knowledge, and for many years, I resisted adding these extra pathways to the generic ETL design; but I eventually gave in because over time I saw potential uses and performance improvements from these extra pathways, the value of which I could no longer deny, particularly as the healthcare sector has shifted toward generating numerous key metrics out of the data warehouse environment.

The generalized ETL workflows developed in the next chapter will only deal with the externally sourced keys. Sourcing a Reference Composite instead of the multiple natural keys requires the developer of an SDI job to know the target structure for a dimension context, but that knowledge is usually known as a result of the analysis that went into the design of the SDI job itself. The risk is minimal if reasonably and rigorously implemented, and the payback from having fewer compound keys to pivot can be significant when loading history into this

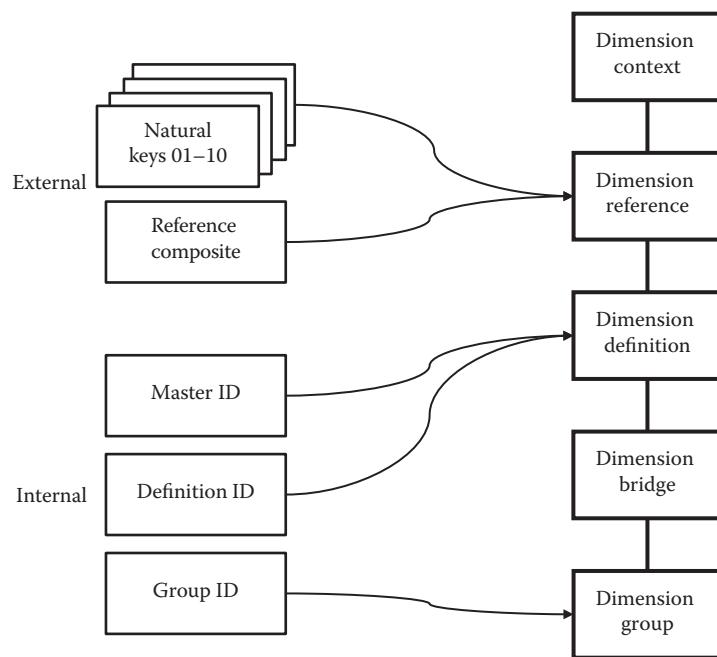


Figure 9.5 Sourcing dimension identifiers.

Beta version, a process that often loads hundreds of millions of rows. I typically defer the internally sourced keys to the Gamma version because the need for them won't arise until there's enough data to warrant sourcing new data from the warehouse itself.

This section has also only dealt with data that populates into the warehouse in standard user fields, primarily the descriptive columns of the definition tables and facts. It has not dealt with the sourcing of control data for the warehouse itself; data that exist in the metadata dimension to control the flow of ETL processing, primarily affecting the Bridge and Hierarchy tables. The Role and Rank that will be populated in the Bridge table and the Perspective and Relation that will be populated in the Hierarchy table are only defined in the metadata dimension in the Beta version. In the Gamma version, you'll add the ability to source the data to the SDI framework, which will make certain source datasets easier to handle because there will be fewer rows of Metadata needed.

The scope boundary between Beta and Gamma versions is arbitrary. My criterion for leaving certain functions and features for the Gamma version is historical. I leave in the Beta version the essential functions that all of my clients have needed in the past to get their warehouse up and running, and I defer to the Gamma version any features that are typically not implemented by my clients at all or that are only implemented after gaining experience with the Beta version. Your implementation will be driven by your requirements, so you might choose to implement certain features from my Gamma version in your Beta version. Nothing I've written in the following chapters should be interpreted as precluding that move.

Single Point of Function

My underlying goal in suggesting that a single set of generic ETL workflows can be used to load the warehouse is to isolate every piece of function to be performed within the ETL subsystem in a single place. Ideally, there will be one, and *only* one, place within the ETL system where any particular function is carried out. In this manner, making a change to the functionality of the ETL subsystem will require making the change in only the single place where the function needing change is performed and, more importantly, adding new functionality to the ETL system will entail adding new functionality in only one place. This functional isolation is the key to very fast development and loading of the warehouse, and it is the critical basis for separating Beta version functionality from Gamma version functionality. It also allows for future investment in modifying and extending the data warehouse in the future after it is placed into production and becomes operational.

Historically, I haven't seen ETL systems developed in this way. The traditional approach has always involved creating new ETL flows for new data sources by somehow cloning the piece of the existing ETL system that is most like the

desired new flow and then customizing that new flow to meet any new or distinctive requirements of the new data source. The development and evolution of ETL toolsets has largely been about enabling this cloning and reuse. Data warehouse implementations end up with hundreds or even thousands of ETL workflows, each some variant of relatively few archetypes. A problem with this approach is that each new workflow adds to the mass of material that needs to be supported by the warehouse team and that would be impacted by any changes or additions to intended functionality.

What I've seen happen in these situations is that certain changes that are desired in the data warehouse end up not being implemented because the number of ETL workflows that would be impacted by the change is simply too large. The development culture based on cloning existing components has resulted in an environment where individual functional logic occurs in far too many components to be manageable. Organizations forgo new functionality because of that impact. Alternatively, organizations that can anticipate these problems before they occur try to build everything they think they'd ever want into their first release. If the function can be built early enough, the reasoning goes, then that function will already be present when the cloning process begins. The problem is that this approach unduly burdens the initial warehouse development with low-priority or future-oriented requirements, and the initial development project fails under that heavy burden.

In the case of the design I am presenting in this book, it would force me to implement all of the functionality this book now defers to the Gamma version during the Beta version development so those Gamma version features would be present early in the cloning process. The distinction between the Beta and Gamma versions would be lost, and the fast Beta implementation would be sacrificed under an argument of completeness and change control. While I value all of the Gamma version features that we'll eventually implement, I don't want to delay the Beta version implementation to worry about these things, and I'm not sure that the user community that I need support from to help prioritize those functions could possibly know enough to do so without first having access to the Beta version. Doing a combined Beta–Gamma version could be an order of magnitude more complex than doing a Beta version followed by a separate Gamma version implementation. Even that approach would still have the problem of future changes and additions that my Gamma version doesn't anticipate. Eventually, any organization will have to stop investing in warehouse expansion because of the burdens imposed by the massiveness of the warehouse configuration.

We can avoid these negative outcomes, and reap many desirable benefits, by avoiding the paradigm of repeatedly cloning successful past ETL components to create new ETL components that result in the massiveness of traditional ETL subsystems. We want to have an ETL subsystem that is as lean as possible and in which every function or feature has a highly predictable and consistent placement. Instead of trying to anticipate and satisfy all future

requirements in the beginning, you'll start out by only satisfying existing short-term requirements but using a system architecture into which the placement of any new function or feature would be at single point in that architecture. The individual components of the ETL subsystem will be more complex than their traditional counterparts, but it won't be because the work they do is any harder. It will be because they have an extra layer of abstraction that traditional ETL code didn't need.

The process is actually fairly simple. Imagine the way the cloning approach to ETL development has always worked. Confronted with the need to create a new ETL workflow, we chose the existing flow that was most like the one we wanted. After cloning the code, we made whatever changes were needed to make the code work for our new set of requirements. Perhaps we altered the table that the ETL was targeting or changed the columns that were being updated. If you think through all of the changes that might have occurred to create those hundreds or thousands of flows in some traditional systems, you'll start to realize that the actual number of specific changes made in cloned code is not that large. Most of our thousands of cloned ETL flows actually only differ by about 20 (usually hard-coded) parameter values. I've moved those values into the metadata dimension of this design, and I've referenced those values in my code. As a result, a single generic flow that might have been cloned a hundred times in a traditional effort simply remains a single generic flow. That flow is more abstract—having variable names where hard-coded literals used to be—but new data can now be processed by adding and updating values in the metadata dimension rather than cloning, refining, testing, and installing new ETL flows. If requirements for how that flow is to perform expand in the future, the generic code is changed once rather than having to make the same effective change to the hundred flows that were cloned from the original. Making the same change in hundreds or thousands of places introduces extensive configuration and regression risk to any project, and ultimately results in desired changes rarely being made at all.

Single Point of Failure

Some would argue that creating single points of function actually creates single points of failure, something to be avoided at all cost in a major systems effort. While I agree that each function in this generic model is a single potential point of failure, I would also argue that the extensive extra testing and support that these functions receive dramatically improves their operational quality and reduces the chances of any failures occurring. Likewise, the traditional model involving hundreds of flows is also subject to many points of failure that can be just as critical as the single-point failure potential in a generic design. While not every ETL flow in a traditional system is a critical component, many functions in those designs are. Given that less per-function testing occurs in the traditional model, I would suggest that the

opportunities for the traditional model to fail actually far exceed those in the generic model in terms of probability and expected total loss. Since all data in this generic architecture flow through the same generic processes, those processes are subject to constant review and testing in ways that would be cost prohibitive in the traditional development approach.

ETL “Pipes”

Unlike in the Alpha version, where distinct code was developed for every step of the ETL process from every source dataset, in the Beta version, we want to develop generalized processes that can be used to load any data from any source without any need to develop new ETL code when new sources are uncovered. Even in its generic form, the ETL subsystem will still be a very complex undertaking. To simplify the discussion, I use the metaphor of pipes to discuss component flows through the ETL subsystems, each serving a purpose, sometimes splitting and sometimes merging. What arrives in the single inbound source flow eventually leaves as data in the data warehouse. In between those points are a number of distinct flows that I call “pipes.”

Figure 9.6 illustrates the ETL pipes that will constitute the Beta version. At the highest level, the source data arriving at the ETL subsystem consists of three types of data that each go down a different pipe: source system keys that are destined for the Reference tables, descriptive data that are destined for the Definition tables, and the actual measurements and observations that are

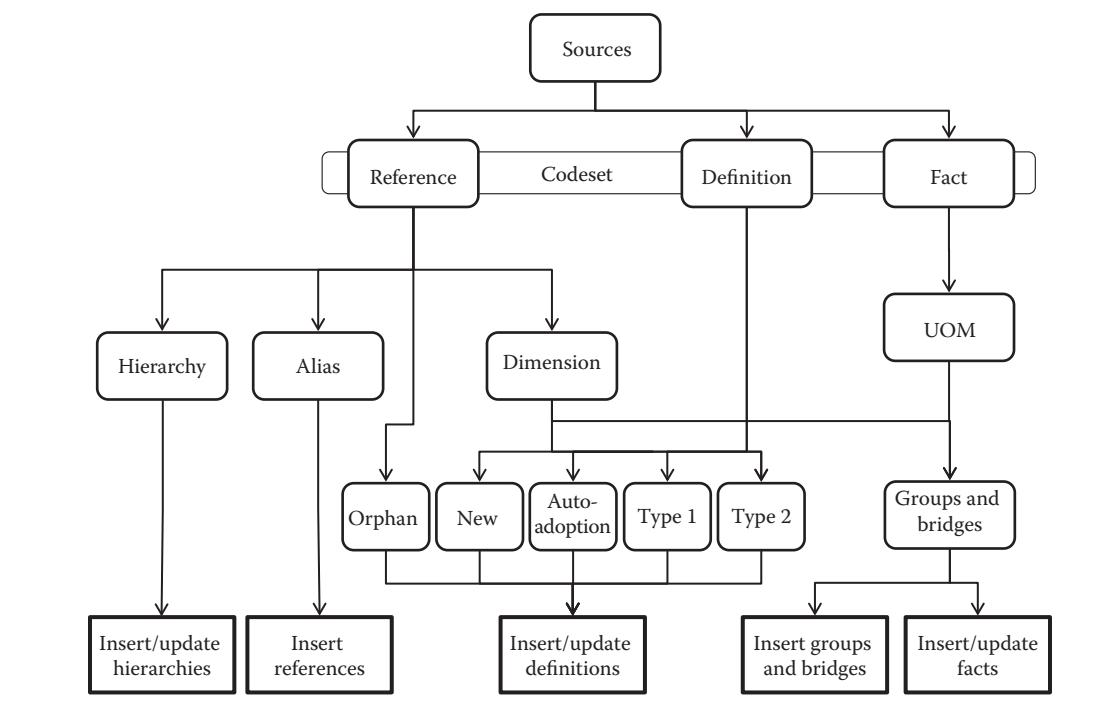


Figure 9.6 ETL subsystem pipes (Beta version).

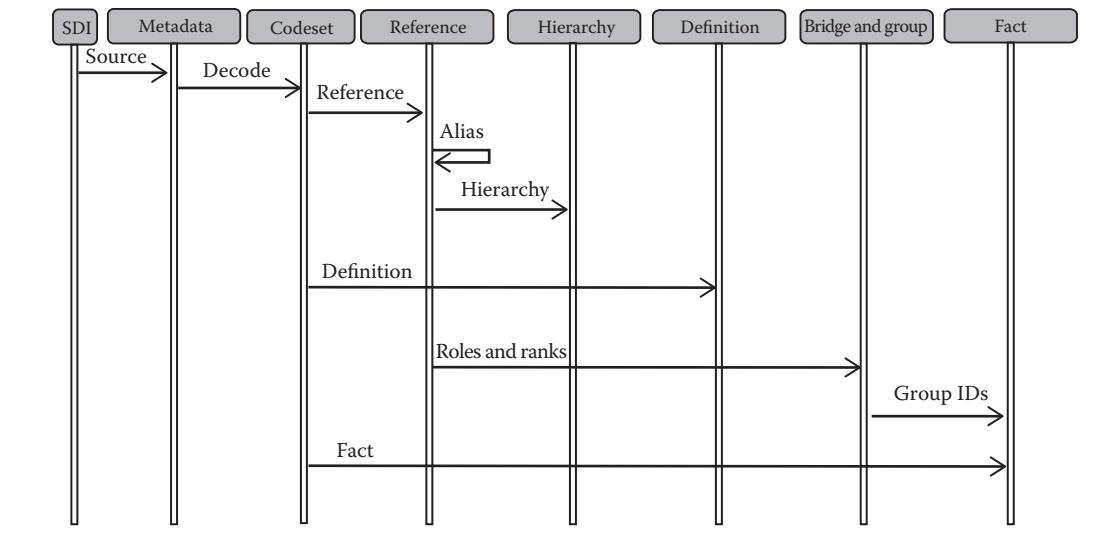


Figure 9.7 Simplified sequence diagram for ETL subsystem.

destined for the Fact tables. Data in all three of these main pipes are subject to the same level of codeset translation as processing is carried out. Although the reasons why different kinds of coded values would need translation into the warehouse will vary, the actual mechanism for accomplishing the translation will be the same across the three main pipes. A high-level view of the three main pipes in the ETL subsystem is depicted in Figure 9.7.

Reference Pipe

The Reference pipe takes the natural and system keys that occur in source system datasets and converts them into the surrogate keys that have meaning only within the warehouse. These surrogates have no meaning outside of the warehouse, and the source keys are never used to directly identify data in the warehouse. This functional separation is what enables the data warehouse to contain a variety of source data from a variety of systems and institutions. The Context table in each dimension provides for a distinct definition for every source key that will be received. Since every Reference table entry is associated with a single context entry, there is no risk that two different source keys that happen to look the same would be accidentally used to incorrectly identify the same entry.

Data that flow through the Reference pipe will eventually end up in one of three other pipes depending upon its purpose: the Definition pipe, if the source system keys are being used to build a master dimension row or to dimensionalize some facts; the Hierarchy pipe, if two source system keys are being defined in a flat hierarchy relationship; or the Alias pipe, if one source system key is being asserted to be an alias for another source system key. The Hierarchy and Alias pipes can be considered as two smaller pipes coming off the main Reference pipe in Figure 9.7.

The Reference pipe executes first in the ETL subsystem because the surrogates it creates as its output are needed in both the main Definition and Fact pipes. To support this functional dependency while still allowing for a maximum of parallel processing during the ETL jobs, surrogate keys are resolved or created very early in the Reference pipe processing so the necessary dependency for conducting Definition and Fact processing in parallel can be achieved. Because the database design does not include referential integrity enforcement on foreign keys, the actual database inserts and updates in the Reference pipe can continue in parallel after the Definition and Fact processes have initiated. Your objective in defining a parallel ETL architecture should be to get your Fact processing active as early as possible since it will typically carry the heaviest load during most historical and prospective loads. Your Definition loads will exhibit high variability in load volume over time but will usually be heavier than your Reference load, particularly when loading extensive history datasets.

Definition Pipe

The Definition pipe takes the data that define the dimension entries for the warehouse and creates, updates, or extends those dimension entries. This includes the standard “stub” descriptions associated with all dimensions, as well as dimension-specific columns that you add to your own implementation of the dimensions. In the Gamma version, it will also include a collection of control variables that helps drive data quality and access control functions.

Data that flows through the Definition pipe will end up in one of four smaller pipes depending upon its purpose: the Insert pipe, if the dimension data are new; the Auto-Adopt pipe, if the dimension data are new but a previous fact required the creation of an orphan row for the source system key; the Update pipe if the data are a time-independent update that stays with the definition row over time; or the slowly-changing Dimension pipe, if the data are a time-dependent update that is specific in time. An example of a Type 1 definition change might be a Date of Birth column in Subject, different versions of which would not be tracked because a person can only have one date of birth. An example of an update definition change might include a Date of Birth column in Subject since it wouldn’t make logical sense to track different values of these data at different points in time, where an example of a slowly-changing definition change might be a Last Name or Ethnicity column, where you might want to keep track of different versions of these data over time.

The Definition pipe is dependent upon the Reference pipe for access to the surrogate keys associated with dimension entries, so it executes after the surrogate keys have been obtained or created in the Reference pipe. There are no dependencies between the Definition and Fact pipes, so they are expected to execute in parallel.

Fact Pipe

The Fact pipe prepares and loads all of the facts that are targeted to any of the Fact tables in the design. Data that flow through the Fact pipe get processed in a variety of ways before they end up in the warehouse. After being merged with the appropriate Reference data for the dimensions, a series of actions takes place to round out the dimensionalization of the facts in order to include any that weren't provided by the source data. Paramount among these steps is the cleanup of the necessary unit of measure data and the building of the Groups for multibridged references.

The Fact pipe is dependent upon the Reference pipe for access to the surrogate keys associated with dimension entries, so it executes after the Reference pipe. However, it is not dependent upon the Definition pipe. Traditionally, we've thought of the facts as being so dependent upon the dimension data that we've always waited until our dimension loads were complete before starting our fact loads. That isn't required in this generic architecture. It's fine if the Fact loader has created dimension Bridges and Groups before the Definition pipe has created the associated Definition entries, as long as they're all properly loaded when all the jobs are complete. This independence can be important for scheduling, precisely because the Fact load is typically the largest and longest running of the ETL components. Starting them sooner, by not waiting for the Definition and Reference pipes to complete, helps minimize load times.

Checkpoint, Restart, and Bulk Loading

The generic ETL architecture described here, and explored in much more detail in the next three chapters, can be implemented in a variety of ways depending upon the technical platforms and architectural requirements of each implementation. No two implementations that I have been involved with have been the same, and yours won't match any that I've done either. Each implementation is specific to the time, place, and requirements of the organization implementing the warehouse. Because of these differences, the code fragments included in these chapters will need to be adapted to your specific requirements. They should be treated as pseudo-code, illustrating the concepts being presented rather than being a definitive implementation.

There are some design patterns in these implementations that are worth considering as high-level architectural guidelines however. First, the three pipes are intended to execute in parallel to the extent that functional dependencies can be avoided. The strongest dependency in the entire ETL subsystem is the need to generate new dimension Master IDs early in the Reference pipe in order to make them available to the Definition and Fact pipes. Avoiding referential integrity constraints in the database removes any downstream dependencies among the three pipes involving those Master IDs.

Second, each pipe begins by generating and manipulating data in staging tables that are empty at the start of processing. Committing data to the database is always referred to as late a point in the processing as possible. Performing database inserts and updates late means that system restart is simplified if any failures occur prior to beginning database updates. No system or database recovery is needed if system exceptions occur before database activities begin. Simply rescheduling the ETL stream to process the same input data provides for simple restart. The ETL jobs purge the staging tables and reprocess normally. Additionally, late database activity also offers greater opportunity to use your database management system's bulk loader capability since functional ETL logic has been separated from the database updates.

How you will implement these architectural design options will vary based on your requirements and technical situation. I don't pursue them here because I'm not a system architect, and I find trying to keep up with the constant technical evolution in this discipline precludes being able to offer any kind of definitive approach here. To the extent possible, the code samples in the next three chapters are flexible enough to be implemented under whatever system configuration you have available at the time of your implementation.

Metadata Transformation

All three of the main ETL pipes are dependent upon a first step of the ETL subsystem where the source data to be loaded are joined to the warehouse metadata dimension to obtain a full mapping of the source data through to its final target in the data warehouse (Figure 9.8). The result of the Metadata join is the full set of transformations that need to be carried out by the various ETL pipes in order to complete the loading of the arriving source data. Each transformation row takes the value of a single source column and provides the processing parameters needed to get that value into a single target column somewhere in the warehouse.

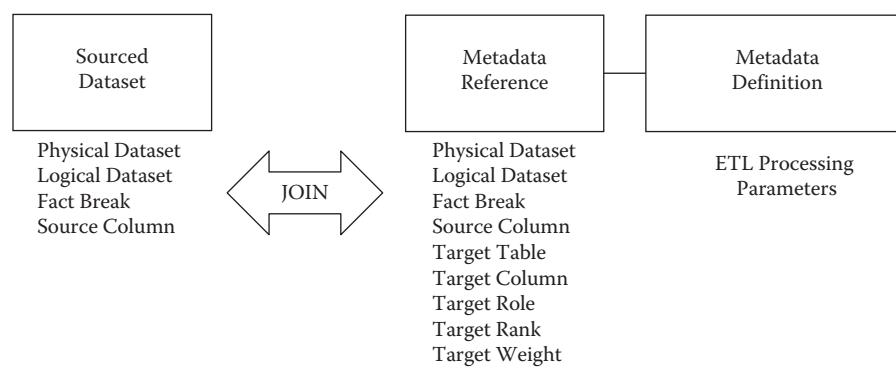


Figure 9.8 Joining source data to metadata.

The Metadata transformation join involves four of the columns created by the SDI process, including the Physical and Logical Dataset names, any Fact Break parameter, and the name of the Source Column. These four values correspond to the first four context identifiers in the Metadata Reference table. These four values must match between source and metadata, or else one of them should contain the “~All~” value that will cause it to match to any row. The INNER JOIN must also ensure that the joined row of metadata is active by looking at the Status Indicator column of the Metadata Definition table row:

```
SELECT *
FROM STAGING.SOURCE S
INNER JOIN METADATA_R R
    ON S.PHYSICAL_DATASET = R.CONTEXT_KEY_01
    AND ( S.LOGICAL_DATASET = R.CONTEXT_KEY_02
        OR S.LOGICAL_DATASET = '~All~'
        OR R.CONTEXT_KEY_02 = '~All~' )
    AND ( S.FACT_BREAK = R.CONTEXT_KEY_03
        OR S.FACT_BREAK = '~All~'
        OR R.CONTEXT_KEY_03 = '~All~' )
    AND S.SOURCE_COLUMN = R.CONTEXT_KEY_04
INNER JOIN METADATA_D D
    ON R.MASTER_ID = D.MASTER_ID
    AND D.STATUS_INDICATOR = 'A';
```

The number of transformed rows will usually not be the same as the number of source rows. If a source value has been mapped to multiple locations in the warehouse, a single source row will result in multiple transformed rows, one for each target. If a source value has not been mapped into the warehouse, it will be dropped by the transformation join. This creates some independence between the analysts who are mapping data into the new warehouse and the analysts who are working to source data. Source data are typically extracted for all available data in a source even if the need for that data haven't been confirmed. If not needed, the transformation join will drop the extra rows. If metadata is created for those values in the future, they'll start loading as soon as the metadata is modified, with no changes required to the source intake jobs. In fact, if historical source data are being archived, new metadata can be used to back-load the historical values that had been previously dropped.

Codeset Translations

The join to the metadata dimension has converted our source data mappings into target data mappings for our generic load. Before we continue, there's another function we want to build into that metadata transformation: codeset translation. The actual value supplied by the source might not be the value we want to load into the warehouse because it is system specific to the source. The most typical use of this codeset processing will be later in the Definition

pipe where coded values are converted to their descriptive values for inclusion in the definition rows, but the processing applies in the Reference and Fact pipes, as well:

```

SELECT *
FROM STAGING.SOURCE S
INNER JOIN METADATA_R R
    ON S.PHYSICAL_DATASET = R.CONTEXT_KEY_01
    AND ( S.LOGICAL_DATASET = R.CONTEXT_KEY_02
        OR S.LOGICAL_DATASET = '~All~'
        OR R.CONTEXT_KEY_02 = '~All~' )
    AND ( S.FACT_BREAK = R.CONTEXT_KEY_03
        OR S.FACT_BREAK = '~All~'
        OR R.CONTEXT_KEY_03 = '~All~' )
    AND S.SOURCE_COLUMN = R.CONTEXT_KEY_04
INNER JOIN METADATA_D D
    ON R.MASTER_ID = D.MASTER_ID
    AND D.STATUS_INDICATOR = 'A'
LEFT JOIN CODESET_R CR
    INNER JOIN CODESET_D CD
        ON CR.MASTER_ID = CD.MASTER_ID
        AND S.SOURCE_TIMESTAMP
        BETWEEN CD.EFFECTIVE_TIMESTAMP AND CD.EXPIRATION_TIMESTAMP
        ON CR.CONTEXT_KEY_01 = D.CODESET_CONTEXT_KEY_01
        AND CR.CONTEXT_KEY_02 = D.CODESET_CONTEXT_KEY_02
        AND CR.CONTEXT_KEY_03 = S.VALUE

```

With the extra join in place, the selection of the value column from the source versus codeset becomes conditional on whether the LEFT JOIN successfully found a match for the sourced value:

```
COALESCE(CD.DESCRIPTION, S.VALUE) AS VALUE
```

Additionally, the metadata dimension provides for the selection of the specific column from the codeset that will be used in place of the sourced value:

```

CASE WHEN D.CODESET_COLUMN = 'DESCRIPTION'
    THEN COALESCE(CD.DESCRIPTION, S.VALUE)
ELSE WHEN D.CODESET_COLUMN = 'FULL_DESCRIPTION'
    THEN COALESCE(CD.FULL_DESCRIPTION, S.VALUE)
ELSE WHEN D.CODESET_COLUMN = 'EXTENDED_DESCRIPTION'
    THEN COALESCE(CD.EXTENDED_DESCRIPTION, S.VALUE)
ELSE WHEN D.CODESET_COLUMN = 'ABBREVIATION'
    THEN COALESCE(CD.ABBREVIATION, S.VALUE)
ELSE WHEN D.CODESET_COLUMN = 'VERNACULAR_CODE'
    THEN COALESCE(CD.VERNACULAR_CODE, S.VALUE)
ELSE COALESCE(CD.DESCRIPTION, S.VALUE) END AS VALUE

```

Specifying the exact column that should be used as a substitute for the sourced value provides flexibility for using different target values in different circumstances. For example, a source value for state might be received as “Fla.” from a source but, through codeset translation involving the Description column, might become “Florida” in a warehouse definition entry. The same source value might be translated through the codeset Abbreviation column as “FL” in a Reference table Context Key. Additional columns can be added to this substitution logic at any time as long as any specific column name placed in the metadata has been included in the preceding case statement in the ETL flow.

The LEFT JOIN to the codeset dimension causes this extra set of joins to have no effect if there’s either no codeset translation defined for the source value in the metadata or the actual value received from the source system doesn’t match a value in the codeset. If the codeset is defined and the value matches, one of the columns from the codeset is used as an alternative value. Note that the sample case statement defaults the substitution to the codeset Description column if the Codeset Column property doesn’t include one of the values in the case statement (or is NULL).

General versus Functional Transformations

Joining source data to metadata can be done logically as a single step for all data pipes, but I don’t recommend it. The data that flows through the Reference, Definition, and Fact ETL pipes is very similar, but it’s not the same. There are distinctive aspects to the metadata join that are unique to each pipe, and a single transformation join would necessitate persisting the data to a staging table to make it available to those subsequent queries (Figure 99). Also, at no point in the ETL process is the data in these distinct pipes needed in the same pipe, making the persistence of the generally transformed source data both temporary and wasteful. If a single generalized metadata transformation query is chosen, it will look something like this:

```
SELECT S.JOB_ID,
       S.PHYSICAL_DATASET,
       S.LOGICAL_DATASET,
       S.FACT_BREAK
      ,S.ENTERPRISE_ID,
       S.DEFAULT_KEY_01,
       S.BREAK_ID,
       S.SOURCE_TIMESTAMP,
       S.DATAFEED_BREAK,
       S.ETL_TRANSACTION,
       S.TARGET_STATE,
CASE WHEN D.CODESET_COLUMN = 'DESCRIPTION'
      THEN COALESCE(CD.DESCRIPTION, S.VALUE)
```

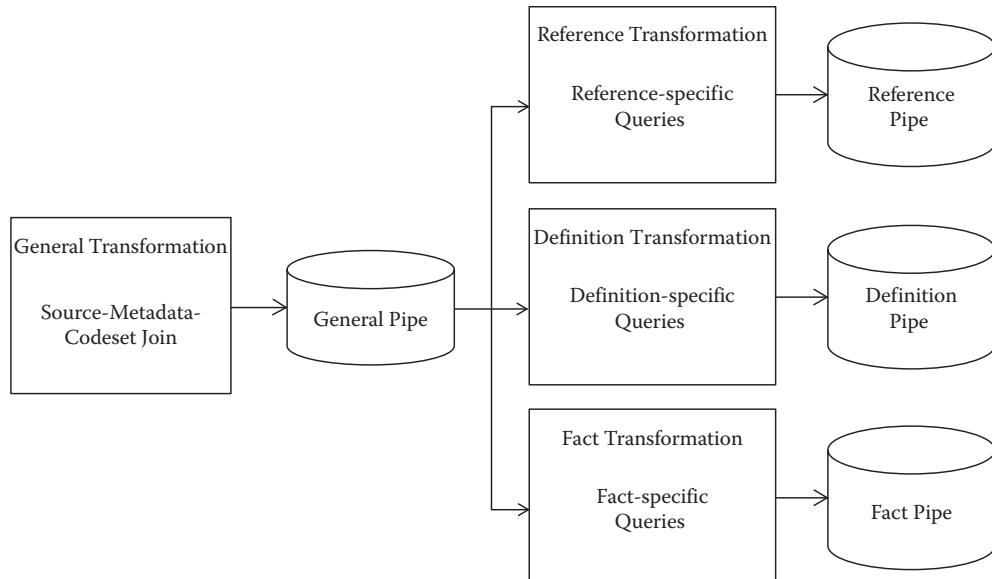


Figure 9.9 Single general metadata transformation.

```

ELSE      WHEN D.CODESET_COLUMN = 'FULL_DESCRIPTION'
          THEN COALESCE(CD.FULL_DESCRIPTION, S.VALUE)
ELSE      WHEN D.CODESET_COLUMN = 'EXTENDED_DESCRIPTION'
          THEN COALESCE(CD. EXTENDED_DESCRIPTION, S.VALUE)
ELSE      WHEN D.CODESET_COLUMN = 'ABBREVIATION'
          THEN COALESCE(CD.ABBREVIATION, S.VALUE)
ELSE      WHEN D.CODESET_COLUMN = 'VERNACULAR_CODE'
          THEN COALESCE(CD.VERNACULAR_CODE, S.VALUE)
ELSE      COALESCE(CD.DESCRIPTION, S.VALUE) END      AS VALUE,
R.MASTER_ID AS METADATA_MASTER_ID,
D.DATASET_TYPE,
D.TARGET_DIMENSION,           -- Reference & Definition Only
D.DIMENSION_CONTEXT,          -- Reference & Definition Only
D.TARGET_TYPE,                -- Reference Only
D.TARGET_SUBDIMENSION,        -- Reference & Definition Only
R.CONTEXT_KEY_04 [TARGET_TABLE],
R.CONTEXT_KEY_05 [TARGET_COLUMN],
R.CONTEXT_KEY_07 [BRIDGE_ROLE],   -- Reference Only
R.CONTEXT_KEY_08 [BRIDGE_RANK],   -- Reference Only
R.CONTEXT_KEY_09 [BRIDGE_WEIGHT]  -- Reference Only
D.IMPLICIT_UNIT,              -- Fact only
D.IMPLICIT_MEASURE,            -- Fact only
D.IMPLICIT_LANGUAGE           -- Fact only
FROM STAGING.SOURCE S
INNER JOIN METADATA_R R
    ON S.PHYSICAL_DATASET  = R.CONTEXT_KEY_01
    AND ( S.LOGICAL_DATASET = R.CONTEXT_KEY_02
          OR S.LOGICAL_DATASET = '~All~'
          OR R.CONTEXT_KEY_02  = '~All~' )
    AND ( SFACT_BREAK     = R.CONTEXT_KEY_03
          OR S.FACT_BREAK    = '~All~' )

```

```

        OR R.CONTEXT_KEY_03 = '~All~' )
        AND S.SOURCE_COLUMN      = R.CONTEXT_KEY_04
INNER JOIN METADATA_D D
        ON R.MASTER_ID = D.MASTER_ID
        AND D.STATUS_INDICATOR = 'A'
LEFT JOIN CODESET_R CR
        INNER JOIN CODESET_D CD
        ON CR.MASTER_ID = CD.MASTER_ID
        AND S.SOURCE_TIMESTAMP
        BETWEEN CD.EFFECTIVE_TIMESTAMP AND CD.EXPIRATION_TIMESTAMP
        ON CR.CONTEXT_KEY_01 = D.CODESET_CONTEXT_KEY_01
        AND CR.CONTEXT_KEY_02 = D.CODESET_CONTEXT_KEY_02
        AND CR.CONTEXT_KEY_03 = S.VALUE
    
```

Note the comments in the sample code listing that indicate where some of the selected columns are only needed in one or two of the main pipes. Also note the LEFT JOIN to the codeset dimension that provides for translating the inbound source value if the metadata dimension has specified a translation codeset, and the value received matches a value in the existing codeset. A LEFT JOIN is used because no translation is done if the sourced value doesn't exist in the associated codeset.

Resolving “~All~” Entries

Two of the columns being selected in the metadata transformation query allow for some special capability relative to the other columns. The Logical Dataset and Fact Break can be set up in the SDI job or in the metadata as “~All~” values. When a source dataset contains a “~All~” value, it means that the source column value should be mapped to any of the respective logical datasets or fact breaks that occur in the metadata. When the “~All~” occurs on the metadata side, it means that the metadata should be matched with incoming source values for any inbound logical dataset or fact break, respectively.

When you select these data in your transformation, you want the more specific value—not “~All~”—to be carried into the associated pipe so the data controls you'll build around the pipes will be as specific as possible. For example, if your physical dataset arrives containing two logical dataset values, such as donor and recipient, you want your data tracking controls to be built around those split values, not a single generic “~All~” value. To accomplish this, the selection of these two columns in the transformation join requires case statements to obtain the more specific value when they differ:

```

CASE WHEN S.LOGICAL_DATASET = R.CONTEXT_KEY_02
    THEN S.LOGICAL_DATASET
    WHEN S.LOGICAL_DATASET = '~All~'
        THEN R.CONTEXT_KEY_02
        ELSE S.LOGICAL_DATASET
    END AS LOGICAL_DATASET,
    
```

```
CASE WHEN S.FACT_BREAK = R.CONTEXT_KEY_03
      THEN S.FACT_BREAK
      WHEN S.FACT_BREAK = '~All~'
          THEN R.CONTEXT_KEY_03
      ELSE S.FACT_BREAK
  END AS FACT_BREAK,
```

Since the default query selection picks up the Logical Dataset and Fact Break from the source intake, these Case statements effectively ensure that the metadata value will be used when it is specific and the source provides the “~All~” value.

The use of “~All~” as Logical Dataset or Fact Break values is completely optional, although its selective use in certain source datasets can dramatically impact the number of rows of data that end up in the staging tables during ETL execution. Some source datasets get very complicated in their mappings. Imagine a source table that includes two kinds of rows (i.e., often representing success or failure of the event described in the data). Each row might contain two kinds of facts that need to be interpreted differently depending upon the row type it occurs in. This one table includes four different facts based on the permutations of rows (Logical Datasets) and columns (Fact Breaks). Each of these facts must have its own metadata definitions in order to correctly map into the data warehouse. The source table will include many columns that also need to be mapped (e.g., Patient ID, Encounter ID, Calendar Date, Clock Time), but that don’t need to be mapped differently for those four different facts. These columns can be sourced and mapped just once, with “~All~” as the Logical Dataset and Fact Break, instead of having to be sourced and mapped separately for each of the four facts. That’s a savings of three mappings for each of the source columns. If you plan to load tens of millions of source rows of history data into the warehouse, the use of “~All~” could save hundreds of millions of extra rows from being written to the staging tables during processing. I’ve seen cases where a row having 5 different kinds of facts includes 12 different dimensional identifiers. Using “~All~” to define and map that source saved 48 transformed source-metadata rows per source table row. If the original source table contained 200 million rows of history, this usage would avoid putting 9.6 billion rows of data into the staging tables at execution time.

Early versus Late Binding

If you are concerned about the amount of system memory that needs to be allocated for the results of this transformation query, one tuning option you can consider is to not select the various processing parameters from the metadata Reference and Definition tables into your staging table. The query already selects the metadata Master ID, and so the two metadata dimension tables can be rejoined at any time to obtain the needed parameters. In fact, software engineers who are proponents for binding to data as late as possible would argue that

we should only build our data that way. If you choose that approach, the list of columns selected in the metadata transform query would stop at the metadata Master ID:

```

SELECT S.JOB_ID,
       S.PHYSICAL_DATASET,
       CASE WHEN S.LOGICAL_DATASET = R.CONTEXT_KEY_02
             THEN S.LOGICAL_DATASET
             WHEN S.LOGICAL_DATASET = '~All~'
                 THEN R.CONTEXT_KEY_02
                 ELSE S.LOGICAL_DATASET
             END LOGICAL_DATASET,
       CASE WHEN S.FACT_BREAK = R.CONTEXT_KEY_03
             THEN S.FACT_BREAK
             WHEN S.FACT_BREAK = '~All~'
                 THEN R.CONTEXT_KEY_03
                 ELSE S.FACT_BREAK
             END FACT_BREAK,
       S.ENTERPRISE_ID,
       S.DEFAULT_KEY_01,
       S.BREAK_ID,
       S.SOURCE_TIMESTAMP,
       S.DATAFEED_BREAK,
       S.ETL_TRANSACTION,
       S.TARGET_STATE,
       COALESCE(CD.DESCRIPTION, S.VALUE) AS VALUE,
       R.MASTER_ID AS METADATA_MASTER_ID
  FROM STAGING.SOURCE S . . .

```

I choose to use the earlier-binding approach of actually selecting my processing parameters as part of the main metadata transformation join. I have several reasons for this. First, the metadata isn't going to change during the execution of the ETL system, so most of the risks of early binding are not present in our ETL subsystem. Second, the late binding approach makes all of my other ETL code more complex because I have to join to the two metadata dimension tables almost every time I touch any of these data since the parameters under which the ETL operates are in those dimension tables. Third, I find that having all of the parameters carried along in the staging tables with the source-target transformations makes debugging and support much easier because I can look at the contents of those staging tables during and between the various steps of the ETL process. Fourth, the performance impacts of moving that extra data through the various pipes isn't nearly as severe as some people think. The impact is actually less severe than the performance hit one accepts by always joining to the metadata dimension tables, although either way your database optimizer is likely to pin the needed metadata data values in memory.

If memory allocation and disk persistence are the key concerns for your implementation, then far more resources can be saved by not doing one monster

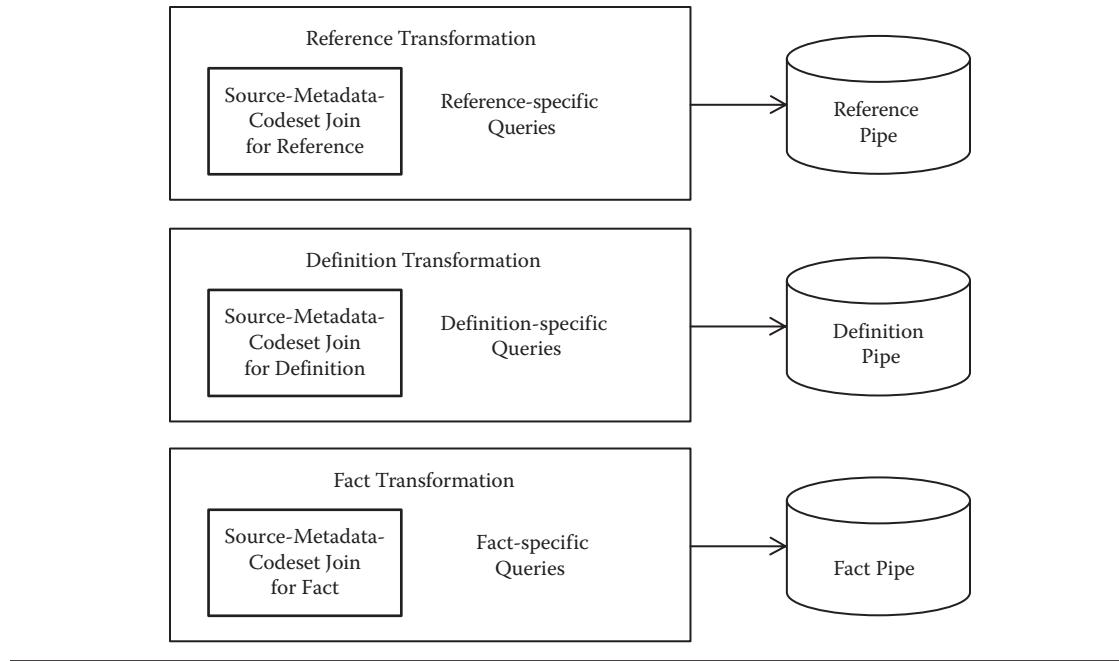


Figure 9.10 Functionally specific metadata transformation.

metadata transformation and instead building a separate metadata transformation join for each of the three main pipes. Tailoring the metadata transformation joins to each functional pipe allows the selection of only the parameters needed to process data in that pipe, and it also allows the main transformation query to be subordinated to a higher-level functionally specific query that will take the data a step further down the pipe than a single generic transformation join would be able to do ([Figure 9.10](#)).

I try to persist these data as seldom as possible since I expect millions of rows of data to process every day and potentially tens or hundreds of millions of rows to process during initial loading. Keeping the data functionally split, and limiting the data persistence operations, allows for better utilization of memory and physical input/output. As a result, each functional pipe will begin with a metadata transformation operation that appears very similar at its core, but that differs in a few key functional characteristics. Those specifics will be discussed at the beginning of each of the three following chapters that deal individually with the requirements of those three main pipes.

Data Control Pipe

The Data Control pipe is part of the system of internal controls for the warehouse that provides traceability for data back to the source datafeeds that brought in the data, as well as a quantitative view into the data and its sources ([Figure 9.11](#)). At the end of all of the ETL jobs, the Data Source pipe is used to populate the Datafeed dimension of the warehouse.

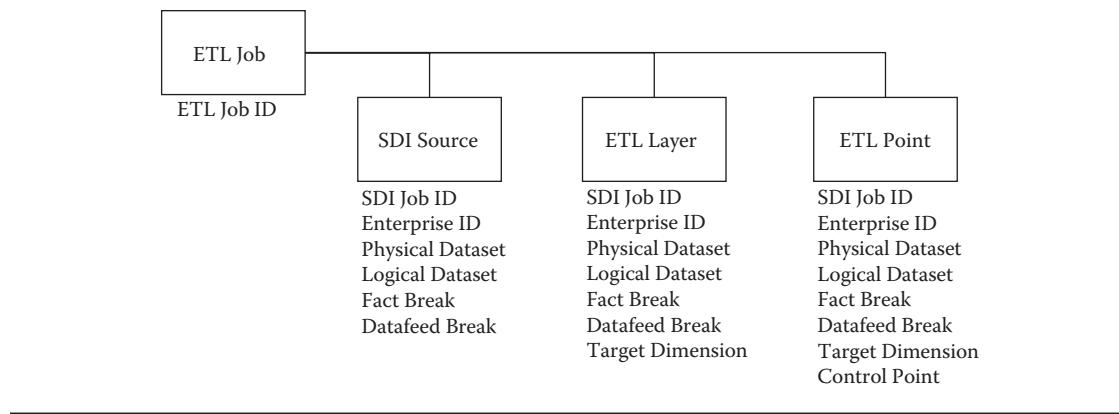


Figure 9.11 Data control pipe entries.

ETL Job

The first step in the Data Control pipe is the assignment of an ETL job ID to the current execution and, in the process, confirming that there isn't already another ETL job set executing. This confirmation is obtained by looking into the New Surrogates table to see if a job ID is already present:

```
SELECT COUNT(*) FROM NEW_SURROGATES
WHERE PROCESS = 'ETL_JOB_ID';
```

If this query returns a zero count, there are no currently executing ETL jobs. The last ETL job deletes the entry in the New Surrogates table. If the row is still present, that job has not completed and a new ETL job should not be started. If it's acceptable to initiate a new ETL job, then inserting a single row into the New Surrogates table will assign the next surrogate as the ETL Job ID:

```
INSERT INTO NEW_SURROGATES (PROCESS)
SELECT 'ETL_JOB_ID';
```

The ETL Job ID can either be retrieved into a global variable if your development environment supports that feature, or you can obtain it by joining to the New Surrogates table at any time:

```
SELECT NEW_SURROGATE AS ETL_JOB_ID
FROM NEW_SURROGATES
WHERE PROCESS = 'ETL_JOB_ID';
```

The ETL Job ID is the highest-order identifier in the Data Control pipe because all of the controls designed into the warehouse take place within the context of an ETL job.

Data Control Layers

The second step in setting up the Data Control pipe is to consider the layers of control that are desired. Data control layers represent the granularity with which

you will be able to discern how and when things were done in the loading of the warehouse. The more layers you include, the more granular your level of control. The layers desired need to be defined in a control table that you create for that purpose in your ETL staging areas. The columns in the Data Control Layers table will determine the granularity of the controls throughout the ETL process.

To illustrate, let's start by assuming that you only want to know which ETL job was the originating source for the data in the warehouse. To do that, you only need to include one column in the Data Control Layers table:

```
CREATE TABLE TEST.DATA_CONTROL_LAYERS (
    ETL_JOB_ID          INT,
    MASTER_ID           INT
);
```

Since the instances of data that are placed in the Data Control Layers table end up as the tracking rows for the data in the Datafeed dimension, this will result in every fact loaded in an ETL job being identifiable as having been loaded by that ETL job. In essence, you'd know when every fact was loaded. That level of control wouldn't be very granular, but it could answer certain high-level questions about the provenance of data in the warehouse.

To gain slightly more control, let's assume that in addition to knowing which ETL job loaded a piece of data, you'd also like to know the original source job that extracted the source data for loading. To do that, you'd add the source job's ID to your Data Control Layers table:

```
CREATE TABLE TEST.DATA_CONTROL_LAYERS (
    ETL_JOB_ID          INT,
    SDI_JOB_ID          INT,
    MASTER_ID           INT
);
```

At this level of granularity in control, you'd be able to know not only what ETL job loaded a piece of data, but you'd also know exactly which SDI job extracted the data from their source. Even with lots of different kinds of data loading in the same ETL job, you'd be able to differentiate data that arrived in a patient extract from a set of clinical results that were extracted from a laboratory system, if you knew the coverage provided by each SDI Job ID.

It can be difficult to anticipate all the layers of control that you might want to be able to offer in your data warehouse in advance of having the warehouse in place and seeing the types of data that get loaded. Fortunately, the numbers of layers you decide to include doesn't really affect the algorithmic complexity of the Data Control pipe. For this Beta version, you can keep the layers fairly simple, and you can revisit the decision in the Gamma version when you have more experience with your data and you are starting to develop your data administration and data governance strategies and procedures.

For the Beta version, I recommend including controls that provide visibility into the details of the SDI job (e.g., enterprise, physical and logical datasets, and

fact break), as well as the Target Dimension into which data loads. You will also include the arbitrary Datafeed Break, even if you haven't used it yet, since it might be used at any time in one of the SDI jobs for the Beta version. This choice will give you the following Data Control Layers table:

```
CREATE TABLE DATA_CONTROL_LAYERS (
    ETL_JOB_ID          INT,
    SDI_JOB_ID          INT,
    ENTERPRISE_ID       INT,
    PHYSICAL_DATASET    VARCHAR(15),
    LOGICAL_DATASET     VARCHAR(15),
    FACT_BREAK          VARCHAR(15),
    DATAFEED_BREAK      VARCHAR(15),
    TARGET_DIMENSION    VARCHAR(15),
    MASTER_ID           INT
);
```

The layers of interest in the Data Control pipe are the permutations of instances of all the data you've selected for inclusion in the Data Control Layers table. Most of that data come from the inbound SDI data, so it is available as soon as the ETL workflows begin executing. The Target Dimension is not available in the inbound source data, so it must be generated explicitly in order to define layers for control that can handle dimensional distinctions. The final values for the dimensions in the control layers are contained in the metadata dimension, so the Target Dimension will be NULL until after each metadata transformation join.

To create the necessary permutations of data in the Beta version, create a separate small table that contains the Target Dimension names that you want to include controls for:

```
CREATE TABLE TEST.ZDIMENSIONS (
    TARGET_DIMENSION VARCHAR(15)
);
```

Then you explicitly load that table with your selected dimension names. You should also load a NULL row that will be used for any controls you implement prior to joining the SDI intake data with the metadata:

```
INSERT INTO TEST.ZDIMENSIONS
    SELECT 'Organization'
UNION SELECT 'Caregiver'
UNION SELECT 'Study'
UNION SELECT 'Subject'
UNION SELECT 'Calendar'
UNION SELECT 'Clock'
UNION SELECT 'Facility'
UNION SELECT NULL;
```

Any dimensions omitted from this table will not be controlled by the Data Control pipe. The choice of what dimensions to control is an organizational decision you'll need to make. I recommend controlling on all of the dimensions during the Beta load. With the control layers and deferred dimension data defined, the Data Control Layers table is populated at the beginning of the execution of the ETL job streams using data in the Consolidated SDI table:

```
INSERT INTO DATA_CONTROL_LAYERS (
    ETL_JOB_ID,
    SDI_JOB_ID,
    ENTERPRISE_ID,
    PHYSICAL_DATASET,
    LOGICAL_DATASET,
    FACT_BREAK,
    TARGET_DIMENSION,
    DATAFEED_BREAK)

SELECT DISTINCT
    NEW_SURROGATE AS ETL_JOB_ID,
    SDI_JOB_ID,
    ENTERPRISE_ID,
    PHYSICAL_DATASET,
    LOGICAL_DATASET,
    FACT_BREAK,
    TARGET_DIMENSION,
    DATAFEED_BREAK
FROM CONSOLIDATED_SDI
CROSS JOIN NEW_SURROGATES
CROSS JOIN ZDIMENSIONS
WHERE PROCESS = 'ETL_JOB_ID';
```

Note that by joining to your table of generated dimension names, your control layers include entries for source-dimension permutations that might not exist in particular source datasets being processed. You might think that if you had waited until after the metadata transformation joins then you could have used the Target Dimension entries in the actual transformed data to limit the creation of control layers to only those source-dimension combinations that are present in the actual source datasets. Creating the extra control layers might seem unnecessary, even wasteful. Here in the Beta version of the warehouse, that assessment actually holds true. You are only going to load data against the dimensions defined in the metadata, and so waiting to create the control layers until after the metadata transforms would produce the same results. However, that notion will break down as you move toward your Gamma version and begin adding additional functions to your ETL capabilities, including abilities to up-dimensionalize facts to include dimensionality not defined in the metadata, nor present in the source data.

Building your control layers to include all of your possible combinations of interest now will provide you more flexibility and control later. Layers that end up not being used will not be inserted into the Datafeed dimension, and so no permanent record of the extra hypothetical permutations will exist until the added functionality that uses them is added to the ETL. For now, the Target Dimension is the only deferred data value that you have had to generate to build your control layers. There will likely be others in the future, so we'll return to the topic in the discussion of controls in the Gamma version. The rows generated in the Data Control Layers table will eventually be loaded as new rows in the Source subdimension of the Datafeed dimension (Chapter 13).

Data Control Points

Having established your desired data control layers, you are now in a position to start incorporating data control points anywhere you want them throughout any of the ETL work flows. The Data Control Points table is defined based on your selection of columns in the Data Control Layers table. At whatever grain you've selected in that table, an additional new Data Control Points table adds three more columns:

```
CREATE TABLE DATA_CONTROL_POINTS (
    ETL_JOB_ID          INT,
    SDI_JOB_ID          INT,
    ENTERPRISE_ID        INT,
    PHYSICAL_DATASET    VARCHAR(15),
    LOGICAL_DATASET     VARCHAR(15),
    FACT_BREAK          VARCHAR(15),
    TARGET_DIMENSION    VARCHAR(15),
    DATAFEED_BREAK      VARCHAR(15),
    CONTROL_POINT       VARCHAR(30),
    CONTROL_COUNT       INT,
    CONTROL_TIMESTAMP   DATETIME
);
```

At any time within your ETL logic, you can take a count of rows contained in your staging tables against the control layers you've specified and; by adding a name to the control point you've taken, you will obtain a count of rows by whatever grain you've placed in your layers table. For example, you can start your initial ETL stream by taking a count of inbound source data in your consolidated SDI input table:

```
INSERT INTO DATA_CONTROL_POINTS (
    ETL_JOB_ID,
    SDI_JOB_ID,
    ENTERPRISE_ID,
```

```

PHYSICAL_DATASET,
LOGICAL_DATASET,
FACT_BREAK,
TARGET_DIMENSION,
DATAFEED_BREAK,
CONTROL_POINT,
CONTROL_COUNT,
CONTROL_TIMESTAMP
)
SELECT
    ETL_JOB_ID,
    SDI_JOB_ID,
    ENTERPRISE_ID,
    PHYSICAL_DATASET,
    LOGICAL_DATASET,
    FACT_BREAK,
    NULL                      AS TARGET_DIMENSION,
    DATAFEED_BREAK,
    MAX('Inbound SDI')        AS CONTROL_POINT,
    COUNT(*)                  AS CONTROL_COUNT,
    MAX(SYSTIME())            AS CONTROL_TIMESTAMP
FROM CONSOLIDATED_SDI
GROUP BY
    ETL_JOB_ID,
    SDI_JOB_ID,
    ENTERPRISE_ID,
    PHYSICAL_DATASET,
    LOGICAL_DATASET,
    FACT_BREAK,
    TARGET_DIMENSION,
    DATAFEED_BREAK;

```

Note that NULL is selected as the Target Dimension in this particular control point precisely because the column is not yet available at the point in the ETL processing represented by the control point. The rows generated in the Data Control Points table will eventually be loaded as new rows in the Count subdimension of the Datafeed dimension.

Assigning Master IDs to ETL Layers

All of the data control information generated so far will end up in the Datafeed dimension toward the end of the ETL subsystem execution. In the meantime, the data will be used and supplemented throughout the main Reference, Definition, and Fact pipes described in the next three chapters. Entries in the dimensions will include the Master ID of the ETL layer that both created and modified them. Each new fact will dimensionalize to the ETL layer that was the source of the fact. To accomplish this connectivity, the logic in those pipes will need access to the Master ID that will define each entry once it is inserted into the Datafeed

dimension. This means the entries in the Data Control Layers table need to be passed into the New Surrogates table to obtain those surrogate identifiers:

```
INSERT INTO NEW_SURROGATES (PROCESS, DIMENSION, SUBDIMENSION, CONTEXT_ID, COMPOSITE)
SELECT 'ETL Points', 'Datafeed', 'Layer', CONTEXT_ID,
      CONCAT(ETL_JOB_ID,           '||',
             SDI_JOB_ID,          '||',
             ENTERPRISE_ID,        '||',
             PHYSICAL_DATASET,     '||',
             LOGICAL_DATASET,       '||',
             COALESCE(FACT_BREAK,   '|~Null~'),  '||',
             COALESCE(TARGET_DIMENSION, '|~Null~'), '||',
             COALESCE(DATAFEED_BREAK, '|~Null~') ) AS COMPOSITE
FROM DATA_CONTROL_LAYERS
INNER JOIN DATAFEED C WHERE CONTEXT COMPOSITE = 'ETL Layer';
```

Wide versus Deep Data

Having established the Data Control pipe, and having discussed the general need for a metadata transformation join to bring source data into the ETL subsystem, we're ready to move forward to the main Reference, Definition, and Fact pipes in the next three chapters. Each will serve its specific function, and collectively, their execution results in getting your data loaded. The loading of these data in the Beta version is all about the source data. As a result, some of the coding conventions presented here might appear overengineered for simply bringing data into the warehouse from the sources. The value of this complexity won't be directly apparent until you move toward the Gamma version where you'll be generating warehouse data from within the ETL workflow. These extra data will include error and warning conditions, semantic annotations, alternative data transformations, various quantitative metrics, and inferred extra dimensionalization. We omit those features here in the Beta version in order to expedite initial data delivery, but the code presented in these chapters is preparatory toward those features.

Throughout the main pipes, you'll see references to the data being *wide* or *deep*. The distinction is important to the generality of the ETL subsystem because it allows data to be handled very generically at the row (wide) or column (deep) level as needed by the general logic. Transitioning from wide data to deep data is generally accomplished with an SQL UNPIVOT command, while the transition from deep data to wide data is generally accomplished with an SQL PIVOT command.

Your data start out wide in their source application systems, and they end up wide in the warehouse dimensions. The Fact table is very deep with respect to the main value column, but it is wide otherwise, with respect to its 26 foreign keys. The SDI jobs take wide source data and unpivot it into the deep 13-column generic structure used to bring data into the ETL subsystem. In the Reference pipe, the 10 deep Context Keys will be pivoted into a wide Reference Composite in order to build that generic key in the Reference tables. In the Fact pipe,

the deep reference data with their Role, Rank, and Weight values will be pivoted into wide Bridge entries. Those bridges will later be pivoted again, by dimension, to create the Group Composites to build the necessary entries in the Group tables. Finally, the Fact pipe will pivot all of those Group entries into a single row for insertion into a Fact table.

The power of this generic process is that a single Group ID in a fact table might be the result of extracting a single column from a source system, or it might be the result of sourcing 50 columns that were used to create 10 multipart keys. A single new Master ID might result from a simple source with 5 columns that simply added a new dimension entry, or it might result from 20 columns of data being received at the same time from 4 different source systems. The complexity of the sources doesn't impact the execution of these generic pipes. The same generic logic handles every case because the parametric data needed to control the load process has been externalized from the programming code into the Metadata dimension.

Chapter 10

ETL Reference Pipe

The Reference pipe takes its name from the notion that the foci of all of the activity in this pipe are the Reference tables of each of the dimensions in the star model (Figure 10.1). The individual transactional purposes might include resolving natural keys to their respective entries in the Definition tables, building Hierarchy entries to relate two Definition entries, or defining alias natural keys for previously defined or existing keys. All of these transaction types share the need to receive, format, look up, and possibly create Reference entries for natural keys that occur in the source systems. This Reference pipe handles all of these situations, adding considerable additional value to the processing as it occurs. Specifically, the processing covered in the Reference pipe includes

- Transforming source-only intake data into source-to-target staged data through a join to the Metadata dimension, including codeset translation of nonstandard source values
- Formatting each of up to 10 natural key source values into the standard single-valued source natural key known as the Reference Composite
- Resolving a Master ID for the dimension against each Reference Composite, either because the Master ID already exists in the dimension or it is created for each distinct Reference Composite not found in the dimension
- Inserting new dimension data for newly created Master IDs, including the standard Group, Bridge, and Hierarchy entries required of all new Definition entries in each dimension
- Inserting new alias entries in the dimension Reference tables
- Inserting new flat hierarchy entries in the dimension Hierarchy tables
- Inserting new dimension Hierarchy entries for any natural hierarchies that have been impacted by the creation of new Reference entries in the dimensions

The processing in this Reference pipe will very likely be much easier to build and test than you might be expecting. The dimension design pattern covered

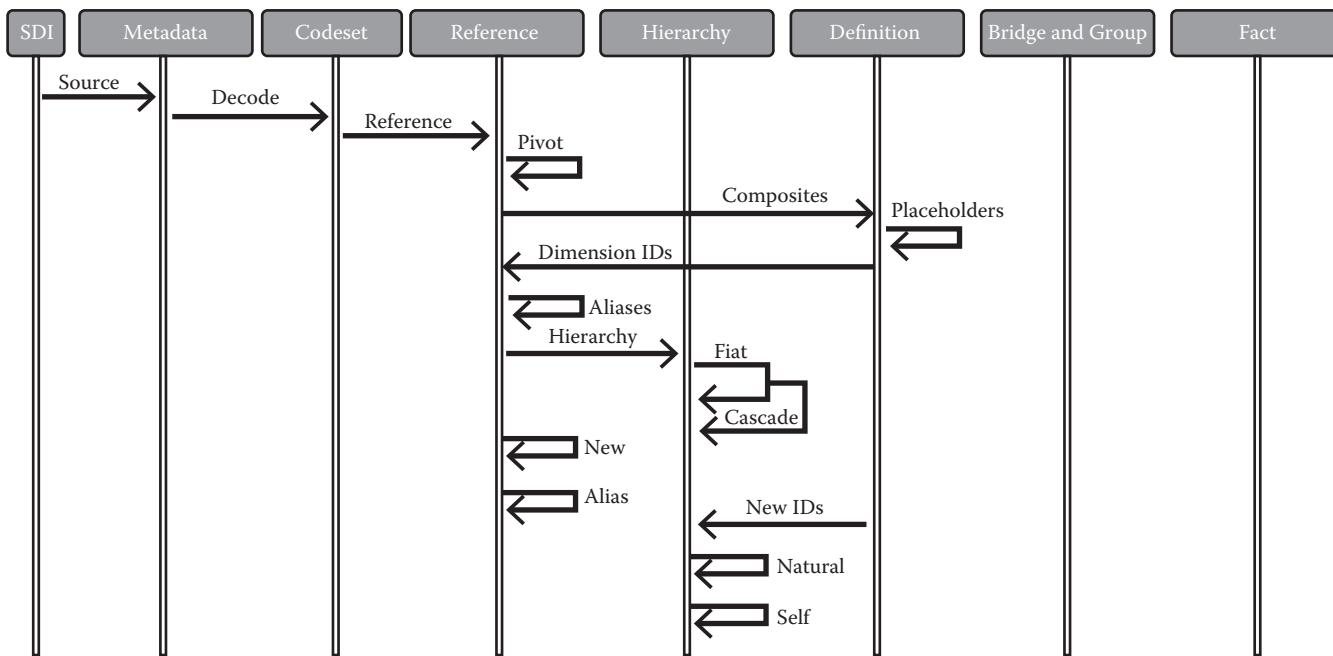


Figure 10.1 Sequence diagram for Reference pipe processing.

back in Chapter 5 allows for each component of the dimension model to be implemented independently of each of the other components. To take advantage of this, it is very important that you are able to let go of the relational or transactional paradigm under which you've probably implemented databases in the past. That paradigm is exemplified by the notion of foreign keys existing between tables, with your database management system enforcing referential integrity among the tables as data are inserted. In this design pattern, we have very few actual foreign key possibilities among the six tables in the dimension.

The main column that ties together most of the data in each dimension is the Master ID, but the Master ID is not the unique identifier of the Definition table in which it is first defined. The actual key to the Definition table is the Definition ID, and it is not present to serve as a foreign key in any of the other five dimension tables. Joining two tables on Master ID (e.g., Definition–Reference, Definition–Hierarchy, Definition–Bridge) always involves the set of rows that share the Master ID in question, with uniqueness being resolved through either the status of the Definition, or the effective date of the transaction for which the join is being accomplished.

I don't want to give you the impression that the design pattern avoids foreign keys simply because there aren't any to enforce. There is an older variant of the dimension model that would have used the Definition ID instead of the Master ID in the Reference, Hierarchy, and Bridge tables. Under that model, joins to the Definition table would always be to a single row defined by the Definition ID. Typically, the joins would always resolve to the Definition row active at the time of the transaction being processed. This approach could be made logically consistent, capable of providing any of the functionality that the design pattern in Chapter 5 can provide but at a high cost in terms of data and processing.

First, different facts pointing to the same Definition, but different temporal versions of the Definition, would need to have different entries inserted to the Bridge and Group tables because of differences in the Definition ID caused by the variation in the Definitions. Pointing to the Definition ID rather than the Master ID unnecessarily pulls knowledge of the Definition variations into the Bridge table, dramatically increases the size of the Bridge and Group tables and also lowers the cardinality of each bridge's connection to the Definition table through any indexes that have been created. This would also mean that the dimensionalization of facts to a dimension could be out of date as a result of late-arriving changes to the Definitions, causing a temporal change that results in some facts pointing to Definitions that weren't the active ones at the time of the facts. This would also make duplicate facts harder to detect because two facts that are logically the same might appear different because of a temporal change that occurred in one of the dimensions since the first version of the duplicate fact arrived.

Second, using Definition ID to join hierarchy definitions would necessitate propagating all slowly-changing dimension transactions that occur in the Definition table over to the Hierarchy table in order to keep the hierarchy data

up to date and in sync with evolving Definition entries. These updates to the hierarchy would add complication to the slowly-changing dimensionality changes that are already being managed in the hierarchy tables. Third, a slowly-changing aspect would have to be added to the Reference table. Otherwise, joining from Reference to Definition would require a second join to the Definition table to ensure that the correct Definition resulted from the join, based on desired status or transaction effective time.

This design pattern avoids all of these problems by using the Master ID as the joining construct, with resolution to any desired temporal variant or status resolved at query time. My point is that, even if I were still using Definition ID in my joins, I would not want to rely on referential integrity against foreign keys to enforce integrity in the data. Adding referential integrity enforcement to a database adds a great deal of overhead to the processing of data during inserts and updates, and you don't want that overhead associated with your ETL executions. Even the two actual foreign keys in the design pattern (e.g., Context ID and Group ID) should *not* have referential integrity turned on.

The logic of this Reference pipe is very straightforward, and this is the only place where these tables are updated in our emerging warehouse. After proper testing of our development, referential integrity violations become virtually nonexistent. When they do occur, it will be because you've added a data source and made a mistake in the definition of metadata for that source. Chapter 13 will focus on mistake proofing any new metadata to avoid those scenarios, but some mistakes still will get through your verification and validation. With referential integrity enforced, your ETL jobs will fail completely when a single violation is encountered. This kind of job failure was bad enough back in the days when you processed hundreds of jobs per day. We could fix that one job and rerun it, but now you will only be running this ETL once for all of your data. You don't want it to fail. Instead, you want violations to load knowing that you can easily spot and correct mistakes right in the warehouse. The loose coupling of data implied by a lack of enforced foreign keys actually has a positive side of making the data much less brittle, making corrective actions very routine to define and carry out in Chapter 13.

If you've grown up in the world of traditional database designs, with very tight coupling of data around foreign keys, a lot of this will seem unusual, even weird. Trust the process as you learn it. Your experience with the Alpha version was meant to help you understand and appreciate the design pattern. Next, you'll see how easy that pattern makes your maintenance of the dimensions.

Metadata Transformation

The first major step in the Reference pipe process is the functionally-specific metadata transformation introduced in Chapter 9, in which the metadata and source data created in Chapter 8 are joined to create the initial reference staging

area for the data. While the source dataset contains everything you know about the contents of the data source, the resulting transformed staging data also contains everything you need to know about the targets of that data in order to be able to completely process the data through the entire ETL subsystem and into the data warehouse without needing to refer to the source again. This step is the key to the generic ETL architecture.

You'll start with the metadata transformation for the Reference pipe data. This data will include any source data that are targeted to make up any of the application keys in any of the Reference tables, including any of the 10 Context Key columns in any of the dimensions. The join is between the source data table and the metadata and codeset dimensions:

```

SELECT S.SDI_JOB_ID,
       S.PHYSICAL_DATASET,
       CASE WHEN S.LOGICAL_DATASET = R.CONTEXT_KEY_02
            THEN S.LOGICAL_DATASET
            WHEN S.LOGICAL_DATASET = '~All~'
            THEN R.CONTEXT_KEY_02
            ELSE S.LOGICAL_DATASET
            END LOGICAL_DATASET,
       CASE WHEN S.FACT_BREAK = R.CONTEXT_KEY_03
            THEN S.FACT_BREAK
            WHEN S.FACT_BREAK = '~All~'
            THEN R.CONTEXT_KEY_03
            ELSE S.FACT_BREAK
            END FACT_BREAK,
       S.ENTERPRISE_ID,
       S.DEFAULT_KEY_01,
       S.BREAK_ID,
       S.SOURCE_TIMESTAMP,
       S.DATAFEED_BREAK,
       S.ETL_TRANSACTION,
       S.TARGET_STATE,
       COALESCE(CD.DESCRIPTION, S.VALUE) AS VALUE,
       R.MASTER_ID AS METADATA_MASTER_ID,
       D.DATASET_TYPE,
       D.TARGET_DIMENSION,
       D.DIMENSION_CONTEXT,
       D.TARGET_SUBDIMENSION,
       R.CONTEXT_KEY_05 [TARGET_TABLE],
       R.CONTEXT_KEY_06 [TARGET_COLUMN],
       R.CONTEXT_KEY_07 [BRIDGE_ROLE],
       R.CONTEXT_KEY_08 [BRIDGE_RANK],
       R.CONTEXT_KEY_09 [BRIDGE_WEIGHT]
  FROM STAGING.SOURCE S
 INNER JOIN METADATA_R R
    ON S.PHYSICAL_DATASET = R.CONTEXT_KEY_01
   AND (S.LOGICAL_DATASET = R.CONTEXT_KEY_02)

```

```

        OR S.LOGICAL_DATASET = '~All~'
        OR R.CONTEXT_KEY_02 = '~All~')
    AND (S.FACT_BREAK = R.CONTEXT_KEY_03
        OR S.FACT_BREAK = '~All~'
        OR R.CONTEXT_KEY_03 = '~All~')
    AND S.SOURCE_COLUMN = R.CONTEXT_KEY_04
INNER JOIN METADATA_D D
    ON R.MASTER_ID = D.MASTER_ID
    AND D.STATUS_INDICATOR = 'A'
LEFT JOIN CODESET_R CR
    INNER JOIN CODESET_D CD
        ON CR.MASTER_ID = CD.MASTER_ID
        AND S.SOURCE_TIMESTAMP
            BETWEEN CD.EFFECTIVE_TIMESTAMP AND CD.EXPIRATION_TIMESTAMP
        ON CR.CONTEXT_KEY_01 = D.CODESET_CONTEXT_KEY_01
        AND CR.CONTEXT_KEY_02 = D.CODESET_CONTEXT_KEY_02
        AND CR.CONTEXT_KEY_03 = S.VALUE
WHERE D.TARGET_TYPE IN ('Reference', 'Alias', 'Hierarchy');

```

This transformation join is very much like the general version illustrated in Chapter 9, except that the sourced columns that would have been needed for the Definition and Fact pipes aren't included in this functionally specific version of the query. The final WHERE clause is what indicates that this join is in the logical Reference pipe. The columns selected include 12 of the 13 generic source data columns (i.e., the 13th Source Column isn't needed after this point in the processing), as well as Metadata columns that will drive the processing through the generic ETL processes.

Codeset Translation

The metadata transformation includes the standard codeset translation covered in Chapter 9. Here in the Reference pipe, the codeset translation plays an important role in harmonizing data quality issues in arriving source keys. As part of analyzing data in order to develop metadata, the analyst typically profiles the source data to determine the range of values that might be encountered while loading data. For source natural keys that will come through this Reference pipe, the analyst is particularly interested in variations in values for logically equivalent natural keys that would result in erroneous extra data being created in the warehouse dimension.

Let's take a look at an example in the Geopolitics dimension. Let's suppose you're adding countries, states, and cities to the dimension as three different subdimensions that will eventually be aggregated into a natural hierarchy structure. The analyst has determined that the Country abbreviation will be used in Context Key 01 and the State/Province abbreviation in Context Key 02. The name of the city will be used in Context Key 03, but the analyst determined that no codeset translation will be used for city names.

Table 10.1 Example Source Geopolitics Country–State–City Data

Break ID	Source Column	Source Value
111	Country	USA
111	State	FL
111	City	Miami
222	Country	U.S.A.
222	State	Fla.
222	City	Miami
333	Country	USA
333	State	Florida
333	City	Maimi

Table 10.1 illustrates some of the variation in values that might be encountered when profiling the source data to be used for this desired loading. The data includes variations in the Country code used for the United States, variations in the State abbreviation used for Florida, and an actual misspelling of Miami. If left unchecked, the data just in Table 10.1 would result in loading two countries, three states, and three cities because of anomalies in those natural key columns. Note that there are only two distinct values for the City in the table, but the two correctly spelled values have different associated State abbreviations, which is why the load would result in three cities rather than just the two that might be expected based on the number of distinct values for the actual City source column.

These anomalies can be managed generically by adding codeset values that will translate the erroneous data into the correct values for loading. Table 10.2 shows the needed codeset entries. It includes a Country Codes codeset that translates “U.S.A.” into the standard “USA” value, and it includes a State Codes codeset that translates the two anomalous State values into their corrected “FL” value. These codesets can contain an arbitrary number of data translations based on the profile analysis completed during the review of source data. The needed codeset entries will allow variations in abbreviations to be harmonized during ETL loading.

Table 10.2 Example Geopolitics Translation Codesets

Context Key 01	Context Key 02	Context Key 03	Description
~Our Warehouse~	Country Codes	U.S.A.	USA
~Our Warehouse~	State Codes	Florida	FL
~Our Warehouse~	State Codes	Fla.	FL

Table 10.3 Geopolitics Translation Codeset Row for Maimi–Miami

Context Key 01	Context Key 02	Context Key 03	Description
~Our Warehouse~	City Name	Maimi	Miami

Note that the aforementioned example does not include any codeset translation for the misspelled city name. The reason for not including a translation for City is that codeset translation is a single-column function. To use a codeset translation to change “Maimi” to “Miami,” you’d have to be sure that you want that change to occur across all States in all Countries. If “Maimi” might be a legitimate spelling of a city in another place, you wouldn’t want to use codeset translation to universally alter it. If your data analyst decides that the change should be universal, the entry in Table 10.3 could simply be added to the codeset.

Once the codesets are defined, the Codeset portion of the metadata transformation join can take advantage of them. Remember that by using a LEFT JOIN and a COALESCE in the Codeset join, the source Value carries through the process if no entry is found in the Codeset. This means that codesets only need to be defined for intended translations and need not be considered or included otherwise. The result of all of this mapping would be that all of the data in [Table 10.1](#) would ultimately resolve to a single city: Miami, FL, USA. The dimension would have a single Definition for the city (as well as a single definition for the state and for the country). The anomalies in the data in the source system will have been harmonized while bringing the data through the Reference pipe.

To the extent that the source data analyst on the warehouse team can profile new source data and properly anticipate these harmonization problems, the codesets can be defined and implemented as part of the build of the Alpha version of the warehouse. Other anomalies in the source data might be newly encountered after implementation, in which case the anomalous values would be seen as orphan entries in the dimensions. These values can be added to the codesets later so proper harmonization will resume once the anomalies are recognized. However, adding codeset translations to the codesets will not retroactively harmonize previously loaded values in the dimension. Retroactively harmonizing those values will be accomplished in the Gamma version, so the warehouse support team might need to do some small manual cleanup operations for harmonization of orphans that get loaded in the short term. It’s best to avoid these situations ahead of time, so I recommend extensive profiling of suspect data sources as the metadata for new natural keys is being defined.

Reference Processing

An overview of Reference pipe processing is depicted in [Figure 10.2](#). The selection and filtering of the arriving deep source data into the deep staging of the source Context Key columns has already been accomplished by the transformation join.

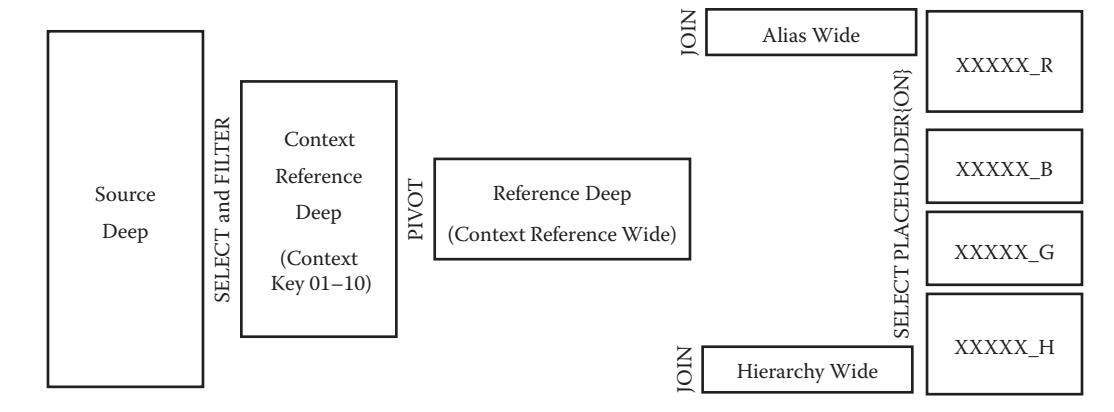


Figure 10.2 Reference processing overview.

The remaining processing will move that data through to the four different tables in each dimension that are maintained through a knowledge of those sourced natural and system keys.

Reference Composite

Once the sourced natural keys have been transformed through the metadata and harmonized through the codesets, you'll need to use the transformed data to make the Reference Composite values that are actually used to define rows in each dimension Reference table. This primarily entails pivoting the multiple reference columns from the up to 10 deep rows of data to one wide row of data. You can accomplish this by embedding your metadata transformation as a subquery in a broader query that completes the desired data pivot:

```

SELECT
    P.SDI_JOB_ID,
    P.PHYSICAL_DATASET,
    P.LOGICAL_DATASET,
    P.DATASET_TYPE,
    P.BREAK_ID,
    P.DATAFEED_BREAK,
    P.ENTERPRISE_ID,
    P.SOURCE_TIMESTAMP,
    P.TARGET_DIMENSION,
    P.TARGET_SUBDIMENSION,
    P.TARGET_TYPE,
    P.DIMENSION_CONTEXT,
    P.BRIDGE_ROLE,
    P.BRIDGE_RANK,
    P.BRIDGE_WEIGHT,
    COALESCE(P.CONTEXT_KEY_01, P.DEFAULT_KEY_01) CONTEXT_KEY_01,
    P.CONTEXT_KEY_02,
    P.CONTEXT_KEY_03,
    P.CONTEXT_KEY_04,
    P.CONTEXT_KEY_05,
    P.CONTEXT_KEY_06,
    P.CONTEXT_KEY_07,
    P.CONTEXT_KEY_08,
  
```

```

P.CONTEXT_KEY_09,
P.CONTEXT_KEY_10,
COALESCE( COALESCE(P.CONTEXT_KEY_01, P.DEFAULT_KEY_01)
+ ISNULL('' + P.CONTEXT_KEY_02, '') + ISNULL('' + P.CONTEXT_KEY_03, '')
+ ISNULL('' + P.CONTEXT_KEY_04, '') + ISNULL('' + P.CONTEXT_KEY_05, '')
+ ISNULL('' + P.CONTEXT_KEY_06, '') + ISNULL('' + P.CONTEXT_KEY_07, '')
+ ISNULL('' + P.CONTEXT_KEY_08, '') + ISNULL('' + P.CONTEXT_KEY_09, '')
+ ISNULL('' + P.CONTEXT_KEY_10, '') ) AS REFERENCE_COMPOSITE
FROM
(
    << Metadata Transformation HERE >>
) PVT
PIVOT
(
    MIN(VALUE)
    FOR TARGET_COLUMN IN (
        CONTEXT_KEY_01,
        CONTEXT_KEY_02,
        CONTEXT_KEY_03,
        CONTEXT_KEY_04,
        CONTEXT_KEY_05,
        CONTEXT_KEY_06,
        CONTEXT_KEY_07,
        CONTEXT_KEY_08,
        CONTEXT_KEY_09,
        CONTEXT_KEY_10,
    )
) P;

```

This query has two important Coalesce functions within it. The first allows the Default Key 01 to substitute for Context Key 01 values that have not been sourced, a very common occurrence when source systems have been architected to have organizationally specific high-order keys associated with some, or all, of their master data. The second Coalesce actually combines the up to 10 natural keys coming through the pivot into the single pipe-delimited Reference Composite column. Table 10.4 illustrates the Reference Composite values that result in the Geopolitics example discussed earlier.

It is the consolidation of multiple natural keys into the single-valued Reference Composite that allows much of the remaining ETL processing to remain generic, since the processing that follows doesn't need to know the makeup or structure of every multipart natural key that might have been used to load data. Through all of the remaining processing, all natural keys

Table 10.4 Example Reference Composite Mappings

Context Key 01	Context Key 02	Context Key 03	Reference Composite
USA			USA
USA	FL		USA FL
USA	FL	Miami	USA FL Miami

are now effectively single-columned entries; making coding and indexing of the data much easier and more generic.

To review, you've developed your metadata transformation against your initially staged source data for all Reference pipe data, including all sourced Context Keys. The innermost subquery joined the source data to the metadata to complete all needed source-to-target mappings, while also including any necessary codeset translations for arriving source values. The higher-level outer query pivoted the up to 10 rows of Context Key data into one row of reference data, further combining those Context Keys into a pipe-delimited single Reference Composite column. The result is that you have all of your sourced natural keys ready to look up in the appropriate dimensions as delimited Reference Composites.

Reference Staging

The data that result from this set of queries can now be persisted into a staging table that will be updated by subsequent lookup processing:

```
CREATE TABLE #SOURCE1_REFERENCE (
    JOB_ID INT,
    PHYSICAL_DATASET VARCHAR(64),
    LOGICAL_DATASET VARCHAR(64),
    DATASET_TYPE VARCHAR(64),
    BREAK_ID INT,
    DATAFEED_BREAK VARCHAR(64),
    ENTERPRISE_ID VARCHAR(64),
    SOURCE_TIMESTAMP DATETIME,
    TARGET_DIMENSION VARCHAR(64),
    TARGET_SUBDIMENSION VARCHAR(64),
    TARGET_TYPE VARCHAR(64),
    DIMENSION_CONTEXT VARCHAR(64),
    BRIDGE_ROLE VARCHAR(64),
    BRIDGE_RANK INT,
    BRIDGE_WEIGHT FLOAT(3,1),
    CONTEXT_KEY_01 VARCHAR(64),
    CONTEXT_KEY_02 VARCHAR(64),
    CONTEXT_KEY_03 VARCHAR(64),
    CONTEXT_KEY_04 VARCHAR(64),
    CONTEXT_KEY_05 VARCHAR(64),
    CONTEXT_KEY_06 VARCHAR(64),
    CONTEXT_KEY_07 VARCHAR(64),
    CONTEXT_KEY_08 VARCHAR(64),
    CONTEXT_KEY_09 VARCHAR(64),
    CONTEXT_KEY_10 VARCHAR(64),
    REFERENCE_COMPOSITE VARCHAR(772),
    CONTEXT_ID INT,
    DEFINITION_ID INT,
    MASTER_ID INT,
    PLACEHOLDER_FLAG BIT,
    UNKNOWN_FLAG BIT,
    ORPHAN_FLAG VARCHAR(1)
)

CREATE NONCLUSTERED INDEX IX_SOURCE1_REFERENCE
ON #SOURCE1_REFERENCE (TARGET_DIMENSION, DIMENSION_CONTEXT, REFERENCE_COMPOSITE);
```

The last six columns of the previously presented staging table represent control columns that will support subsequent Reference pipe processing. Their meanings and their expected initial values here in the staging table are as follows:

- *Context ID*: The key to the dimension Context table that will be resolved from the Dimension Context value during the lookup resolution process. It can be initialized to NULL.
- *Definition ID*: The key of the dimension Definition table to which this entry eventually resolves. It can be initialized to NULL.
- *Master ID*: The Master ID of the dimension Definition table to which this entry eventually resolves. It can be initialized to NULL.
- *Placeholder Flag*: An indicator as to whether or not the entry is to be created new in the dimension. It can be initialized to *True*.
- *Unknown Flag*: An indicator as to whether or not errors were encountered in creating the Reference Composite that would require data using this key to be mapped to the Unknown row of the System subdimension of the associated dimension. It can be initialized to *False*.
- *Orphan Flag*: An indicator as to whether or not a new entry in the dimension needs to be added as an orphan to support one or more loading facts. It can be initialized to “N.”

You can create the staging table by including an Insert command just before the mult-tiered query you've been developing in the preceding text:

```
INSERT INTO #SOURCE1_REFERENCE
<<< multi-tiered metadata transformation query HERE >>>
```

Beta Limitations

There are two specific capabilities of the Reference pipe that are being omitted from the Beta version of the warehouse in this discussion. We'll defer them to the Gamma version development later in order to simplify the Beta version.

First, the aforementioned transformation join presumes that the Metadata dimension is not a slowly-changing dimension. You'll drop that assumption in the Gamma version, adding a source effective timestamp qualification to the join transformation. I do not recommend trying to add this capability to the Beta version because the nuances of slowly-changing dimensions are more difficult to plan and manage than unchanging dimensions, and the planning and development of metadata is generally a difficult activity for a novice data warehousing team. Trying to handle the metadata development and change planning at the same time overcomplicates the Beta version and can delay implementation by months. Also, slowly-changing dimension functionality is highly dependent upon the choice of correct and applicable source effective timestamps in every source dataset, something that often doesn't happen

during the sourcing of data for the Beta loads. These nuances have typically sorted themselves out through team experience by the time the Gamma development rolls around.

The second capability that hasn't been included in the aforementioned transformation is the capability to source columns other than source-originated natural keys, specifically Reference Composites, Master IDs, and Group IDs. As the data warehouse matures, there will be data to be loaded into the warehouse through the ETL that (in some way) were actually sourced from the warehouse. This will become particularly common as you start building measures and metrics, the values of which will be derived from clinical and operational facts already in the warehouse. In many of these cases, you'll want to source the data using the dimension keys already in the warehouse because it would be wasteful to resolve those keys during extraction back into their natural source keys only to have to resolve those natural keys back into their warehouse keys. You avoid all that processing by allowing this Reference pipe to accept the Reference Composites, Master IDs, and Group IDs directly. Sourcing Reference Composite values directly can simplify some source extraction and can improve the performance of some of the required Pivot operations in the ETL.

Both of these capabilities are much more complex than the code you're developing for the Beta version and would cause significant delays in your Beta implementation. You'll add them in the Gamma version when you'll have a bit more time, and you'll have gained the experience needed with this design to implement them correctly.

Resolve References

With all of your reference data staged and Reference Composites properly formatted, the next step is to resolve those references to their respective definitions in the dimensions. Other than during initial loading and during new master data updates, the majority of sourced keys in the staging tables will resolve to existing rows in the dimension. Since each dimension is a different set of physical tables, each dimension must be resolved separately, although the code convention is identical for each dimension. Here's an example for the Diagnosis dimension:

```
UPDATE #SOURCE1_REFERENCE
SET   CONTEXT_ID      = R.CONTEXT_ID,
      DEFINITION_ID = D.DEFINITION_ID,
      MASTER_ID      = D.MASTER_ID,
      ORPHAN_FLAG    = D.ORPHAN_FLAG
FROM
      #SOURCE1.REFERENCE S
      INNER JOIN DIAGNOSIS_C C ON S.DIMENSION_CONTEXT = C.COMPOSITE_LABEL
      LEFT JOIN DIAGNOSIS_R R ON C.CONTEXT_ID = R.CONTEXT_ID
      INNER JOIN DIAGNOSIS_D D ON R.MASTER_ID = D.MASTER_ID
```

```

    ON S.REFERENCE_COMPOSITE = R.REFERENCE_COMPOSITE
    AND S.SOURCE_TIMESTAMP
        BETWEEN D.EFFECTIVE_TIMESTAMP AND D.EXPIRATION_TIMESTAMP
WHERE
    S.TARGET_DIMENSION = 'Diagnosis';

```

The resolved reference now has the Context ID of the target context, the Master ID of the construct in the dimension, the Definition ID of the row in the Definition table that was active at the time of the source timestamp, and the Orphan Flag of the definition in the dimension. For rows that remain unresolved, the Context ID has been updated, which will simplify some of the following steps.

You'll need to repeat this code for each dimension within the model, including the Codeset. Although that might sound like a lot of code to have to repeat, I recommend doing it in that manner. I've had clients who didn't want to repeat the code 20–25 times, so they looked for fancy shortcuts that would allow them to do the dimension resolution with fewer commands. It's possible to accomplish a similar result using a very large query built using LEFT JOINS to each of the dimensions, taking advantage of a presumption that the contexts are all unique, so a query that looks at multiple dimensions at the same time can only resolve to one possible entry. This combined join can be made for multiple dimensions, but at a cost of higher complexity in the code. You would have to add Coalesce statements in the update assignments to pick up the correct result from the right dimension each time. Even if the single large-scale join were going to be more efficient—which nobody has ever demonstrated to me convincingly—I wouldn't recommend it just from the perspective of writing, testing, and maintaining such complex code.

In reality, the presumption that contexts are unique to only one dimension breaks down as you add more complicated data to your warehouse, although this might not occur during your loading of the Beta version. An example of a breakdown might include your placing of ICD-9 diagnosis codes in more than one dimension (e.g., Diagnosis and Procedure). When that happens, you'll be likely to load individual codes into more than one dimension. If so, coding a multidimensional lookup against LEFT JOINS to all of the dimensions will result in a Cartesian join when the Reference Composite values happen to contain any of these values that were loaded into multiple dimensions. Unless you write test code to specifically trap these conditions, your loads would be likely to create erroneous dimensionality around some of your data, or even to fail later in the load because the Master ID you seek to use for an update doesn't actually exist in the dimension you are attempting to update. These complications can be completely avoided by coding the lookup for each dimension separately.

The large-scale multidimensional join also has trouble with obtaining the Context ID since it has to become part of the LEFT JOIN rather than an INNER JOIN. As a separate dimensional query, the single-dimension update provides the Context ID even for unresolved rows, making the next steps much easier. Whatever way you decide to complete this step—one massive update or 25+ small

updates—you've now resolved all of the system sourced keys that already exist in the dimensions. The exceptions, those that remain unresolved, are handled next.

Unknown or Broken Keys

Of the composites in the Reference pipe that remain unresolved at this point, some of them might not be able to be resolved because they do not contain values for all of the required components in the target context. If one or more of the required elements of a multipart natural key were not present in the source data, there would be no way to recover that situation in order to create a legitimate key for resolution in the dimension.

You can update the staging area for the Reference pipe to reflect this situation by comparing the data in each Reference row that remains unresolved to the defining data for the natural key in the Context table:

```
UPDATE #SOURCE1_REFERENCE
SET   DEFINITION_ID = 2,
      MASTER_ID      = 2,
      UNKNOWN_FLAG   = 'Y',
      ORPHAN_FLAG    = 'N'
FROM
      #SOURCE1.REFERENCE S
      INNER JOIN DIAGNOSIS_C C ON S.CONTEXT_ID = C.CONTEXT_ID
WHERE
      S.MASTER_ID IS NULL
AND (
      (C.REFERENCE_LABEL_01 IS NOT NULL AND S.CONTEXT_KEY_01 IS NULL)
      OR(C.REFERENCE_LABEL_02 IS NOT NULL AND S.CONTEXT_KEY_02 IS NULL)
      OR(C.REFERENCE_LABEL_03 IS NOT NULL AND S.CONTEXT_KEY_03 IS NULL)
      OR(C.REFERENCE_LABEL_04 IS NOT NULL AND S.CONTEXT_KEY_04 IS NULL)
      OR(C.REFERENCE_LABEL_05 IS NOT NULL AND S.CONTEXT_KEY_05 IS NULL)
      OR(C.REFERENCE_LABEL_06 IS NOT NULL AND S.CONTEXT_KEY_06 IS NULL)
      OR(C.REFERENCE_LABEL_07 IS NOT NULL AND S.CONTEXT_KEY_07 IS NULL)
      OR(C.REFERENCE_LABEL_08 IS NOT NULL AND S.CONTEXT_KEY_08 IS NULL)
      OR(C.REFERENCE_LABEL_09 IS NOT NULL AND S.CONTEXT_KEY_09 IS NULL)
      OR(C.REFERENCE_LABEL_10 IS NOT NULL AND S.CONTEXT_KEY_10 IS NULL)
);

```

This query looks only at rows that are not yet resolved (Master ID is still NULL). It checks to see if there is a value in the Context Key for each Reference Label in the Context table. If a Reference Label is present, and the Context Key has not been sourced, the sourced key is considered broken. The update to the Reference pipe signals that the Unknown system row should be used in place of the dimension entry that would have been the intended target row.

Broken keys can be messy, so it's fortunate that they are very rare in most source systems. However, they do occur and must be handled. I've seen the occurrence of broken natural keys increase in recent years as more of our data

sources are coming from outside our own organizations in the form of XML-based data transmissions from other organizations. You need to be prepared for data quality problems associated with these datasets.

The use of the Unknown row in the System subdimension allows whatever data are being sourced into the warehouse (usually facts) to be loaded even though one or more natural keys for the data was incomplete. Resolution of the problem is a function of data governance, coupled with more investigation by the data warehouse support team. Here in the Beta version, the load will result in the broken natural key being discarded, and the information in the Data Control pipe can be used to trace the bad data back to its source. In the Gamma version, you'll implement automatic audit trails for selected control data. Keeping copies of broken keys is typically among the early audit controls implemented.

I've had developers at some clients who tried to insert the broken keys into the Reference table by substituting some form of arbitrary value (i.e., “~Missing~”) in the missing unsourced position in the Reference Composite. I strongly recommend *against* this approach. If you receive the same broken key twice in the same load, perhaps a five-part key with the third component missing, there is no way to determine whether the missing third component of each would have been the same if it had been present. Loading the broken key could easily create the illusion that the facts associated with each instance of the key were for the same dimension entry. There simply isn't any way to know that, and loading data into the warehouse that you know can be misinterpreted would be very bad for our data integrity. Marking both entries as Unknown is the safest approach, and we allow the audit trail or other data controls to keep track of what broken data was actually received. Eventually, the data will be corrected, and the appropriate dimension entries will be present and properly aligned with all of the original facts.

Alias Entry Collisions

An alias collision occurs when an alias entry in the Reference pipe is in conflict with either its primary Reference, or with another Alias for the same Reference. These situations typically arise when two different natural keys are meant to serve as aliases of each other; but before that connection can be signaled to the warehouse through the ETL, data are loaded for each separate key without any indication that the keys are aliases of each other. Each key will have a separate Reference table entry that points to a separate and independent Definition table entry.

A common example of alias collisions is the maintenance of multiple identifiers for the same patient as a result of receiving data from multiple electronic health record (EHR) systems. Suppose an institution has three separate and independent EHRs, each assigning a medical record number (MRN) to

a patient independent of any action in the other EHRs. As each of these sources sends data to the warehouse independently, each of the three MRNs will be loaded as separate identifiers for three different patients. Facts from each source system will be associated with the locally correct MRN; but the relationships among the three different MRNs will remain unknown to the warehouse, making it impossible to query all of the data for these patients in any consolidated manner.

Now suppose that one of those three EHRs is modified to contain alternative patient identifiers, including the two MRNs from the other EHRs. A new data source could be extracted from that EHR to establish the desired Alias relationships. The source transactions would send the EHR's main MRN as a Reference entry in the metadata, and the other two MRNs as Alias entries in the metadata. The transaction would effectively assert that the two extra MRNs are aliases of the first main MRN. This transaction is correct, but it will create a collision when this Reference pipe attempts to resolve those Reference Composite keys to their associated Master IDs. We want all three MRNs to resolve to the same patient row in the Definition table, but they'll actually resolve to three different patients because of the independence of the historical loading.

To see the collisions in the Reference pipe, query the data by transaction for all transactions that have more than one key in a dimension and for which those keys are not the same:

```

SELECT SDI_JOB_ID,
       PHYSICAL_DATASET,
       LOGICAL_DATASET,
       DATASET_TYPE,
       BREAK_ID,
       DATAFEED_BREAK,
       TARGET_DIMENSION,
       TARGET_SUBDIMENSION,
       COUNT(*),
       MAX(MASTER_ID) AS MAX_MASTER_ID,
       MIN(MASTER_ID) AS MIN_MASTER_ID
  FROM #SOURCE1_REFERENCE
 WHERE TARGET_TYPE IN ('Reference', 'Alias')
   AND DATASET_TYPE = 'Dimension'
   AND MASTER_ID IS NOT NULL
   AND MAX(MASTER_ID) <> MIN(MASTER_ID)
 GROUP BY SDI_JOB_ID,
          PHYSICAL_DATASET,
          LOGICAL_DATASET,
          DATASET_TYPE,
          BREAK_ID,
          DATAFEED_BREAK,
          TARGET_DIMENSION,
          TARGET_SUBDIMENSION
 HAVING COUNT(*) > 1;

```

The aforementioned query returns a set of transactions where the set of keys in a dimension resolved to more than one Master ID. In our example, the Count will be three because there are three keys in the pipe, and the Min and Max values will be the smallest and largest of the three MRNs. If the Min and Max Master IDs were the same, we'd know the different MRNs resolve to the same Definition row, and there would be no collision.

Note that if there is no collision, the Master ID for the transaction can be used to continue resolving Master IDs in the Reference pipe that weren't resolved by the previous joins to the dimensions:

```

UPDATE #SOURCE1_REFERENCE
SET   MASTER_ID      = A.MAX_MASTER_ID,
      PLACEHOLDER_FLAG = 'A' -- Alias reference for existing definition
FROM
      #SOURCE1.REFERENCE S
      INNER JOIN (
          SELECT SDI_JOB_ID,
                 PHYSICAL_DATASET,
                 LOGICAL_DATASET,
                 DATASET_TYPE,
                 BREAK_ID,
                 TARGET_DIMENSION,
                 TARGET_SUBDIMENSION,
                 COUNT(*),
                 MAX(MASTER_ID) AS MAX_MASTER_ID,
                 MIN(MASTER_ID) AS MIN_MASTER_ID
          FROM #SOURCE1_REFERENCE
          WHERE TARGET_TYPE IN ('Reference', 'Alias')
              AND DATASET_TYPE = 'Dimension'
              AND MASTER_ID IS NOT NULL
              AND MAX(MASTER_ID) = MIN(MASTER_ID)
          GROUP BY SDI_JOB_ID,
                  PHYSICAL_DATASET,
                  LOGICAL_DATASET,
                  DATASET_TYPE,
                  BREAK_ID,
                  TARGET_DIMENSION,
                  TARGET_SUBDIMENSION
          HAVING COUNT(*) > 1
      ) A
WHERE S.MASTER_ID IS NULL
    AND S.SDI_JOB_ID           = A.SDI_JOB_ID
    AND S.PHYSICAL_DATASET     = A.PHYSICAL_DATASET
    AND S.LOGICAL_DATASET      = A.LOGICAL_DATASET
    AND S.DATASET_TYPE         = A.DATASET_TYPE
    AND S.BREAK_ID              = A.BREAK_ID
    AND S.TARGET_DIMENSION      = A.TARGET_DIMENSION
    AND S.TARGET_SUBDIMENSION    = A.TARGET_SUBDIMENSION;

```

At this point in the processing, valid Alias transactions in the staging table that already resolve to the correct Master ID represent transactions that don't

require any extra processing. These transactions can optionally be deleted from the Reference pipe at this point, although you should do so only if it doesn't impact performance in any way:

```
DELETE FROM #SOURCE1_REFERENCE
WHERE PLACEHOLDER_FLAG = 'N'
AND TARGET_TYPE = 'Alias';
```

All subsequent processing continues to filter transactions using the Placeholder Flag, so choosing not to delete these transactions doesn't impact the continuing logic. Whether to delete these transactions from staging is your decision. My experience is that a lot of alias transactions are processed every day even though the majority of those alias entries are already in the warehouse. If the volume of these transactions causes performance degradation in subsequent processing (in excess of the overhead required to delete them), then delete them. If not, simply leave them in place in the staging table and subsequent processing will ignore them.

Unresolved References

At this point, the references that remain unresolved are all new entries in the dimensions. Inserting them will require creating surrogate Master IDs, and this process will work the same as the processes we used in the Alpha version that were based on the New Surrogates table that would assign the next surrogate identifier to any Composite key loaded into it:

```
CREATE TABLE NEW_SURROGATES (
    PROCESS      VARCHAR(64),
    DIMENSION    VARCHAR(20),
    SUBDIMENSION VARCHAR(64),
    CONTEXT_ID   INT,
    COMPOSITE    VARCHAR(772),
    NEW_SURROGATE INT NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (NEW_SURROGATE)
);
ALTER TABLE NEW_SURROGATES AUTO_INCREMENT = 10001;
```

The data at this point in the Reference pipe are transactional, meaning that each row represents an individual source transaction identified by the source Break ID. Any new natural keys in the source reference data are likely to occur many times, so a separate set of distinct identifiers needs to be extracted from the main pipe. The new staging table will allow those distinct

identifiers to be processed separately from the transactional detail in the main Reference pipe:

```
CREATE TABLE #UNRESOLVED_REFERENCE (
    TARGET_DIMENSION      VARCHAR(64),
    TARGET_CATEGORY        VARCHAR(64),
    TARGET_TYPE            VARCHAR(64),
    DIMENSION_CONTEXT     VARCHAR(64),
    CONTEXT_ID             INT,
    REFERENCE_COMPOSITE   VARCHAR(772),
    CONTEXT_KEY_01          VARCHAR(64),
    CONTEXT_KEY_02          VARCHAR(64),
    CONTEXT_KEY_03          VARCHAR(64),
    CONTEXT_KEY_04          VARCHAR(64),
    CONTEXT_KEY_05          VARCHAR(64),
    CONTEXT_KEY_06          VARCHAR(64),
    CONTEXT_KEY_07          VARCHAR(64),
    CONTEXT_KEY_08          VARCHAR(64),
    CONTEXT_KEY_09          VARCHAR(64),
    CONTEXT_KEY_10          VARCHAR(64),
    DEFINITION_ID           INT,
    MASTER_ID               INT,
    PLACEHOLDER_FLAG        BIT,
    UNKNOWN_FLAG             BIT,
    ORPHAN_FLAG              VARCHAR(1)
);

CREATE NONCLUSTERED INDEX IX_UNRESOLVED_REFERENCE
ON #UNRESOLVED_REFERENCE (TARGET_DIMENSION, DIMENSION_CONTEXT, REFERENCE_COMPOSITE);
```

Distinct Composites

You'll populate this staging table by selecting all of the distinct source key combinations from the Reference pipe that are still unresolved, as indicated by the Master ID still being NULL:

```
INSERT INTO #UNRESOLVED_REFERENCE
SELECT DISTINCT
    S.TARGET_DIMENSION,
    S.TARGET_SUBDIMENSION,
    S.TARGET_TYPE,
    S.DIMENSION_CONTEXT,
    S.CONTEXT_ID,
    S.REFERENCE_COMPOSITE,
    S.CONTEXT_KEY_01,
    S.CONTEXT_KEY_02,
    S.CONTEXT_KEY_03,
    S.CONTEXT_KEY_04,
    S.CONTEXT_KEY_05,
    S.CONTEXT_KEY_06,
    S.CONTEXT_KEY_07,
    S.CONTEXT_KEY_08,
```

```

S.CONTEXT_KEY_09,
S.CONTEXT_KEY_10,
S.DEFINITION_ID,
S.MASTER_ID,
S.PLACEHOLDER_FLAG,
S.UNKNOWN_FLAG,
S.ORPHAN_FLAG
FROM
#SOURCE1_REFERENCE S
WHERE S.MASTER_ID IS NULL

```

The DISTINCT clause in this operation is critically important because it ensures only one new surrogate key is generated for each distinct Reference Composite in each context. The reference staging table could easily contain hundreds or thousands of rows for each of those keys, so not using the distinct operation when generating keys could result in a massive insertion of the Cartesian set of rows generated. Since each of these rows would end up having a legitimately assigned surrogate, the error would be very difficult to spot in the dataset, and future fact loads would also Cartesian join to all of those rows, further exacerbating the error.

Surrogate Assignments

Having isolated the distinct Reference Composites not yet resolved to the associated dimension, you can insert all of those unresolved nonalias source keys into the New Surrogates table:

```

DELETE FROM NEW_SURROGATES WHERE PROCESS = 'Beta Unresolved'
INSERT INTO NEW_SURROGATES ( PROCESS, TARGET_DIMENSION,
                             TARGET_SUBDIMENSION,
                             CONTEXT_ID, COMPOSITE)
SELECT DISTINCT
      'Beta Unresolved',
      U.TARGET_DIMENSION,
      U.TARGET_SUBDIMENSION,
      U.CONTEXT_ID,
      U.REFERENCE_COMPOSITE
  FROM #UNRESOLVED_REFERENCE U
 WHERE TARGET_TYPE IN ('Reference', 'Hierarchy')

```

It's important not to include Alias data in this operation since, by definition, the alias keys should not have their own definition rows built for them. Any remaining unresolved Master IDs for alias transactions can be resolved after their corresponding Reference rows have been resolved. Having inserted the distinct

composites into the New Surrogates table, you can then immediately bring those new surrogates back to the unresolved staging table:

```
UPDATE #UNRESOLVED_REFERENCE
SET
    DEFINITION_ID      = NS.NEW_SURROGATE,
    MASTER_ID          = NS.NEW_SURROGATE,
    PLACEHOLDER_FLAG   = 'Y',
    ORPHAN_FLAG        = 'Y'
FROM
    #UNRESOLVED_REFERENCE U
    INNER JOIN NEW_SURROGATE NS
        ON NS.PROCESS = 'Beta Unresolved'
        AND U.CONTEXT_ID = NS.DIMENSION_CONTEXT
        AND U.REFERENCE_COMPOSITE = NS.REFERENCE_COMPOSITE
```

The initial update sets the Orphan Flag for each new surrogate to Yes, so you'll need to confirm the situations where that is correct and change the flag to No when it is not correct. The reference will result in an orphan if the only sources needing that reference were fact loads. To confirm orphanness, you'll turn off the Orphan Flag for any reference that appears in at least one sourced dimension load:

```
UPDATE #UNRESOLVED_REFERENCE
SET ORPHAN_FLAG = 'N'
FROM #UNRESOLVED_REFERENCE U
INNER JOIN (
    SELECT DISTINCT CONTEXT_ID, REFERENCE_COMPOSITE
    FROM #SOURCE1_REFERENCE
    WHERE DIMENSION_ID IS NULL
        AND DATASET_TYPE = 'Dimension'
) D
WHERE U.CONTEXT_ID          = D.CONTEXT_ID
    AND U.REFERENCE_COMPOSITE = D.REFERENCE_COMPOSITE
```

At this point in the processing, you aren't concerned with the details of the Dimension load that uses any particular Reference Composite value. You only need to be concerned that, as long as there is at least one dimension load source referencing the Master ID in question, the resulting new Definition in the dimension would not be an orphan.

Transactional Redistribution

With the new Master IDs assigned, and the Orphan Flag set appropriately, you'll now redistribute these new surrogate Master IDs across the deeper transaction data in the full reference staging table:

```

UPDATE #SOURCE1_REFERENCE
SET
    DEFINITION_ID      = NS.NEW_SURROGATE,
    MASTER_ID          = NS.NEW_SURROGATE,
    PLACEHOLDER_FLAG   = 'Y',
    ORPHAN_FLAG        = U.ORPHAN_FLAG
FROM
    #SOURCE1_REFERENCE R
    INNER JOIN NEW_SURROGATE NS
        ON NS.PROCESS = 'Beta Unresolved'
        AND R.CONTEXT_ID = NS.CONTEXT_ID
        AND R.REFERENCE_COMPOSITE = NS.REFERENCE_COMPOSITE
    INNER JOIN #UNRESOLVED_REFERENCE U
        ON R.CONTEXT_ID = U.CONTEXT_ID
        AND R.REFERENCE_COMPOSITE = U.REFERENCE_COMPOSITE

```

As a result of this processing, every Reference and Hierarchy entry in the reference staging table now has an appropriate Master ID for further processing. Alias rows have a key defined if they were found on the dimension Reference table but are NULL otherwise. The various control values in the staging table have also been set to support processing flow in the Definition pipe in the next chapter.

Alias Propagation

Because the redistribution confirms that all Reference entries now have their Master ID assigned, those Master IDs can be propagated to the remaining Alias entries in the Reference pipe that still have NULL Master IDs:

```

UPDATE #SOURCE1_REFERENCE
    SET MASTER_ID = R.MASTER_ID,
FROM #SOURCE1.REFERENCE A
    INNER JOIN #SOURCE1.REFERENCE R
        ON (S.SDI_JOB_ID           = A.SDI_JOB_ID
            AND S.PHYSICAL_DATASET = A.PHYSICAL_DATASET
            AND S.LOGICAL_DATASET  = A.LOGICAL_DATASET
            AND S.DATASET_TYPE     = A.DATASET_TYPE
            AND S.BREAK_ID         = A.BREAK_ID
            AND S.TARGET_DIMENSION = A.TARGET_DIMENSION
            AND S.TARGET_UBDIMENSION = A.TARGET_SUBDIMENSION)
    WHERE S.TARGET_TYPE = 'Alias'
        AND R.TARGET_TYPE = 'Reference'
        AND S.MASTER_ID IS NULL
        AND R.MASTER_ID IS NOT NULL

```

At this point in the Reference pipe, there are no entries remaining without existing or newly assigned Master IDs. All of the processing to this point has also been in memory or in staging tables. No transactions have been

committed to the warehouse database yet, providing for an effective checkpoint when planning testing activities.

Reference Entries

Any dimensional keys that were assigned new Master IDs now need to be inserted into the data warehouse. This will entail new rows in the Reference, Bridge, Group, and Hierarchy tables, as follows:

New References

Each new surrogate generated in the New Surrogate table represents a new entry in the dimension Reference table. The code is dimension specific, so it will need to be repeated for each dimension. Here is an example for inserting new Reference entries in the Subject dimension:

```
INSERT INTO SUBJECT_R
  (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE,
   CONTEXT_KEY_01, CONTEXT_KEY_02, CONTEXT_KEY_03, CONTEXT_KEY_04, CONTEXT_KEY_05,
   CONTEXT_KEY_06, CONTEXT_KEY_07, CONTEXT_KEY_08, CONTEXT_KEY_09, CONTEXT_KEY_10)
SELECT NS.CONTEXT_ID,
      NS.NEW_SURROGATE,
      U.REFERENCE_COMPOSITE,
      U.CONTEXT_KEY_01,
      U.CONTEXT_KEY_02,
      U.CONTEXT_KEY_03,
      U.CONTEXT_KEY_04,
      U.CONTEXT_KEY_05,
      U.CONTEXT_KEY_06,
      U.CONTEXT_KEY_07,
      U.CONTEXT_KEY_08,
      U.CONTEXT_KEY_09,
      U.CONTEXT_KEY_10
  FROM NEW_SURROGATES NS
 INNER JOIN #UNRESOLVED_REFERENCE U
   ON NS.COMPOSITE = U.REFERENCE_COMPOSITE
 WHERE NS.PROCESS = 'Beta Unresolved'
   AND NS.TARGET_DIMENSION = 'Subject';
```

Alias Entries

The last set of sourced Reference pipe transactions involves the definition of alias entries for existing natural keys (e.g., Target Type is “Alias”). Every alias transaction actually involves two of the Reference entries processed earlier, one serving as the foundational or target key, and the other serving as the alias key. These two entries in the reference staging table will need to be joined together in a single row in order to be inserted into the Reference table.

Table 10.5 Example Alias Metadata

Context Key 4	Context Key 5	Context Key 6	Target Type	Target Dimension	Target Context	Required Flag
MRN	Subject R	Context key 01	Reference	Subject	MRN	Y
PersonID	Subject R	Context key 01	Alias	Subject	PersonID	Y
SSN	Subject R	Context key 01	Alias	Subject	SSN	N

Table 10.5 illustrates some metadata that defines two alias types in the Subject dimension that might be available in some source systems. The first row of the table shows the main primary reference entry for a Subject, the MRN. The next two rows show additional aliases that a Subject might be known by the PersonID and the SSN. The PersonID is defined as a required alias, and the SSN is defined as an optional alias.

To process alias data, the subset of data in the Reference pipe associated with these metadata must be joined together so the Alias entries are paired up with their target Reference entries. The entries of interest here are those where the target Reference resolved to an entry in the dimension, and the Alias entry did not resolve. These are the entries that need to be inserted:

```

INSERT INTO SUBJECT_R
  (CONTEXT_ID, MASTER_ID, REFERENCE_COMPOSITE,
  CONTEXT_KEY_01, CONTEXT_KEY_02, CONTEXT_KEY_03, CONTEXT_KEY_04, CONTEXT_KEY_05,
  CONTEXT_KEY_06, CONTEXT_KEY_07, CONTEXT_KEY_08, CONTEXT_KEY_09, CONTEXT_KEY_10)
SELECT D.CONTEXT_ID
  A.MASTER_ID,
  D.REFERENCE_COMPOSITE,
  D.CONTEXT_KEY_01,
  D.CONTEXT_KEY_02,
  D.CONTEXT_KEY_03,
  D.CONTEXT_KEY_04,
  D.CONTEXT_KEY_05,
  D.CONTEXT_KEY_06,
  D.CONTEXT_KEY_07,
  D.CONTEXT_KEY_08,
  D.CONTEXT_KEY_09,
  D.CONTEXT_KEY_10
FROM #SOURCE1_REFERENCE A
INNER JOIN #SOURCE1_REFERENCE D
  ON A.JOB_ID
  AND A.PHYSICAL_DATASET
  AND A.LOGICAL_DATASET
  AND A.DATASET_TYPE
  AND A.BREAK_ID
  AND A.TARGET_DIMENSION
WHERE A.TARGET_DIMENSION
  and A.TARGET_TYPE
  AND A.UNKNOWN_FLAG
  AND A.MASTER_ID
  AND D.TARGET_TYPE
  AND D.UNKNOWN_FLAG
  AND D.MASTER_ID
  AND D.PLACEHOLDER_FLAG
      = D.JOB_ID
      = D.PHYSICAL_DATASET
      = D.LOGICAL_DATASET
      = D.DATASET_TYPE
      = D.BREAK_ID
      = D.TARGET_DIMENSION
      = 'Subject'
      = 'Reference'
      = False
      IS NOT NULL
      = 'Alias'
      = False
      IS NULL
      = 'Y';

```

Of the entries that are not inserted by this logic, there is the possibility of some errors occurring that will need to be handled later. In many cases, both the Reference and Alias entries resolve to the dimension. If they resolved to the same Master ID, it means that the alias was already present in the dimension and no further action was needed. However, if they each resolved to different Master IDs, it means that the alias already exists in the dimension for a different target entry. This is an error that we are ignoring in the Beta version, and we will handle it in the Gamma version where we add extensive error checking logic to the various pipes. If the analyst who defines the metadata takes time to avoid defining aliases for source data that isn't guaranteed to be unique, this error should not occur during Beta loading. If the error occurs, the rows are dropped by this Beta version code.

Bridges and Groups

In addition to new Reference rows, the standard default single-bridge groups also need to be inserted into each dimension. Here are the insertions for the Subject dimension:

```
INSERT INTO SUBJECT_G
  (GROUP_ID,
   GROUP_COMPOSITE)
SELECT NS.NEW_SURROGATE,
      NS.NEW_SURROGATE + '+Subject'+1+1.0'
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = 'Beta Unresolved'
   AND NS.TARGET_DIMENSION = 'Subject'

INSERT INTO SUBJECT_B
  (GROUP_ID,
   MASTER_ID,
   ROLE, RANK, WEIGHT)
SELECT NS.NEW_SURROGATE, MS.
      NEW_SURROGATE, 'Subject', 1, 1.0
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = 'Beta Unresolved'
   AND NS.TARGET_DIMENSION = 'Subject';
```

Hierarchy Entries

There are several types of hierarchy entries that must be created in this Reference pipe, including self-referencing entries for any new Master IDs created, sourced fiat hierarchy transactions present in the pipeline, and natural hierarchy impacts from new Master ID natural keys mingling with each other and previously present data.

New Self-Hierarchies

Each new Reference entry requires a new self-referencing hierarchy entry into each dimension. Here is the insertion for the Subject dimension:

```
INSERT INTO SUBJECT_H
  (ANCESTOR_MASTER_ID,
   DESCENDENT_MASTER_ID,
   PERSPECTIVE, DEPTH_FROM_ANCESTOR)
SELECT NS.NEW_SURROGATE,
       NS.NEW_SURROGATE, '~Self~', 0
  FROM NEW_SURROGATES NS
 WHERE NS.PROCESS = 'Beta Unresolved'
   AND NS.TARGET DIMENSION = 'Subject';
```

Fiat Hierarchies

With Reference entries confirmed or inserted, we now turn our attention to the subset of the data in the Reference pipe that is targeting the definition of fiat hierarchies (e.g., target type is “Hierarchy”). Every fiat hierarchy transaction actually involves two of the Reference entries processed earlier, one serving as the ancestor in the hierarchy entry and one serving as the descendent. These two entries in the reference staging table will need to be joined together in a single row in order to be inserted into the dimension hierarchy table:

```
SELECT A.TARGET_DIMENSION,
       A.MASTER_ID           AS ANCESTOR_MASTER_ID,
       D.MASTER_ID           AS DESCENDENT_MASTER_ID
      ,A.HIERARCHY_CLASS
      ,A.HIERARCHY_PERSPECTIVE
      ,A.HIERARCHY_RELATION
      ,A.HIERARCHY_ORIGIN_FLAG
      ,A.HIERARCHY_DEPTH_FROM_ANCESTOR
      ,A.HIERARCHY_TERMINUS_FLAG
  FROM #SOURCE1_REFERENCE A
 INNER JOIN #SOURCE1_REFERENCE D
    ON A.SDI_JOB_ID          = D.SDI_JOB_ID
   AND A.PHYSICAL_DATASET     = D.PHYSICAL_DATASET
   AND A.LOGICAL_DATASET      = D.LOGICAL_DATASET
   AND A.DATASET_TYPE        = D.DATASET_TYPE
   AND A.BREAK_ID             = D.BREAK_ID
   AND A.TARGET_DIMENSION     = D.TARGET_DIMENSION
   AND A.HIERARCHY_CLASS      = D.HIERARCHY_CLASS
   AND A.HIERARCHY_PERSPECTIVE = D.HIERARCHY_PERSPECTIVE
   AND A.HIERARCHY_RELATION    = D.HIERARCHY_RELATION
   AND A.HIERARCHY_ORIGIN_FLAG = D.HIERARCHY_ORIGIN_FLAG
```

```

AND A.HIERARCHY_DEPTH_FROM_ANCESTOR = D.HIERARCHY_DEPTH_FROM_ANCESTOR
AND A.HIERARCHY_TERMINUS_FLAG = D.HIERARCHY_TERMINUS_FLAG
WHERE A.TARGET_TYPE = 'Hierarchy'
    AND A.HIERARCHY_ROLE = 'Ancestor'
    AND A.UNKNOWN_FLAG = False
    AND A.MASTER_ID IS NOT NULL
    AND D.TARGET_TYPE = 'Hierarchy'
    AND D.HIERARCHY_ROLE = 'Descendent'
    AND D.UNKNOWN_FLAG = False
    AND D.MASTER_ID IS NOT NULL;

```

The only errors that might be expected when processing flat hierarchy transactions would be if one or both of the ancestor or descendants ultimately resolve to the Unknown row in the dimension because of a problem with the source natural keys. Here in the Beta version, those transactions are skipped. The Gamma version will include error checking components that will trap and record those errors. If the analyst who defines the metadata and source intake logic for each data source is careful to avoid errors in source mapping and if the data source columns are typically correct, these errors should not occur in this Beta version.

Natural Hierarchies

The last piece of processing in the Reference pipe is to update any natural hierarchies that occur in the dimension that might have been impacted by the creation of new entries in the dimensions. There are two types of natural hierarchies in the warehouse that need separate processing because of differences in the columns involved. The first, and most common, is a natural hierarchy that occurs among dimension entries that share similar components in their multipart Reference Composite values. To share a component across two contexts, you need to see that the Key Label columns in the Context tables match and that the corresponding Context Key columns in the Reference table match.

For example, let's revisit the Miami, Florida data discussed earlier in this chapter. The city row in Geopolitics for this city is a natural hierarchy descendent of the state row for Florida, as well as a descendent of the country row for the United States. Tables 10.6 and 10.7 illustrate why this statement is true.

Table 10.6 Example Geopolitics Contexts

Context ID	Context Name	Key Label 01	Key Label 02	Key Label 03
1	Country	Country code		
2	State	Country code	State abbreviation	
3	City	Country code	State abbreviation	City name

Table 10.7 Example Geopolitics Reference Data

Context ID	Reference Composite	Context Key 01	Context Key 02	Context Key 03
1	USA	USA		
2	USA FL	USA	FL	
3	USA FL Miami	USA	FL	Miami

Table 10.6 shows three hypothetical contexts in the Geopolitics dimension related to our data. We know the Reference Composites across these three contexts form a natural hierarchy because they share the same data in portions of their definition. All three use the Country Code in Context Key 01, making the State and City data descendants of the Country data. In addition, both the State and City contexts share the State Abbreviation in Context Key 02, making the City a descendent of the State. At this point, you're not looking at the data yet, so you call the relationships defined here natural hierarchy *candidates*—the contexts are candidates for natural hierarchy relationships.

Next, we look at actual Reference Composite values in the entries of the Reference table, but we'll limit your comparisons to Reference Composites that are in contexts that have already been identified as candidates for natural hierarchy participation. Table 10.7 illustrates the three hypothetical rows we have in our dimension for Miami, Florida.

```

INSERT ANCESTOR_MASTER_ID,
       DESCENDENT_MASTER_ID,
       CLASS,
       PERSPECTIVE,
       RELATION,
       ORIGIN_FLAG,
       DEPTH_FROM_ANCESTOR,
       TERMINUS_FLAG,
       STATUS_INDICATOR,
       EFFECTIVE_TIMESTAMP,
       EXPIRATION_TIMESTAMP,
       SOURCE_TIMESTAMP
               INTO GEOPOLITICS_R
SELECT PR.CONTEXT_KEY
      AS ANCESTOR_MASTER_ID,
      DR.CONTEXT_KEY
      AS DESCENDENT_MASTER_ID,
      'Hierarchy'
      AS CLASS,
      'Natural'
      AS PERSPECTIVE,
      'Natural'
      AS RELATION,
      ORIGIN_FLAG,
      DEPTH_FROM_ANCESTOR,
      'X'
      AS TERMINUS_FLAG,
      'A'
      AS STATUS_INDICATOR,
      '1900-01-01'
      AS EFFECTIVE_TIMESTAMP,
      '2300-12-31'
      AS EXPIRATION_TIMESTAMP,
      @SYSDATE
      AS SOURCE_TIMESTAMP
FROM GEPOLITICS_R PR,
     GEPOLITICS_R DR,
     (SELECT PC.CONTEXT_ID PID,
            DC.CONTEXT_ID DID,
            (LEN(DC.COMPOSITE_LABEL) - LEN(REPLACE(DC.COMPOSITE_LABEL, '|', '')))
            - (LEN(PC.COMPOSITE_LABEL) - LEN(REPLACE(PC.COMPOSITE_LABEL, '|', '')))
            AS DEPTH_FROM_ANCESTOR

```

```

CASE WHEN LEN(PC.COMPOSITE_LABEL)
      - LEN(REPLACE(PC.COMPOSITE_LABEL, '|', '')) = 0
      THEN 'Y' ELSE 'N' END ORIGIN_FLAG
FROM GEPOLITICS_C P,
     GEPOLITICS_C D
WHERE PC.COMPOSITE_LABEL <> DC.COMPOSITE_LABEL
      AND DC.COMPOSITE_LABEL LIKE PC.COMPOSITE_LABEL || '|%''
      AND PC.NATURAL_HIERARCHY_CROSSOVER_FLAG = 'Y'
      AND DC.NATURAL_HIERARCHY_CROSSOVER_FLAG = 'Y') PAIRS
WHERE PR.CONTEXT_ID = PAIRS.PID
      AND DR.CONTEXT_ID = PAIRS.DID
      AND DR.REFERENCE_COMPOSITE LIKE PR.REFERENCE_COMPOSITE || '|%';

```

Some of the logic in this Select statement might not be intuitively obvious, so let's take a closer look at a couple of the core elements. The first calculates the Depth from Ancestor value for the generated new hierarchy entry:

```

(LEN(DC.COMPOSITE_LABEL) - LEN(REPLACE(DC.COMPOSITE_LABEL, '|', '')))
      - (LEN(PC.COMPOSITE_LABEL) - LEN(REPLACE(PC.COMPOSITE_LABEL, '|', '')))
AS DEPTH_FROM_ANCESTOR,

```

This code effectively counts pipe delimiters in both candidate contexts, taking advantage of a commonly used coding convention for determining the number of times a specific character occurs within a variable: Take the length of the value in the variable, and subtract the length of the same variable where the character of interest has been removed. In this case, that calculation yields the number of pipe delimiters that are present in the relevant Composite Label. The difference between these two values is the value of the Depth from Ancestor for the new candidate hierarchy entry being generated. The second code snippet of interest involves the determination of a value for the Origin Flag for the generated hierarchy entry:

```

CASE WHEN LEN(PC.COMPOSITE_LABEL) - LEN(REPLACE(PC.COMPOSITE_LABEL, '|', '')) = 0
      THEN 'Y' ELSE 'N' END ORIGIN_FLAG

```

This code uses the same conventional approach to counting the pipe delimiters in the Composite Label of the candidate ancestor. If there are no pipe delimiters, the candidate origin is an origin node in the Perspective, and the Origin Flag is set accordingly. Note that the Terminus Flag cannot be set in the same way because there isn't enough information in the generated candidate hierarchy entry to determine whether or not the descendent node is a terminus node in the Perspective. The "X" serves as a marker for the resolution logic that is described in the following text.

The second type of natural hierarchy supported in this Reference pipe is for contexts that involve one-part keys (i.e., Reference Composite = Context Key 01), where one entry is a subset of another. An example is ICD-9 code 384.2 that is a subset of ICD-9 code 384, making 384 a natural hierarchy parent of the child 384.2 code.

```

INSERT ANCESTOR_MASTER_ID,
      DESCENDENT_MASTER_ID,
      CLASS,
      PERSPECTIVE,

```

```

RELATION,
ORIGIN_FLAG,
DEPTH_FROM_ANCESTOR,
TERMINUS_FLAG,
STATUS_INDICATOR,
EFFECTIVE_TIMESTAMP,
EXPIRATION_TIMESTAMP,
SOURCE_TIMESTAMP      INTO GEOPOLITICS_R
SELECT PR.MASTER_ID      AS ANCESTOR_MASTER_ID,
DR.MASTER_ID      AS DESCENDENT_MASTER_ID,
'Hierarchy'        AS CLASS,
'Natural'          AS PERSPECTIVE,
'Natural'          AS RELATION,
CASE WHEN LEN(PR.REFERENCE_COMPOSITE)
- LEN(REPLACE(PR.REFERENCE_COMPOSITE,NATURAL_HIERARCHY_INTERNAL_DELIMITER,'')) = 0
    THEN 'Y' ELSE 'N' END ORIGIN_FLAG,
(LEN(DR.REFERENCE_COMPOSITE)
- LEN(REPLACE(DR.REFERENCE_COMPOSITE,NATURAL_HIERARCHY_INTERNAL_DELIMITER,'')) )
- (LEN(PR.REFERENCE_COMPOSITE)
- LEN(REPLACE(PR.REFERENCE_COMPOSITE,NATURAL_HIERARCHY_INTERNAL_DELIMITER,''))) AS DEPTH_FROM_ANCESTOR,
'X'                  AS TERMINUS_FLAG,
'A'                  AS STATUS_INDICATOR,
'1900-01-01'          AS EFFECTIVE_TIMESTAMP,
'2300-12-31'          AS EXPIRATION_TIMESTAMP,
NULL                 AS SOURCE_EFFECTIVE_TIMESTAMP
FROM GEOPOLITICS_R PR,
GEOPOLITICS_R DR,
GEOPOLITICS_C C
WHERE C.NATURAL_HIERARCHY_INTERNAL_FLAG = 'Y'
AND PR.CONTEXT_ID = C.CONTEXT_ID
AND DR.CONTEXT_ID = C.CONTEXT_ID
AND DR.REFERENCE_COMPOSITE LIKE CONCAT(PR.REFERENCE_COMPOSITE,
NATURAL_HIERARCHY_INTERNAL_DELIMITER, '%');

```

Note that in the sample code, the Origin Flag and Depth from Ancestor values are established using the same approach as with the cross-column natural hierarchy builder. The Terminus Flag is set to “X” as a marker for the resolution logic to follow as well.

Terminus Resolution

Once all candidate hierarchy entries have been generated and loaded, the Terminus Flags that have been set to “X” can be resolved. This involves a simple left self-join between each dimension’s Hierarchy table and itself to see if the Descendent in each entry ever appears as an Ancestor in another entry. If it does, the node is not a terminus of the hierarchy in that perspective. If it doesn’t, a NULL will be the returned from the LEFT JOIN for the ancestor data and the flag can be turned on:

```

UPDATE GEOPOLITICS_H
SET TERMINUS_FLAG = CASE
    WHEN D.ANSCESTOR_MASTER_ID IS NULL
        THEN 'Y' ELSE 'N' END
FROM GEOPOLITICS_H A
LEFT JOIN GEOPOLITICS_H D

```

```
WHERE A.DESCENDENT_MASTER_ID = D.ANCESTOR_MASTER_ID
AND A.CLASS = D.CLASS
AND A.PERSPECTIVE = D.PERSPECTIVE
AND A.RELATION = D.RELATION
AND A.TERMINUS_FLAG = 'X';
```

This strategy for updating the Terminus Flag exposes some risk if it is executed too early in the loading of the dimension. If the deepest nodes of a hierarchy haven't been loaded yet, running this update will turn on the Terminus Flag for entries that might ultimately receive descendants on subsequent loads. To avoid this, run this update only after all of the hierarchies have been loaded into the table. Alternatively, something I would advise in the long run would be to remove the qualification for an "X" value from the WHERE clause and allow the update to span the entire table. Work with your support team and database administrators if a full table process creates performance concerns, although I've never had a serious problem with that. If necessary, qualify the broader query to process only certain Perspectives that might be larger than others (e.g., SNOMED).

Fiat Hierarchy Cascade

The last step in the processing of fiat hierarchies is to cascade the relations that have been loaded to include all implied ancestor–descendant relations, as shown in Figure 10.3. To identify candidate cascade hierarchy entries, look for nodes

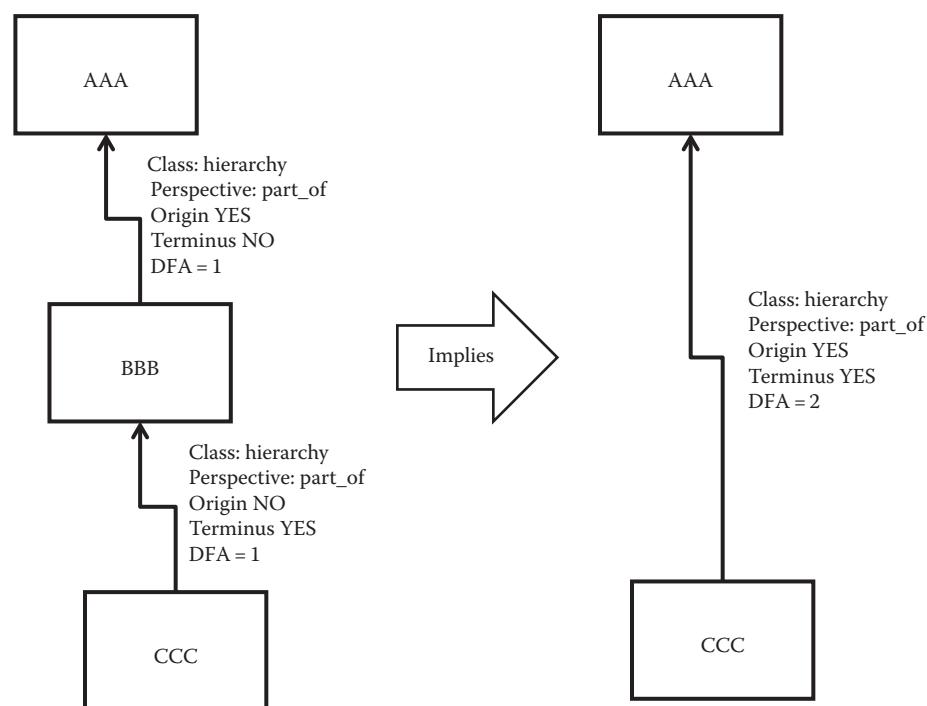


Figure 10.3 Example implied cascade of multilevel hierarchic entries.

with Master IDs that appear as both ancestor and descendent keys in two hierarchy entries having the same Class, Perspective, and Relation. Finding them implies a new broader hierarchy entry in that Class, Perspective, and Relation in which the Origin Flag is taken from the ancestor component, the Terminus Flag is taken from the descendent component, and the Depth from Ancestor is the sum of the two Depth from Ancestor values:

```

SELECT A.ANCESTOR_MASTER_ID      AS ANCESTOR_MASTER_ID,
       D.DESCENTENT_MASTER_ID   AS DESCENTENT_MASTER_ID,
       A.CLASS                  AS CLASS,
       A.PERSPECTIVE             AS PERSPECTIVE,
       A.RELATION                AS RELATION,
       A.ORIGIN_FLAG              AS ORIGIN_FLAG,
       A.DEPTH_FROM_ANCESTOR     AS DEPTH_FROM_ANCESTOR,
       + D.DEPTH_FROM_ANCESTOR    AS DEPTH_FROM_ANCESTOR,
       D.TERMINUS_FLAG            AS TERMINUS_FLAG,
       D.STATUS_INDICATOR          AS STATUS_INDICATOR,
       D.EFFECTIVE_TIMESTAMP      AS EFFECTIVE_TIMESTAMP,
       D.EXPIRATION_TIMESTAMP     AS EXPIRATION_TIMESTAMP,
       D.SOURCE_TIMESTAMP          AS SOURCE_EFFECTIVE_TIMESTAMP
  FROM GEOPOLITICS_H A,
       GEOPOLITICS_H D
 WHERE A.DESCENTENT_MASTER_ID = D.ANCESTOR_MASTER_ID
   AND A.CLASS      = D.CLASS
   AND A.PERSPECTIVE = D.PERSPECTIVE
   AND A.RELATION   = D.RELATION;

```

This cascade strategy is subject to the same postload risks as the Terminus resolution strategy. Building implied entries, or inferring the status of entries by looking at others, carries the risk that those implications or inferences might be invalidated by subsequent maintenance of the associated data. This is particularly the case when dealing with data in dimensions that can be managed as slowly-changing data. The sample code presented here presumes minimal impact of these changes, expecting that changes to one node in an entry will be mirrored by changes to the other nodes. While this condition isn't guaranteed to be met, it often is.

In my years of doing this activity in this way, I've never run into a data crisis as a result. Fortunately, most often hierarchies change by adding new nodes and relations, less often changing relations among existing nodes, but it can happen. For highly volatile hierarchies, the easiest way to correct any problems encountered is to delete the cascading entries in the hierarchy (e.g., having Depth from Ancestor of 2 or more) and allow this algorithm to rebuild correct ones. There are some conditions where even this fix won't realign overlapping or orthogonal effective expiration periods in the data, so the Gamma version will add some capability for handling that rare condition without human intervention.

This concludes the processing required for the Reference pipe. Four of the six tables in the dimension design pattern are now processed completely: Reference, Hierarchy, Bridge, and Group. Orphan rows have also been created in the Definition table. These tables and entries are all dependent on the Master ID for the Definition data built in the next chapter to be properly accessible and aligned with the Reference Composite of natural keys arriving from each source. We can now turn to those Definition entries: the core of each dimension.

Chapter 11

ETL Definition Pipe

The Definition pipe focuses exclusively on the Definition table in each dimension. The Reference pipe has already taken care of all the other table constructs in the star design pattern. What remains is the complex logic of managing change in the definition entries themselves, that complexity taking two forms in the Definition pipe. The high-level sequence diagram in [Figure 11.1](#) illustrates the main steps in the processing. Specifically, the processing covered in the Definition pipe includes

- Transforming source-only intake data into source-to-target staged data through a join to the metadata dimension, including codeset translation of nonstandard source values
- Pivoting Definition data columns by Master ID to obtain wide rows for insertion or update in each dimension
- Marrying the generic wide rows to their respective dimension identifiers from the Reference pipe
- Applying data to each dimension Definition table as new inserts (including orphans), auto-adoption of existing orphans, and data updates (including slowly-changing dimension [SCD] controls)

Processing Complexities

Definition processing in the dimensions is more complex than the Reference processing described in Chapter 10. Definition handling is more complex because each definition table can have an arbitrary number of dimension-specific columns defined. These columns require your ETL subsystem code to be specialized in ways that none of the other five dimension tables required. Definition processing is also more complex because a single execution of the ETL subsystem might need to process multiple transactions for the same entry in the dimension. An entry added to the source master might have been updated several times before the source data intake extract obtains the source extract. Applying all of those changes to the

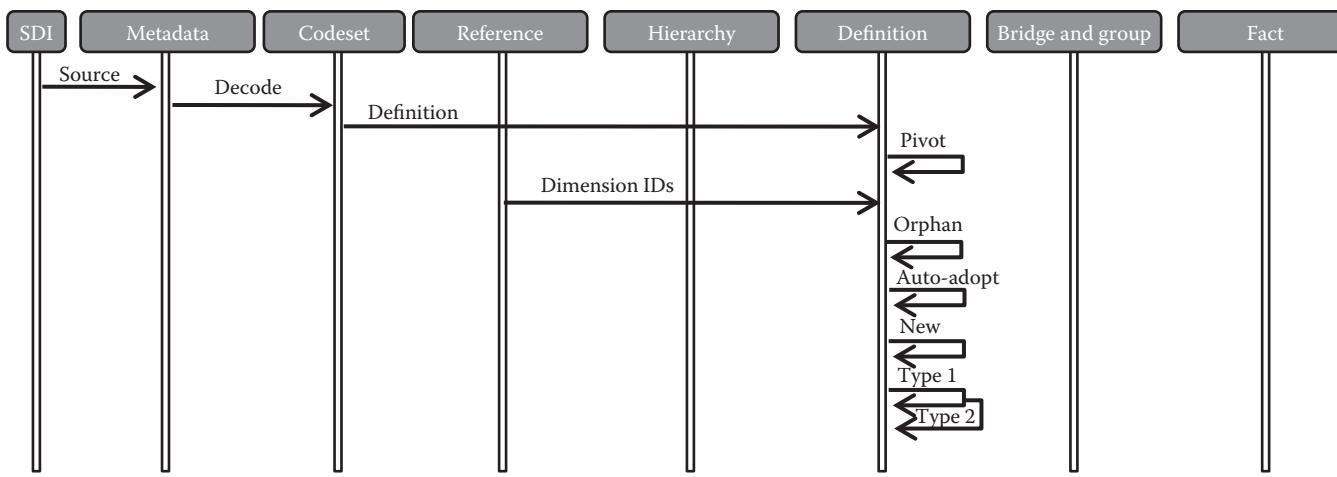


Figure 11.1 Sequence diagram for Definition pipe processing.

entries in the Definition table, in the correct order, makes the Definition pipe much more complex than if only a single transaction per execution were possible.

The complexity of processing, involving multiple transactions against the same definition entry in the same ETL run, is exacerbated by some of your warehouse performance requirements. You probably want your ETL to run as quickly as possible, so a transaction-oriented design where transactions are applied directly to the dimension tables would not be optimal, although it would solve several problems. If you could actually insert new definitions into the dimension prior to having to handle change transactions, the logic of those transactions would be much easier to design. However, you won't want to design your ETL that way precisely because it would maximize the use of database resources in ways that would become prohibitive for larger dimensions like Subject or Interaction.

The design of the Definition pipe is intended to allow for most of the processing of transactions to take place directly in memory or in the ETL staging tables. This allows you to discard logical transactions that don't need to be applied to the database because their net effect is nullified by other transactions that occur later. For example, when a new patient is added to the Subject dimension, it is fairly common to see many of the descriptive characteristics of the patient altered in the electronic health record (EHR) during the first few hours of the patient's clinical encounter. Values already entered change, or data that wasn't available earlier are added to the record. If the ETL source data for all of those transactions is processed in the same ETL job, many of those values would simply overlay each other, wasting the processing of many of the earlier elements.

Here in the ETL pipe, you'll filter out those transactions so they never actually get inserted to the warehouse database. This includes dropping transaction values where the arriving value already matches the value in the dimension, an extremely common occurrence when the source data extracts don't discriminate static versus dynamic data in their logic. Because this pipe can "ignore" data that haven't actually changed, it can be effective to have source jobs extract an entire dataset during each scheduled execution. For small datasets where it can be difficult to judge what has changed since the last extraction, this can be an effective strategy that doesn't substantially harm ETL performance.

There are basically four logical pathways through the Definition pipe processing, each being a variant of the four possible combinations of the Placeholder and Orphan flags in the staged Reference data for the Definition transactions (Figure 11.2). Each of these processing modes is relatively simple

		Orphan flag	
		False	True
Placeholder flag	False	Update	Auto-adopt
	True	New	Orphan

Figure 11.2 Orphan and Placeholder flags determine core definition processing.

when occurring in the data individually, but much more complex when arriving simultaneously during the same ETL execution:

1. *New*: New definition rows will be created in the dimensions whenever new source Definition data are received for natural keys that are not already present in the dimensions. The keys for these rows are in the Reference pipe, with the Placeholder flag set to *True* and the Orphan flag set to *False*.
2. *Orphan*: New orphan rows will be created in the dimensions whenever new natural keys have been sourced for fact or hierarchy loads that have no sourced Definition data. The keys for these rows are in the Reference pipe with the Placeholder and Orphan flags both set to *True*.
3. *Auto-adopt*: Existing orphan dimension rows will be updated with newly sourced definition values, resulting in definitions that are no longer orphaned. The keys for these rows are in the Reference pipe with the Placeholder Flag set to *False* and the Orphan Flag set to *True*. Had these values arrived prior to the Orphan being created in a previous load, these transactions would have been processed simply as new data.
4. *Updates*: Existing dimension rows will be updated with newly sourced definition values. The keys for these rows are in the Reference pipe with the Placeholder Flag set to *False* and the Orphan Flag set to *False*. These updates will be applied directly against columns that are not being controlled as SCD. For SCD data, updates will be applied that result in the expiration of a previously used data row in favor of one or more new values for some combination of columns received from one of the sources.

The processing in the Definition pipe can get fairly complex when updates arrive in the same ETL job as new or auto-adopt transactions. The complexity doesn't occur in the actual processing of any one of the transactions, but in the processing of the multiple transactions against the same definitions in the same ETL runs. Most warehouse ETL jobs run nightly or periodically throughout the day. Master data from source systems might have changed many times since the last ETL run, so a single ETL run might need to coordinate multiple transactions against the same definition rows while correctly keeping track of the exact time that each change would have been effective.

In this Beta version of the warehouse, you'll keep the processing as simple as possible in order to get a lot of data loaded quickly and as early as possible. Most of your early data in the Beta version is likely to be historical data, including any datasets that you have already been loading through the Alpha version ETL code. Historical datasets tend to be stable and will not usually present the kind of data complexity to this Definition pipe that you'll eventually have to be able to handle in your loading of day-to-day prospective data. Because of this, you'll be able to simplify the first iteration of your ETL here in the Beta version and add the more complex components later in the Gamma version.

For example, when you source your historical encounters, your source dataset is likely to be a massive table where most encounters have already gone through their full life cycle of discharge and billing. You'll load these data as new data in the Interaction dimension and probably not include any updates until much later, and certainly not during the same ETL execution. Likewise, if you have sources that you want to load into Beta that include both insert and update transactions, you can schedule the two types of transactions to run as separate ETL jobs, allowing the data to be loaded without the need to build the fully integrated logic into the Definition pipe that you'll eventually implement in the Gamma version. You want to load lots of data quickly, so here in Beta, you should be very aware of what you are loading, and plan your executions accordingly. In your final production warehouse, you won't need to keep track of the nuances of transaction types arriving from each source system, because the generalized logic you'll implement in the Gamma version will be handling these transactional interactions.

Metadata Transformation

As in the Reference pipe, processing in the Definition pipe begins with a join from the source data to the Metadata dimension in order to transform the source-only data in the intake tables into the source-to-target mappings needed for ETL processing:

```
SELECT S.SDI_JOB_ID,
       S.PHYSICAL_DATASET,
       CASE WHEN S.LOGICAL_DATASET = R.CONTEXT_KEY_02
            THEN S.LOGICAL_DATASET
            WHEN S.LOGICAL_DATASET = '~All~'
            THEN R.CONTEXT_KEY_02
            ELSE S.LOGICAL_DATASET
            END LOGICAL_DATASET,
       CASE WHEN S.FACT_BREAK = R.CONTEXT_KEY_03
            THEN S.FACT_BREAK
            WHEN S.FACT_BREAK = '~All~'
            THEN R.CONTEXT_KEY_03
            ELSE S.FACT_BREAK
            END FACT_BREAK,
       S.ENTERPRISE_ID,
       S.BREAK_ID,
       S.SOURCE_TIMESTAMP,
       S.DATAFEED_BREAK,
       S.ETL_TRANSACTION,
       S.TARGET_STATE,
       D.DATASET_TYPE,
       D.TARGET_DIMENSION,
```

```

D.DIMENSION_CONTEXT,
D.TARGET_SUBDIMENSION,
D.CHANGE_TYPE,
R.CONTEXT_KEY_05 AS TARGET_TABLE,
R.CONTEXT_KEY_06 AS TARGET_COLUMN,
COALESCE(CD.DESCRIPTION, S.VALUE) AS VALUE
FROM STAGING.SOURCE S
INNER JOIN METADATA_R R
    ON S.PHYSICAL_DATASET = R.CONTEXT_KEY_01
    AND (S.LOGICAL_DATASET = R.CONTEXT_KEY_02
        OR S.LOGICAL_DATASET = '~All~'
        OR R.CONTEXT_KEY_02 = '~All~' )
    AND (S.FACT_BREAK = R.CONTEXT_KEY_03
        OR S.FACT_BREAK = '~All~'
        OR R.CONTEXT_KEY_03 = '~All~' )
    AND S.SOURCE_COLUMN = R.CONTEXT_KEY_04
INNER JOIN METADATA_D D
    ON R.MASTER_ID = D.MASTER_ID
    AND D.STATUS_INDICATOR = 'A'
LEFT JOIN CODESET_R CR
    INNER JOIN CODESET_D CD
        ON CR.MASTER_ID = CD.MASTER_ID
        AND S.SOURCE_TIMESTAMP
            BETWEEN CD.EFFECTIVE_TIMESTAMP AND CD.EXPIRATION_TIMESTAMP
        ON CR.CONTEXT_KEY_01 = D.CODESET_CONTEXT_KEY_01
        AND CR.CONTEXT_KEY_02 = D.CODESET_CONTEXT_KEY_02
        AND CR.CONTEXT_KEY_03 = S.VALUE
WHERE D.TARGET_TYPE = 'Definition';

```

The metadata transformation includes the same standard codeset translation that was used in the Reference pipe. It also pulls the Target table from the metadata, although here in the Beta version, it is presumed that the Target table is always the Definition table in the dimension. The targeting of alternative Outrigger tables will be deferred until the Gamma version.

Deep and Wide Staging

The data flowing through this Definition pipe will be needed in two different orientations: a *deep* orientation as it comes out of the metadata transformation where each source value has its own row and a *wide* orientation where the source values have been pivoted into a single row for processing along with the needed reference data from the Reference pipe. Your database operations to update the Definition tables will depend upon the wide orientation, while your column-level editing, validation, and change control in the staging tables will reply upon the deep orientation.

The first orientation you'll need is the deep variant so you can apply column-specific processing rules to the single-valued deep staging table. The staging table for deep processing matches the output of the actual metadata transformation join:

```
CREATE TABLE #SOURCE_DEFINITION_DEEP (
    SDI_JOB_ID           INT,
    PHYSICAL_DATASET     VARCHAR(64),
    LOGICAL_DATASET      VARCHAR(64),
    FACT_BREAK           VARCHAR(64),
    ENTERPRISE_ID        VARCHAR(64),
    BREAK_ID             INT,
    SOURCE_TIMESTAMP     DATETIME,
    DATAFEED_BREAK       VARCHAR(64),
    ETL_TRANSACTION      VARCHAR(64),
    TARGET_STATE          VARCHAR(64),
    DATASET_TYPE         VARCHAR(10),
    TARGET_DIMENSION      VARCHAR(64),
    TARGET_CONTEXT        VARCHAR(64),
    TARGET_SUBDIMENSION   VARCHAR(64),
    CHANGE_TYPE           VARCHAR(10),
    TARGET_TABLE          VARCHAR(64),
    TARGET_COLUMN         VARCHAR(30),
    VALUE                VARCHAR(64)
);
```

You can load the deep staging table directly from the metadata transformation query:

```
INSERT INTO #SOURCE_DEFINITION_DEEP
<<< transformation query HERE >>>
    AND (D.REQUIRED_FLAG = 'Y' OR VALUE <> '~Missing~');
```

Note that the extension of the WHERE clause prevents optional columns with no sourced values from being placed into the deep staging table. These values aren't needed for processing and, if retained, will use extra processing unnecessarily.

The staging table for the wide orientation of Definition data requires more columns because the wider pivoted data will only have one row per transaction:

```
CREATE TABLE #SOURCE_DEFINITION_WIDE (
    JOB_ID               INT,
    PHYSICAL_DATASET     VARCHAR(64),
    LOGICAL_DATASET      VARCHAR(64),
    FACT_BREAK           VARCHAR(64),
    BREAK_ID             INT,
```

```

DATAFEED_BREAK          VARCHAR(64),
ETL_TRANSACTION         VARCHAR(64),
TARGET_STATE            VARCHAR(64),
SOURCE_TIMESTAMP        DATETIME,
TARGET_DIMENSION        VARCHAR(64),
TARGET_SUBDIMENSION     VARCHAR(64),
TARGET_TYPE              VARCHAR(64),
DIMENSION_CONTEXT       VARCHAR(64),
REFERENCE_COMPOSITE    VARCHAR(772),
CONTEXT_ID               INT,
DEFINITION_ID           INT,
MASTER_ID                INT,
PLACEHOLDER_FLAG        BIT,
UNKNOWN_FLAG             BIT,
ORPHAN_FLAG              VARCHAR(1),
CHANGE_TYPE              VARCHAR(10),
ROW_ASCENDING            INT,
ROW_DESCENDING           INT,
-- STUB COLUMNS
DESCRIPTION              VARCHAR(30),
FULL_DESCRIPTION          VARCHAR(64),
EXTENDED_DESCRIPTION      VARCHAR(250),
CATEGORY                 VARCHAR(64),
ABBREVIATION              VARCHAR(64),
VERNACULAR_CODE           VARCHAR(64),
-- DIMENSION-SPECIFIC COLUMNS
LAST_NAME                 VARCHAR(64),
FIRST_NAME                 VARCHAR(64),
GENDER                     VARCHAR(64),
DATE_OF_BIRTH              VARCHAR(64),
CLINICAL_SPECIALTY        VARCHAR(64),
GENDER_SPECIFICITY         VARCHAR(64),
THERAPEUTIC_CLASS          VARCHAR(64),
DEGREES_LONGITUDE          VARCHAR(64)
);

```

The number of dimension-specific column names causes some variability in the definition of this staging table because any distinct column name added to any of the Definition tables needs to be represented explicitly in this processing. My experience is that the total number of specific columns doesn't grow as quickly as you might think and therefore does not create a performance strain on processing. If it seems to create problems, there will be several alternatives for adjusting or tuning this approach available during the Gamma version development. For the Beta version, I recommend sticking with this single generic staging table in order to keep this version moving forward quickly. Loading this deep staging table requires a pivot of the deep columns in the source data, plus a join to the Reference pipe in order to pick up the needed Reference information for the definition tables:

```

INSERT INTO #SOURCE_DEFINITION_WIDE
SELECT P.JOB_ID,
       P.PHYSICAL_DATASET,
       P.LOGICAL_DATASET,
       P.FACT_BREAK,
       P.ENTERPRISE_ID,
       P.BREAK_ID,
       P.SOURCE_TIMESTAMP,
       P.DATAFEED_BREAK,
       P.ETL_TRANSACTION,
       P.TARGET_STATE,
       P.DATASET_TYPE,
       P.TARGET_DIMENSION,
       P.DIMENSION_CONTEXT,
       P.TARGET_SUBDIMENSION,
       P.TARGET_TABLE,
       REF.TARGET_SUBDIMENSION,
       REF.TARGET_TYPE,
       REF.REFERENCE_COMPOSITE,
       REF.CONTEXT_ID,
       REF.DEFINITION_ID,
       REF.MASTER_ID,
       REF.PLACEHOLDER_FLAG,
       REF.UNKNOWN_FLAG,
       REF.ORPHAN_FLAG,
       COALESCE(DESCRIPTION,          '~Null~') AS DESCRIPTION,
       COALESCE(FULL_DESCRIPTION,    '~Null~') AS FULL_DESCRIPTION,
       COALESCE(EXTENDED_DESCRIPTION, '~Null~') AS EXTENDED_DESCRIPTION,
       COALESCE(CATEGORY,           '~Null~') AS CATEGORY,
       COALESCE(ABBREVIATION,        '~Null~') AS ABBREVIATION,
       COALESCE(VERNACULAR_CODE,     '~Null~') AS VERNACULAR_CODE,
-- DIMENSION-SPECIFIC COLUMNS
       LAST_NAME,
       FIRST_NAME,
       GENDER,
       DATE_OF_BIRTH,
       CLINICAL_SPECIALTY,
       GENDER_SPECIFICITY,
       THERAPEUTIC_CLASS,
       DEGREES_LONGITUDE

FROM  (
#SOURCE_DEFINITION_DEEP S
INNER JOIN #SOURCE1_REFERENCE REF
      ON REF.JOB_ID          = S.SDI_JOB_ID
      AND REF.PHYSICAL_DATASET = S.PHYSICAL_DATASET
      AND REF.LOGICAL_DATASET  = S.LOGICAL_DATASET
      AND REF.DATASET_TYPE     = S.DATASET_TYPE
      AND REF.BREAK_ID         = S.BREAK_ID
      AND REF.TARGET_DIMENSION = S.TARGET_DIMENSION
)
PIVOT
(
  MIN(VALUE)
  FOR TARGET_COLUMN IN      (
                                DESCRIPTION,
                                FULL_DESCRIPTION,
                                EXTENDED_DESCRIPTION,
                                CATEGORY,

```

```

ABBREVIATION,
VERNACULAR_CODE,
-- DIMENSION-SPECIFIC COLUMNS
LAST_NAME,
FIRST_NAME,
GENDER,
DATE_OF_BIRTH,
CLINICAL_SPECIALTY,
GENDER_SPECIFICITY,
THERAPEUTIC_CLASS,
DEGREES_LONGITUDE
)
) P;

```

The inclusion of Reference pipe data in this wide orientation is important. Proponents of late binding might suggest that we leave the reference data separate and join to it as needed. That would actually work for some of the simple examples we'll be seeing first, but it would fail later as more complicated cases are included in the pipe capability. The reason for this is that certain control flags have been set in the Reference pipe according to the requirements of the individual rows in that pipe, as they were being processed in the Reference pipe, but here in the Definition pipe, we have a more longitudinal view of the data. There could be multiple rows in this pipe for the same Reference data, if the source system from which the data was taken is processing multiple transactions per day. As a result, if multiple rows have the Placeholder flag turned on in the Reference pipe, only one of those transactions will be treated as new data here in this Definition pipe. Subsequent references with Placeholder turned on will need to be treated as updates. The same holds true for orphans: An orphan can only be adopted once. Subsequent updates to that Reference entry in the same ETL run will need to be processed as updates. This is why the reference data are brought over into the wide view staging table. It allows the control flags to be altered for Definition processing without impacting Reference pipe processing that might be continuing in parallel with Definition pipe processing.

Example Master Loads

In the vast majority of warehouse loads, a dimension entry will have no more than one transaction flowing through the Definition pipe during any single execution of the ETL. This will be particularly true for historical data loads. For example, a simple load of a patient master source will, presumably, have a single transaction for each patient in that source table, and each transaction will be one of three of the types outlined earlier: new, auto-adopt, or update. A simple master-only load will never result in an Orphan row being created,

and the auto-adopt option will only occur if facts have already been loaded against these master data resulting in orphans having been created. During history loading, you'll typically load your master data first, so auto-adopt transactions will be relatively rare. Because orphan and auto-adopt transactions are rare during history loading of master data, new and update transactions are dominant while loading the Beta version. These transactions are the easiest to handle because they are processed directly since updates are usually not received for the new data in the same ETL run.

Let's look at an example of some simple master data to be loaded into the Diagnosis dimension from a source that provides SNOMED codes and descriptions for diagnoses. Table 11.1 illustrates three example codes with their descriptions.

In this simplified case, the metadata for the example data in Table 11.1 will likely map the SNOMED code into the Reference pipe as the natural key of the data, but will probably also map it into this Definition pipe to populate the Vernacular Code stub column (Table 11.2). The SNOMED Description column will likely be mapped to the Extended Description column because it might exceed the length of the stub Description or Full Description columns, and you will want to avoid truncating the longer descriptions. Recall that the value will also end up in the Description and Full Description columns upon loading if no other values are mapped to those columns. This defaulting logic is why there is no need to map individual source columns to more than one of the three stub description columns. Just map the input value to the shortest description column that will not truncate any of the source values.

The metadata transformation join combines the source data with the metadata in order to obtain the standard ETL intake dataset that provides for mappings

Table 11.1 Simple Diagnosis Definition Data

<i>SNOMED_CD</i>	<i>SNOMED_DESC</i>
423894005	Refractory migraine
26150009	Lower half migraine
128187005	Vascular headache

Table 11.2 Example of Simple SNOMED Metadata Mappings

<i>SOURCE_COLUMN</i>	<i>TARGET_TABLE</i>	<i>TARGET_COLUMN</i>
SNOMED_CD	DIAGNOSIS_R	CONTEXT_KEY_01
SNOMED_CD	DIAGNOSIS_D	VERNACULAR_CODE
SNOMED_DESC	DIAGNOSIS_D	EXTENDED_DESCRIPTION

Table 11.3 Example of Unpivoted SNOMED Metadata Source Data

SOURCE_COLUMN	SOURCE_VALUE	TARGET_COLUMN
SNOMED_CD	423894005	VERNACULAR_CODE
SNOMED_DESC	Refractory migraine	FULL_DESCRIPTION
SNOMED_CD	423894005	VERNACULAR_CODE
SNOMED_DESC	Lower half migraine	FULL_DESCRIPTION
SNOMED_CD	128187005	VERNACULAR_CODE
SNOMED_DESC	Vascular headache	FULL_DESCRIPTION

from source column values to target column values (Table 11.3). After the staging of the Definition data into the deep and wide orientations, the data are positioned for processing against the various dimensions. Continuing with this overly simplified case, let's assume that all three entries need to be added to the Diagnosis dimension as new data. The insertion will be accomplished directly from the wide staging data:

```
INSERT INTO DIAGNOSIS_D
SELECT MASTER_ID AS DIMENSION_ID,
MASTER_ID,
TARGET_SUBDIMENSION AS SUBDIMENSION,
CATEGORY,
DESCRIPTION,
FULL_DESCRIPTION,
EXTENDED_DESCRIPTION,
ABBREVIATION,
VERNACULAR_CODE,
'N' AS ORPHAN_FLAG,
SOURCE_TIMESTAMP
FROM #SOURCE_DEFINITION_WIDE
WHERE TARGET_DIMENSION = 'Diagnosis'
    AND PLACEHOLDER_FLAG = 'Y'
    AND ORPHAN_FLAG      = 'N';
```

Note that this example code leaves a lot of columns in the Definition row NULL in order to illustrate what might be used to set up very quick initial loads. (We'll discuss the missing details after going through this example.) By inserting basic definition rows, you have accomplished roughly the equivalent of the Alpha version processing of the dimensions. Each needed key is now established as a legitimate row in its appropriate dimension. The Alpha version occasionally dropped some facts because keys existed in certain facts that had not yet been loaded into the Alpha version dimensions. Here in the Beta version, you close this gap by creating an orphan entry in the dimension for

any key received in a fact that hasn't yet been received in a dimension load. You can easily load orphans directly from the Reference pipe data:

```
INSERT INTO DIAGNOSIS_D
SELECT DISTINCT
    MASTER_ID AS DIMENSION_ID,
    MASTER_ID,
    TARGET_SUBDIMENSION AS SUBDIMENSION,
    NULL AS CATEGORY,
    REFERENCE_COMPOSITE AS DESCRIPTION,
    REFERENCE_COMPOSITE AS FULL_DESCRIPTION,
    CONCAT(TARGET_SUBDIMENSION, ': ', REFERENCE_COMPOSITE) AS EXTENDED_DESCRIPTION,
    NULL AS ABBREVIATION,
    REFERENCE_COMPOSITE AS VERNACULAR_CODE,
    'Y' AS ORPHAN_FLAG,
    SOURCE_TIMESTAMP
FROM #SOURCE1_REFERENCE
WHERE TARGET_DIMENSION = 'Diagnosis'
    AND PLACEHOLDER_FLAG = 'Y'
    AND ORPHAN_FLAG = 'Y';
```

With the addition of this second basic insert statement, the Beta version is already more powerful than the entire Alpha version because facts will not be dropped for lack of Definition data in a dimension. To make this simplified example complete, you need to be able to process an auto-adoption of a new definition entry that might have been loaded as an orphan in a previous ETL execution:

```
UPDATE DIAGNOSIS_D
SET ORPHAN_FLAG = 'N',
    SOURCE_TIMESTAMP = S.SOURCE_TIMESTAMP,
    CATEGORY = S.CATEGORY,
    DESCRIPTION = S.DESCRIPTION,
    FULL_DESCRIPTION = S.FULL_DESCRIPTION,
    EXTENDED_DESCRIPTION = S.EXTENDED_DESCRIPTION,
    ABBREVIATION = S.ABBREVIATION,
    VERNACULAR_CODE = S.VERNACULAR_CODE
FROM
    #SOURCE_DEFINITION_WIDE S
    INNER JOIN DIAGNOSIS_D D
        ON S.MASTER_ID = D.MASTER_ID
        AND S.DIMENSION_ID = D.DIMENSION_ID
WHERE
    S.TARGET_DIMENSION = 'Diagnosis'
    AND S.PLACEHOLDER_FLAG = 'N'
    AND S.ORPHAN_FLAG = 'Y';
```

With this addition of a rudimentary orphan adoption update, three of the four transaction types within the Definition pipe are now working for very simple source loads: new data inserts, orphan inserts, and auto-adoption of orphans. Most of the dimension data in the Beta version can now be loaded. Still to be added are

steps for processing changes to the data. That capability is more complex because of differences in the ways that certain updates will need to be processed. The steps needed to process multiple transactions for the same keys in the same runs are also still missing. These additional capabilities will be accomplished here in the Beta version, but the advantage to where you are now is that a portion of the development resource can now be dedicated to actually starting to load master dimension data into the dimensions, making the warehouse useful for early queries at the same time that Beta development continues.

Let's back up now and go through the logic of the Definition pipe dimension inserts and updates somewhat more systematically. As we proceed, we must keep in mind the narrow objectives of the Beta version in order to properly prioritize what the logic needs to be able to do. The Beta version is about getting a basic working warehouse in place into which we can load most of the data you want in your warehouse. Many of the quality control characteristics that you need in the finished warehouse, as well as some of the more complex transaction types, are being deferred into the Gamma version. The reality, in my experience to date, is that the Beta version actually handles over 95% of the loads you'll ever want or need to do. The exceptions are rare enough, and many work-arounds are available for accomplishing those exceptions using Beta version logic that many organizations take an extended break in the development schedule as they complete this Beta version. The basic logic of loading and changing data in the dimensions is accomplished by the code illustrated in the remainder of this chapter.

Insert New Definitions

Inserting new entries into the Definition tables in the dimensions is fairly straightforward, with the main complexity simply coming from the fact that each dimension needs to be loaded separately. The logic for loading each dimension is the same, but since each dimension has its own set of physical tables in the database, a separate `INSERT` statement is needed for each. The Definition tables include control values that must be populated when rows are inserted, so the various dimension control features in this generic design will work correctly, even though some of those controls won't be implemented until later in the Gamma version. The seven columns that must be set include the following:

1. *Status Indicator*: Always set this column to Active. The logic for SCDs might later change the value to Superseded, but a new insert will always be Active. The Deleted status won't be used here until the Gamma version.
2. *Orphan Flag*: The value of this is determined by the Reference pipe lookup of the natural keys in the dimension. Always initialize this column based on that value.
3. *Redaction Control Flag*: Always initialize this value to No.

4. *Unexpected Flag*: Always initialize this value to *No*.
5. *Undesired Flag*: Always initialize this value to *No*.
6. *Effective Timestamp*: Always initialize this value to the lowest timestamp you've chosen for the warehouse. The below example uses "1800-01-01" to indicate an earliest timestamp.
7. *Expiration Timestamp*: Always initialize this value to the highest timestamp you've chosen for the warehouse. The following example uses "2099-12-31" to indicate a latest timestamp.

The combination of these values, along with the data in the wide Definition pipe, is sufficient to insert all of the newly arriving Definition entries:

```
INSERT INTO DIAGNOSIS_D
SELECT DIMENSION_ID,
       MASTER_ID,
       TARGET_SUBDIMENSION,
       CATEGORY,
       DESCRIPTION,
       FULL_DESCRIPTION,
       EXTENDED_DESCRIPTION,
       ABBREVIATION,
       VERNACULAR_CODE,
       'A' AS STATUS_INDICATOR,
       'N' AS ORPHAN_FLAG,
       'N' AS REDACTION_CONTROL_FLAG,
       'N' AS UNEXPECTED_FLAG,
       'N' AS UNDESIRED_FLAG,
       '1800-01-01' AS EFFECTIVE_TIMESTAMP,
       '2099-12-31' AS EXPIRATION_TIMESTAMP,
       SOURCE_TIMESTAMP,
       GENDER_SPECIFICITY
FROM #SOURCE1_DEFINITION_WIDE
WHERE TARGET_DIMENSION = 'Diagnosis'
      AND PLACEHOLDER_FLAG = 'Y'
      AND ORPHAN_FLAG = 'N';
```

The capability to externally source the Redaction Control, Unexpected, and Undesired flags will be added to the Definition pipe in the Gamma version. At that time, the code will use these default values only if a sourced value doesn't arrive. Having a source determine when to turn these flags on is a data governance issue for the warehouse and requires that a governance function be in place to discuss and approve the setting of these flags. The role and activities of the governance function are discussed in Chapter 20. Those activities will result in the data warehouse support team putting together specialized data sources to set or reset these flags in the various dimensions. Until then, you might want to experiment with manually setting these flags when circumstances arise where

they might be helpful. Just be advised that some of the logic to actually use these values won't be present until the Gamma version.

One example where you might choose to turn on the Redaction Control flag is in the Diagnosis dimension for any row that references HIV in one of its descriptions:

```
UPDATE DIAGNOSIS_D
    SET REDACTION_CONTROL_FLAG      = 'Y'
  FROM DIAGNOSIS_D
 WHERE REDACTION_CONTROL_FLAG = 'N'
   AND (DESCRIPTION LIKE '%HIV%'
     OR FULL_DESCRIPTION LIKE '%HIV%'
     OR EXTENDED_DESCRIPTION LIKE '%HIV%');
```

I've also encountered environments where some user group would alter the name of medications in the drug master that were no longer considered valid in the source system (but were still needed for historical purposes). In some cases, the name of the medication would be changed to insert "ZZZ" at the front of the drug name so the obsolete drug would fall to the bottom of a pick list in the source system. In a case like that, you could choose to set both the Unexpected and Undesired flags for those medications:

```
UPDATE MATERIAL_D
    SET UNEXPECTED_FLAG      = 'Y',
        UNDESIRED_FLAG       = 'Y'
  FROM MATERIAL_D
 WHERE SUBDIMENSION = 'Medication'
   AND EXTENDED_DESCRIPTION LIKE 'ZZZ%';
```

The opportunities to perform analysis and alter data flags will seem endless once you start looking. Be careful not to get pulled too far in that direction during Beta version development. While I encourage a certain amount of exploration and experimentation in this area in order to better learn the purpose of the controls in the dimension, the reality is that many of these flags need to be set more carefully than the code illustrated here implies. The flags will typically be controlled according to the logic of SCDs so you'll not only mark an entry as unexpected or undesired, but also according to the date and time that the change occurred. The Gamma version will flag ETL data that loads against entries with these flags on, and you won't want that to occur while loading historical data that might precede the conditions that caused those flags to be turned on.

Splitting data in the Definition pipe so rows can be processed into each dimension is something I refer to as the "fan out" of the pipe to the dimension. The origin of the term is an analogy to traditional accordions-folded fans that collapse to a narrow, closed form and then expand out for use.

The single-staged data pipe is fanned out so data for each dimension are processed separately. The only basic difference across the separate `INSERT` statements is in the presence or absence of dimensionally specific columns that cause each dimension to be unique. The Gender Specificity column in the earlier diagnosis example code illustrates this kind of difference.

The logic of each dimension code after the fan-out is similar enough that you can fairly easily build out all of the needed code for all of the dimensions. I recommend against that approach, however, at least as a single step in the development. There are bound to be dimensions in your design that won't have any data in the sources you choose to load early in the Beta version. Providing logic for those dimensions wastes your time until some data are identified for loading. Beyond that, any development effort is subject to mistakes and errors that creep into the code over time. Why code the logic for 26 dimensions until you're sure the logic is correct. I suggest coding the logic for the first few dimensions that are needed for your initial source datasets, perhaps only the data sources you loaded in your Alpha version. It's okay if there are more data than that in the pipe at any one time. Data for dimensions you haven't coded yet will simply fall away. Logic for subsequent dimensions can be added as new data becomes available, or you reach a point where you've tested your code through enough iterations so you have high confidence it is defect free.

New Orphans

The fanned-out logic you need to insert new Orphan entries into each dimension is very similar to the logic of newly inserted entries except that there are no data in the Definition pipe to use to populate the columns. Instead, you'll populate the columns with values that might be useful to analysts until such time as the orphans get adopted by legitimate data sources.

```
INSERT INTO DIAGNOSIS_D
SELECT DISTINCT
    MASTER_ID AS DIMENSION_ID,
    MASTER_ID,
    TARGET_SUBDIMENSION AS SUBDIMENSION,
    REFERENCE_COMPOSITE AS DESCRIPTION,
    REFERENCE_COMPOSITE AS FULL_DESCRIPTION,
    CONCAT(TARGET_SUBDIMENSION, ': ', REFERENCE_COMPOSITE) AS EXTENDED_DESCRIPTION,
    REFERENCE_COMPOSITE AS VERNACULAR_CODE,
    'A' AS STATUS_INDICATOR,
    'Y' AS ORPHAN_FLAG,
    'Y' AS REDACTION_CONTROL_FLAG,
    'N' AS UNEXPECTED_FLAG,
    'N' AS UNDESIRED_FLAG,
    '1800-01-01' AS EFFECTIVE_TIMESTAMP,
    '2099-12-31' AS EXPIRATION_TIMESTAMP,
    SOURCE_TIMESTAMP
FROM #SOURCE1_REFERENCE
WHERE TARGET_DIMENSION = 'Diagnosis'
    AND PLACEHOLDER_FLAG = 'Y'
    AND ORPHAN_FLAG      = 'Y';
```

Note the way this code defaults several of the stub description columns to the value in the Reference Composite, which was the natural key that identified the data in the source system. In addition, the extended description includes the subdimension for that key. These descriptions provide basic support for these orphan rows since they might be used as selected columns in queries. Blank (NULL) description columns in query results can cause confusion among users not expecting them, and some business intelligence interfaces might incorrectly aggregate data across multiple orphans if the descriptions are part of the grouping of data in the queries. To be safe, having the Reference Composite in these columns guarantees uniqueness for aggregation and will usually be sufficiently informative for users to understand what the data mean.

Also note that you should turn on the Redaction Control flag in orphan rows. Since, by definition, you don't know what the orphan is, you can't assert that it would not be patient identifying if displayed. Therefore, you should turn on the Redaction Control flag as a precaution to prevent inadvertent disclosure of data that would have been protected if loaded from a direct source. The auto-adopt logic that follows will turn the flag off when complete data arrive in a subsequent source.

Orphan Auto-Adoption

The fanned-out logic for automatic adoption of existing orphans updates the columns of the Definition row in the same way they would have been inserted had the Definition data arrived before the orphan row had been created. This is really a new entry that happens to have been inserted as an orphan because one or more facts were loaded into the warehouse that needed this definition to be present. After the update, the definition will look the same as if it had been loaded prior to those initial facts:

```
UPDATE DIAGNOSIS_D
  SET ORPHAN_FLAG          = 'N',
      REDACTION_CONTROL_FLAG= 'N',
      SOURCE_TIMESTAMP       = S.SOURCE_TIMESTAMP,
      PAY_SOURCE             = S.PAY_SOURCE,
      FINANCIAL_CLASS        = S.FINANCIAL_CLASS,
      DESCRIPTION             = S.DESCRIPTION,
      FULL_DESCRIPTION        = S.FULL_DESCRIPTION,
      EXTENDED_DESCRIPTION    = S.EXTENDED_DESCRIPTION,
      ABBREVIATION            = S.ABBREVIATION,
      VERNACULAR_CODE         = S.VERNACULAR_CODE
      GENDER_SPECIFICITY      = S.GENDER_SPECIFICITY
FROM
  #SOURCE1_DEFINITION S
```

```

    INNER JOIN DIAGNOSIS_D D
        ON S.MASTER_ID      = D.MASTER_ID
        AND S.DIMENSION_ID  = D.DIMENSION_ID
WHERE
    S.TARGET_DIMENSION     = 'Diagnosis'
    AND S.PLACEHOLDER_FLAG = 'N'
    AND S.ORPHAN_FLAG      = 'Y';

```

This logic turns off the Orphan Flag now that the row has been adopted by the arrival of legitimate dimension load data. The Redaction Control Flag also gets turned off now because the data needed to otherwise determine the protected status of this entry is now available for a more realistic appraisal of the entry contents.

Because these updated rows now look just like newly inserted entries in the dimension, you'll also want to apply any updates to those rows that you might have used after inserting new rows to update the various flags. In practice, you'll probably defer those updates all together until you've completed the auto-adoption logic.

Definition Change Processing

Making changes to data in the Definition tables of the dimensions is the most complicated aspect of this Definition pipe. Complications arise because of the need to be able to manage certain levels of slow change within the dimensions themselves and to be able to coordinate multiple changes to the same dimension entries that arrive and are processed in the same ETL job. Each transaction that impacts a definition in a dimension isn't usually particularly complex to handle by itself. The complications arise in the generality of this ETL subsystem design and the fact that all of the processing happens at the same time prior to any direct updates to the warehouse tables taking place.

Slowly-Changing Dimensions

The first major complication comes from the notion of SCDs, the idea that some of the properties of the dimensions change slowly enough that you can keep track of the historical versions of those properties directly in the dimension rather than having to insert facts into a fact table. In doing so, you can join facts to the dimensions in three standard ways: (1) against the currently active definition values, (2) against the values that were current on the day that the fact was current, or (3) against the values at any other arbitrary point in time. Of these three options, you'll find that the vast majority of all queries use the first option: joining facts to the current value of the dimension definitions.

This realization that most queries don't take advantage of any processing we do to support SCDs is very important. Why invest so much effort in implementing a complex feature that virtually no one ever uses? In my experience, when the feature is helpful at all, it is *extremely* helpful. You know the descriptions of your various ICD-9, ICD-10, CPT-4, or other codes change periodically, some on a regular schedule. You often don't think very much about *how* they are changing. Imagine querying some clinical data to see if the current descriptions of the diagnosis codes are materially different than the descriptions of those same codes at the times they were used in patient records. If those descriptions have been maintained as slowly-changing properties of the dimension, a simple query can return results for the situations where the two descriptions are different.

A more extreme example might involve the gender of the patient. Gender can be very important to understanding the assignment of diagnosis and procedure codes to a clinical record where either of those codes can have a gender specificity property that indicates that it is only to be used for one gender. The clinical complexity of gender reassignment, and how an institution assigns values to all of these coded properties, is beyond our scope here, but if you'd simply like to query your warehouse to see how often those conditions are being violated in the data, you need to be able to query the gender of the patient *as it was on the dates of the assignments* of the clinical codes, not just the gender recorded for the patient today.

A list of legitimate and powerful examples of the use of slowly-changing properties would be very long. I'm not asserting that these examples are rare, only that the actual queries that take advantage of this capability are extremely rare. Because of this, the design of the star pattern in this book differs from traditional approaches in the literature in an important way: The Bridge table of the dimension points to the Definition table using the Master ID, not the Definition ID. If you scan the literature on star schemas, you'll typically see that in SCDs the facts are aligned with the Definition rows that were current at the time of the fact. An extra join is needed to get to the current row:

```
SELECT C.DESCRIPTION AS CURRENT_DESCRIPTION
  FROM FACT F
 INNER JOIN PROCEDURE_B B ON F.PROCEDURE_GROUP_ID = B.GROUP_ID
 INNER JOIN PROCEDURE_D D ON B.DEFINITION_ID = D.DEFINITION_ID
 INNER JOIN PROCEDURE_D C ON D.MASTER_ID = C.MASTER_ID AND C.STATUS_INDICATOR = 'A' ;
```

The traditional set of joins to get from a Fact entry to a *current* description in a dimension requires joining the Fact to the Bridge table on the Group ID, using the Definition ID in the Bridge to get to the Definition table entry that was current at the time of the Fact, and finally using the Master ID in that Definition row to join again to the Definition row to get the current entry for that definition (where the Status Indicator is Active or Deleted). In many cases, the same row is returned for both joins to the Definition table because the row at the time of the Fact is still the current row.

Instead of using the Definition ID in the Bridge tables, I use the Master ID. This means that instead of pointing to the one row that was current at the time of the fact, the Master ID in the Bridge actually points to the *set* of Definition rows for the entry of interest. That set is reduced to a single returned row by filtering specifically on the Active status, or else on a specific timestamp of interest. This approach never requires a second join to the Definition table whether a historical or current value is desired. Obtaining a current description of the dimension entry requires only one join:

```
SELECT C.DESCRIPTION AS CURRENT_DESCRIPTION
  FROM FACT F
  INNER JOIN PROCEDURE_B B ON F.PROCEDURE_GROUP_ID = B.GROUP_ID
  INNER JOIN PROCEDURE_D C ON B.MASTER_ID = C.MASTER_ID AND C.STATUS_INDICATOR = 'A';
```

The Master ID in the Bridge joins to the *set* of Definition rows, and the Active status filter reduces the set to just the current row. If, instead of the current row, the historical row that was current at the time of the fact is desired, it is returned by filtering the query on the Source Timestamp of the fact being between the Effective and Expiration Timestamps of the Definition row:

```
SELECT A.DESCRIPTION AS ASOF_DESCRIPTION
  FROM FACT F
  INNER JOIN PROCEDURE_B B ON F.PROCEDURE_GROUP_ID = B.GROUP_ID
  INNER JOIN PROCEDURE_D A ON B.MASTER_ID = A.MASTER_ID
    AND F.SOURCE_TIMESTAMP IS BETWEEN C.EFFECTIVE_TIMESTAMP AND C.EXPIRATION_TIMESTAMP;
```

In those relatively rare cases where both historical and current versions of the Definition entry are required, the two-join strategy returns:

```
SELECT A.DESCRIPTION AS ASOF_DESCRIPTION,
      C.DESCRIPTION AS CURRENT_DESCRIPTION
  FROM FACT F
  INNER JOIN PROCEDURE_B B ON F.PROCEDURE_GROUP_ID = B.GROUP_ID
  INNER JOIN PROCEDURE_D A ON B.MASTER_ID = A.MASTER_ID
    AND F.SOURCE_TIMESTAMP IS BETWEEN C.EFFECTIVE_TIMESTAMP AND C.EXPIRATION_TIMESTAMP
  INNER JOIN PROCEDURE_D C ON B.MASTER_ID = C.MASTER_ID AND C.STATUS_INDICATOR = 'A';
```

The two approaches described here provide the same basic access to both current and historical views of the Definition entries. I prefer an approach where the simplest query is the one used most often and the most complex query is the one used least often. Unless a specific performance concern impacts how a query will be designed, I always prefer coding the simplest possible query.

One major functional advantage to the set-based approach to these joins is that the choice of which Definition row was current at the time of the Fact is not *bound* until the query executes. The traditional approach of using the Definition ID in the Bridge represents *early binding*, being bound as the ETL loads the Facts into the warehouse. This means that if a slowly managed change takes place in the dimension *after* the Facts are loaded, there's a chance that

some of the facts will no longer be left pointing to the correct Definition entries. Suppose you have a Definition row that is valid throughout 2014, with a number of sourced Facts pointing to dates throughout the entire year. All of those facts are pointing to the Definition row that was current at the time of the facts. If a late-arriving change dated July 1, 2014, arrives against that definition, such as something as simple as a change to the description, the original Definition entry that was valid throughout 2014 will now be updated as expiring at the end of June. All of the original facts from the second half of 2014 are now pointing to the wrong entry in the Definition table.

To compensate for this particular data error, I used to include a fact *realignment* job that I inserted fairly late in the ETL job sequence to correct the errors created by this form of data collision. The correction is logically simple: find any fact, the Source Timestamp of which doesn't fall between the Effective and Expiration Timestamps of the dimension entries to which it is connected. While logically simple, the actual writing, testing, and operation of this procedure are quite cumbersome. First, there are over 20 dimensions that need to be checked, and there can be any arbitrary number of Definition entries per dimension that are connected to the facts through the bridge structure. In a fact table that might grow to include billions of rows, the overhead of such an undertaking causes concern.

Before I converted this design from using the Definition ID in the Bridge to using the Master ID—which made the fact realignment problem disappear—I worked to mediate the performance concerns associated with the realignment queries by using the staging tables here in the Definition pipe to drive the logic of realignment. Only facts connected to Definition rows that were actually undergoing slow-change management needed to be checked for realignment, dramatically reducing the scale of the realignment query processing. Once I committed myself to a Master ID-oriented algorithm for the Bridge entries, the need to be concerned with fact alignment disappeared completely, an example of the power of maximizing the late binding of data.

Multiple Simultaneous Transactions

The complexity of handling multiple transactions against the same Definition entry in the same ETL job is a challenge that you'll eventually need to confront. The issue could be avoided in the Alpha version, and even in the early Beta version loads, by planning and controlling the sourcing of data so only single transaction sources were processed by any given job. That approach becomes untenable as you move toward production status. It simply isn't possible to prevent multiple transactions from arriving against a single Definition entry. Even if you define controls to avoid the problem during first-generation warehouse loads, the problem will likely resurface as new sources are subsequently added to the warehouse in production.

A difficulty in handling these transactional classifications will be in recognizing when multiple types of definitional events come through the pipe at the same time. For example, suppose that your source table can handle multiple versions of patient definitions. If a patient was added to your source table at noon and then was updated at 2:00 p.m., there would be two source transactions in tonight's load. If that 2:00 p.m. update included the patient's last name and date of birth, the update transaction would be a mixture of SCD (last name) and non-SCD (date of birth) data for update.

Suppose further that this patient had been added to your warehouse as an orphan definition on a previous fact load. This patient identifier would be resolved in the Reference pipe, so the Placeholder Flag will be False and the Orphan Flag will be True. This combination typically is processed as an auto-adoption of the existing orphan row using the data in the transaction. In this case, the noon add transaction will auto-adopt the dimension orphan. The 2:00 p.m. update transaction will ultimately process as an update transaction when the auto-adoption turns off the Orphan Flag, leaving both Placeholder and Orphan flags as False by the time the second patient transaction is processed. It will all work, but it'll be more complex in the code than you might have otherwise expected.

The situation can also be compounded by extra transactions, although once the code handles two simultaneous transactions, it handles any number of transactions just as well. For clarity, imagine that your patient had been updated not only at 2:00 p.m. but also at 3:00 p.m., 4:00 p.m., and 5:00 p.m. You'd actually have four distinct update transactions instead of only one. To compound this, if the patient's date of birth (a non-SCD value) was updated in each transaction, you'd actually want to discard (not process or apply) all of the values except the final 5:00 p.m. value. (If you recoiled at that idea, it means you'd rather define that column as an SCD value, which you could do—but ask yourself what would cause a patient to have multiple dates of birth over time.)

As a single generic ETL subsystem, all of this cross-transactional complexity has to be handled by a single set of generic code. Under the older traditional model, each transaction or data exception could be handled in isolated code that specifically handled the few data sources in which these situations arose. The complexity of resolving all of the potential transaction collisions or overlaps was spread out over hundreds of ETL jobs, making it difficult to standardize or maintain. This generic structure doesn't add any new complexity, but it compresses the historical complexity into a much smaller space where it becomes more visible and easier to manage and maintain. You'll focus on the more essential typical cases here in the Beta version and leave some of the less typical transaction problems for the Gamma version.

There are many examples of simultaneous transactions showing up in a single ETL job, and they all share similar characteristics. A common example that I typically see fairly early in the Beta version is the dimensional load of diagnosis codes, particularly ICD-9 codes in the Diagnosis dimension.

Since ICD-9 codes change periodically, the system from which you'll extract them is likely to keep effective and expiration dates associated with the master data that you'll use as a data source. Extracting that data for a definition load into your new warehouse will entail correctly processing all of the various instances that will be available in the data. It doesn't matter how many different variations there are of any single code in the source structure. Many codes will have a single version, while others might have as many as a dozen. What matters is that there will be variations on the definitions that occur over time and that need to be reflected historically in the Definition table as a series of slowly-changing values.

To handle this complexity, the logic of this Definition pipe needs to be oriented around *sets* of data transactions, each set defined by the Master ID of the entry being maintained. For example, suppose you are trying to load a dataset that you've sourced containing master data about ICD-9 diagnosis codes. To keep it simple, let's suppose that no data of this type have previously been loaded. This means that all of the data from the source will be identified as new data in the Reference pipe (e.g., Placeholder Flag is on; Orphan Flag is off). For codes in the source that have not gone through any type of change, the simplified new entry logic described earlier would work fine. Each pivoted row of data could be inserted into the Definition table in Diagnosis, but for codes that have gone through one or more changes the one-row-at-a-time approach wouldn't work. The simple row-based logic would insert a new Definition row for each transaction in the source. Instead, you want to process the data as though the oldest transaction is the new data, and all of the other transactions are changes to that data. One problem is that the new data aren't on the database yet, so there's actually nothing to apply those changes to. Another problem is that we don't know in advance which transaction is the oldest, so we won't know which one to treat as the new data versus all of the others being treated as change transactions.

The situation gets even more complicated if you drop the presumption that no data are already present on the Definition table. In addition to having multiple transactions arriving in the source, you might already have multiple variations of the entry in the Definition table, as a result of previous loads. You could have multiple arriving transactions that need to be inserted among multiple existing rows in the Definition table. Some of the arriving data will have been flagged by the Reference pipe as new, while other values will be flagged as existing and in need of updates. Getting this right, including inserting new rows and adjusting effective and expiration dates of some of the existing rows, requires a rigorous process that will end up with all of the data temporally sequenced with correct gapless time periods.

To overcome this obstacle, you'll shift your logic in the Definition pipe to always use a set-oriented logic in favor of the simpler row-oriented logic you've used through Alpha and early Beta versions. As a set, there will be 1 – N source rows per transaction. You'll use the Source Timestamp from the source dataset to

sort those rows into the correct temporal order in order to correctly process them into the warehouse. In many cases, the set will be a single row because the data being processed have not been sourced through multiple transactions. By having logic that is set based, you won't need to know that structure for any given entry in advance.

Building SCD Transaction Sets

Implementing the notion of transaction sets for SCDs requires some additions and changes to the Definition staging tables described earlier. As additions, you'll need to add into the staging tables any already existing rows in the Definition tables for the Master IDs that are being impacted by the load, namely, those with the Placeholder Flag off. The changes required include determining the relative row number of each row in the set, so rows can be paired for establishing the Expiration Timestamp of earlier rows against the Effective Timestamp of subsequent rows. Together, these additions and changes will cause the staging table to contain the entire SCD set for each Master ID regardless of whether or not the row is new in this load. Keep in mind that rows previously loaded in the Definition table are likely to be changed as a result of SCD changes since their effective and expiration timestamps must always be reconciled across the entire set of data, and the status of the currently active row might be changed to superseded.

Staging Existing Definitions

For any Master ID in the wide staging area that is subject to SCD controls, the existing rows from the various Definition tables must be copied into the deep staging table, so the different rows in the resulting transaction set will be sequenced properly. This extra staging must be done before the pivot to the wide staging table occurs, so all necessary data are present. Since this logic is specific to each dimension, the necessary query will be fanned out, so there is a separate piece of logic for each dimension:

```
INSERT INTO #SOURCE_DEFINITION_DEEP

SELECT 0          AS SDI_JOB_ID,
       'Existing' AS PHYSICAL_DATASET,
       'Existing' AS LOGICAL_DATASET,
       'Existing' AS FACT_BREAK,
       'Existing' AS ENTERPRISE_ID,
       MASTER_ID   AS BREAK_ID,
       D.SOURCE_TIMESTAMP,
       'Existing'   AS DATAFEED_BREAK,
       'Existing'   AS ETL_TRANSACTION,
       'Existing'   AS TARGET_STATE,
       'Existing'   AS DATASET_TYPE,
```

```

'Diagnosis'      TARGET_DIMENSION,
'Existing'       AS DIMENSION_CONTEXT,
D.SUBDIMENSION AS TARGET_SUBDIMENSION,
'Existing'       AS CHANGE_TYPE,
'DIAGNOSIS_D'   AS TARGET_TABLE,
P.SOURCE_COLUMN AS TARGET_COLUMN,
P.SOURCE_VALUE AS VALUE
FROM
  (SELECT DISTINCT MASTER_ID
    FROM #SOURCE_DEFINITION_DEEP
   WHERE TARGET_DIMENSION = 'DIAGNOSIS'
     AND PLACEHOLDER_FLAG = 'N') S
INNER JOIN DIAGNOSIS_D
  ON S.MASTER_ID = D.MASTER_ID
 AND D.ORPHAN_FLAG = 'N'
) I -- Dimension Access
UNPIVOT
  ([VALUE] FOR SOURCE_COLUMN IN
  (
    CATEGORY,
    DESCRIPTION,
    FULL_DESCRIPTION,
    EXTENDED_DESCRIPTION,
    ABBREVIATION,
    VERNACULAR_CODE,
    GENDER_SPECIFICITY
  )
) AS UNPVT;

```

With these rows copied into the deep staging table before the deep data are pivoted into the wide staging table, no changes to that wide staging table build are required. The data already in the Definition tables will look like sourced transaction data in the processing logic. By inserting the data into the deep pipe before the wide-pipe pivot, the data are available for both column-based processing (e.g., establishing net changes to columns across multiple transactions) as well as row-based processing (e.g., establishing ordering of changes over time against definition entries).

Assigning Deep Row Numbers

Much of the processing of definition updates in this pipe is centered around the common requirement to manage values of individual columns as slowly-changing values, the temporal values of which are maintained for historical querying. However, not all of the values that have been sourced will necessarily be for Definition columns that are being managed as slowly-changing values. There will certainly be columns in many Definition rows that are meant to be overlaid with new values as they occur without regard to what previous values might have been

stored (i.e., columns for which multiple simultaneous values wouldn't make sense, like a person's date of birth) or that change too dynamically to be managed using SCD logic because of the volatility that would overwhelm dimension volumes.

The deep staging table might include many different transactional data values for any of these target columns that are not being managed dynamically. Each of these column-level transactions requires that an update be applied to all of the rows in the SCD transaction set for a Master ID. The net effect is that only the *last* ordered transaction for each of these columns will have any net effect on the contents of the Definition entries in the table since each update would overlay the data updated by the earlier transactions. As a result, you'll want to be able to identify the last update in the deep table before applying any updates, so all but the last one for any particular column can be discarded. Only the latest transaction for each of these sourced columns should actually be applied to the Definition table in the dimension.

```
UPDATE #SOURCE_DEFINITION_DEEP A
    SET COLUMN_DESCENDING = B.COLUMN_DESCENDING
SELECT COLUMN_NUMBER() OVER(PARTITION BY MASTER_ID
                            TARGET_DIMENSION, TARGET_TABLE, TARGET_COLUMN
                            ORDER BY SOURCE_DATETIME DESC) AS COLUMN_DESCENDING
FROM #SOURCE_DEFINITION_WIDE B
WHERE A.MASTER_ID          = B.MASTER_ID
    AND A.TARGET_DIMENSION = B.TARGET_DIMENSION
    AND A.TARGET_TABLE    = B.TARGET_TABLE
    AND A.TARGET_COLUMN   = B.TARGET_COLUMN
    AND A.SOURCE_TIMESTAMP = B.SOURCE_TIMESTAMP
    AND A.CHANGE_TYPE     = 'Type 1';
```

Once you've established the descending order of the column-based transactions, only the rows with a Column Descending value of one need to be applied to the database. In fact, if the *last* transaction in the set comes from an *existing* Definition row—meaning that *all* of the sourced transactions were late-arriving changes to the Definition—even that last transaction need not be applied to the database.

Distribute Deep Non-SCD Updates

Having established the relative order for which multiple updates to the same column have been sourced, you can now take that last sourced value—the one with the latest source effective timestamp that you actually want to apply to the Definition table—and use it to overlay any other earlier source value for the same Master ID:

```
UPDATE #SOURCE_DEFINITION_DEEP
    SET SOURCED_VALUE = B.VALUE,
        VALUE         = A.VALUE
FROM #SOURCE_DEFINITION_DEEP A
INNER JOIN #SOURCE_DEFINITION_DEEP B
    ON A.COLUMN_DESCENDING = 1
```

```

AND B.COLUMN_DESCENDING      > 1
AND A.MASTER_ID              = B.MASTER_ID
AND A.TARGET_DIMENSION       = B.TARGET_DIMENSION
AND A.TARGET_TABLE            = B.TARGET_TABLE
AND A.TARGET_COLUMN           = B.TARGET_COLUMN
AND A.CHANGE_TYPE             = 'Type 1'

```

As you overlay those values, you should also save the original sourced values so they are available when you start producing audit trails in the Gamma version. The advantage of doing this distribution of values in the deep staging table is that the logic of the assignments to each column is represented in the JOIN logic in the code. The name of the column being impacted is still in the data. Once you move on to the wide staging table, that column-by-column processing ability is lost. Applying these changes after that processing would require specific UPDATE statements to be written for each target column. Creating and maintaining that code is avoided by distributing the final value across all transactions here.

Now that you've added the set-processing capabilities to your deep staging table, it is time to pivot the deep staging table into the wide staging table, just as you did prior to shifting toward set-based processing. Because all of the changes you've made take place within the deep staging table, the actual PIVOT command to create the wide staging table is the same as before.

Assigning Relative Wide Row Numbers

When all the data for each transaction set is available in the wide staging table, relative row numbers can be assigned. You'll need to know the relative position of each row of the transaction set, both ascending and descending, in order to correctly set the effective and expiration timestamps and status indicators. Collectively, these rows constitute the set of data needed for a Master ID (Figure 11.3). Calculating the relative row numbers is accomplished by partitioning the wide staging table by Master ID, using both ascending and descending ordering against the Source Timestamp:

```

UPDATE #SOURCE_DEFINITION_WIDE A
    SET ROW_ASCENDING  = B.ROW_ASCENDING
        ROW_DESCENDING = B.ROW_DESCENDING
SELECT ROW_NUMBER() OVER(PARTITION BY MASTER_ID
                        ORDER BY SOURCE_DATETIME)          AS ROW_ASCENDING,
        ROW_NUMBER() OVER(PARTITION BY MASTER_ID
                        ORDER BY SOURCE_DATETIME DESC)     AS ROW_DESCENDING
    FROM #SOURCE_DEFINITION_WIDE B
    WHERE A.MASTER_ID          = B.MASTER_ID
        AND A.SOURCE_TIMESTAMP    = B.SOURCE_TIMESTAMP;

```

With the ascending and descending row numbers established, the Effective and Expiration Timestamps, as well as the Status Indicator, can be set for each row:

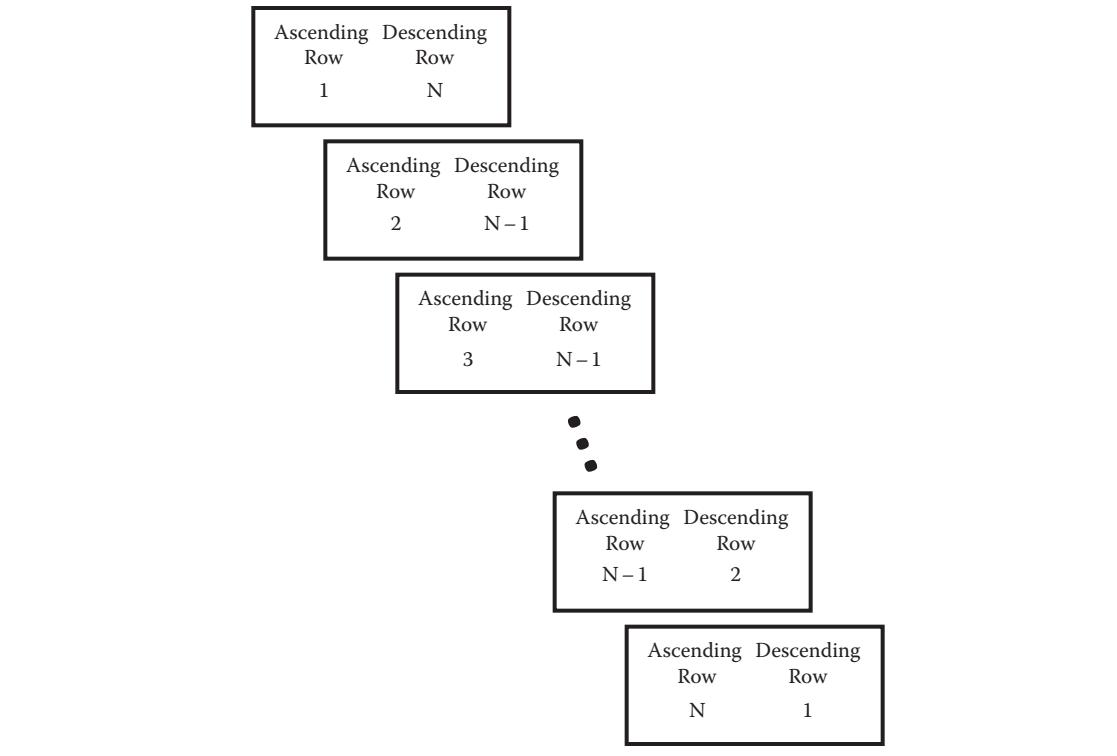


Figure 11.3 Ascending and Descending rows with Master ID set.

```

UPDATE #SOURCE_DEFINITION_WIDE
SET EFFECTIVE_DATETIME =
CASE WHEN B.ROW_ASCENDING = 1
THEN '1900-01-01'
ELSE B.SOURCE_TIMESTAMP
END,
EXPIRATION_DATETIME =
CASE WHEN B.ROW_DESCENDING = 1
THEN '9999-12-31 23:59:59.999'
ELSE DATEADD(MS, -D.SOURCE_TIMESTAMP)
END,
STATUS_INDICATOR =
CASE WHEN B.ROW_DESCENDING = 1
THEN CASE WHEN A.STATUS_INDICATOR = 'D'
THEN 'D'
ELSE 'A'
END
ELSE 'S'
END
FROM #SOURCE_DEFINITION_WIDE A
INNER JOIN #SOURCE_DEFINITION_WIDE B
ON A.MASTER_ID = B.MASTER_ID
AND A.SOURCE_TIMESTAMP = B.SOURCE_TIMESTAMP
LEFT JOIN #SOURCE_DEFINITION_WIDE D
ON B.MASTER_ID = D.MASTER_ID
AND B.ROW_ASCENDING + 1 = D.ROW_ASCENDING;

```

The Source Timestamp is assigned as the new Effective Timestamp for all rows except number one ascending row, which gets the default minimum timestamp as its Effective Timestamp. The Expiration Timestamp is set to 1 millisecond before the Effective Timestamp of the next sequenced row, except for the number one descending row, which is assigned the default maximum timestamp. The number one descending row is assigned the Active or Deleted status, with all others being marked Superseded. Note that this logic is not dependent upon knowing the order or status of any particular existing row prior to processing, nor is it dependent on knowing the number or type of transactions that are being processed simultaneously in the ETL process. It carries the extra overhead of accessing all of the previously existing rows in a Master ID set and reprocessing them as though they are inbound transactions, but the alternative would require extensive logic to establish the correct place for each sourced transaction in the existing set of rows.

The logic required for that kind of in-place sequencing has been problematic in many warehouse implementations I've work with, and it is usually very error prone. Accepting this small increase in overhead has solved that problem for me, and I urge you adopt this approach over any objections you receive from team members expressing concerns about performance. If you have data in one or more of your dimensions that changes often enough for these performance concerns to become real, you should reconsider using the slowly-changing feature to manage change and shift that data into the fact pipe. By definition, SCD logic is for slowly-changing values, and so the final performance of those transactions shouldn't be a major concern.

Applying Transactions to Dimensions

With the data cleaned up and properly sorted in the wide staging table, it is finally time for you to apply those changes in the Definition tables. There are two basic approaches to this processing: (1) apply the data as transactions to the tables using `INSERT` and `UPDATE` statements, or (2) `DELETE` any existing Definition entries for these Master IDs, and apply all of the data at one time using the database management system's bulk loader. Both approaches are viable, and I actually recommend allowing either approach so the warehouse support team can select which approach to use for any particular load, based on the requirements and expected performance of each individual source load (i.e., typically defined at the Physical Dataset level). However, here in the Beta version, I typically stick to the transactional approach because it allows for more incremental development and testing and helps you learn the model more thoroughly. The bulk load approach requires you to develop *all* of the logic that follows before you can effectively test *any* of that logic, because the database isn't impacted

until the very end. We'll return to the bulk loading approach in the Gamma version where you'll have the option to selectively load subsets of your sourced data in bulk or as transactions.

Obtain New Dimension IDs

New Definition rows being created as part of the processing of slowly-changing values must be assigned new Definition IDs. Most of the IDs are generated back in the Reference pipe as new dimension entries are established, but these identifiers are only needed when column-level SCD changes require splitting those original rows. In those original rows, the Master ID and Definition ID were the same value. Here, the Master ID will stay the same to maintain the integrity of the full set of data for the Definition, but a new Definition ID surrogate will be generated using the New Surrogates table in the same manner as it was used in the Reference pipe:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS = 'Beta New Definition IDs'
INSERT INTO NEW_SURROGATES (PROCESS, TARGET_DIMENSION, COMPOSITE)
SELECT DISTINCT
    'Beta New Definition IDs',
    TARGET_DIMENSION,
    CONCAT(MASTER_ID, '||', SOURCE_TIMESTAMP)
FROM #SOURCE_DEFINITION_WIDE
WHERE PHYSICAL_DATASET <> 'Existing'
    AND PLACEHOLDER_FLAG = 'N';
```

The logic looks for a new surrogate for any Master ID that had been found in the Reference pipe (e.g., Placeholder Flag turned off) for data that don't already exist on the Definition table. That value is then assigned to the staging table Definition ID, and the Placeholder Flag for that particular row is turned on, so the new row will be inserted:

```
UPDATE #SOURCE_DEFINITION_WIDE
SET DEFINITION_ID = NS.NEW_SURROGATE
PLACEHOLDER_FLAG = 'Y'
SELECT NEW_SURROGATE AS DEFINITION_ID
    FROM #SOURCE_DEFINITION_WIDE S
INNER JOIN NEW_SURROGATES NS
    ON NS.COMPOSITE = CONCAT(MASTER_ID, '||', SOURCE_TIMESTAMP)
    AND PROCESS = 'Beta New Definition IDs';
```

The load logic to insert new rows will now include these rows because the Placeholder Flag is on. It won't matter to the insertion logic whether the flag was set here or back in the Reference pipe.

Auto-Adopt Orphan Definitions

If the Master ID for an SCD is indicated in the staging table as being an Orphan, you will auto-adopt that row using the transaction data in the first ascending row of the SCD data in the staging table:

```

UPDATE DIAGNOSIS_D
    SET ORPHAN_FLAG          = 'N',
        REDACTION_CONTROL_FLAG = 'N',
        SOURCE_TIMESTAMP       = S.SOURCE_TIMESTAMP,
        PAY_SOURCE              = S.PAY_SOURCE,
        FINANCIAL_CLASS         = S.FINANCIAL_CLASS,
        DESCRIPTION              = S.DESCRIPTION,
        FULL_DESCRIPTION         = S.FULL_DESCRIPTION,
        EXTENDED_DESCRIPTION     = S.EXTENDED_DESCRIPTION,
        ABBREVIATION             = S.ABBREVIATION,
        VERNACULAR_CODE          = S.VERNACULAR_CODE
        GENDER_SPECIFICITY       = S.GENDER_SPECIFICITY
    FROM
        #SOURCE_DEFINITION_WIDE S
    INNER JOIN DIAGNOSIS_D D
        ON S.MASTER_ID      = D.MASTER_ID
        AND S.DIMENSION_ID   = D.DIMENSION_ID
    WHERE
        S.TARGET_DIMENSION    = 'Diagnosis'
        AND S.ROW_ASCENDING    = 1
        AND S.PLACEHOLDER_FLAG = 'N'
        AND S.ORPHAN_FLAG      = 'Y';

```

Insert New Definitions

You will then insert new rows from the SCD set inserted to the Definition table:

```

INSERT INTO DIAGNOSIS_D
SELECT DIMENSION_ID,
    MASTER_ID,
    TARGET_SUBDIMENSION,
    CATEGORY,
    DESCRIPTION,
    FULL_DESCRIPTION,
    EXTENDED_DESCRIPTION,
    ABBREVIATION,
    VERNACULAR_CODE,
    STATUS_INDICATOR,
    'N' AS ORPHAN_FLAG,
    'N' AS REDACTION_CONTROL_FLAG,
    'N' AS UNEXPECTED_FLAG,
    'N' AS UNDESIRED_FLAG,
    EFFECTIVE_TIMESTAMP,

```

```

EXPIRATION_TIMESTAMP,
SOURCE_TIMESTAMP,
GENDER_SPECIFICITY
FROM #SOURCE_DEFINITION_WIDE
WHERE TARGET_DIMENSION = 'Diagnosis'
AND PLACEHOLDER_FLAG = 'Y';

```

Update Existing Definitions

Finally, you'll apply updates to any existing rows in the SCD set:

```

UPDATE DIAGNOSIS_D
    CATEGORY          = S.CATEGORY
    DESCRIPTION       = S.DESCRIPTION,
    FULL_DESCRIPTION = S.FULL_DESCRIPTION,
    EXTENDED_DESCRIPTION = S.EXTENDED_DESCRIPTION,
    ABBREVIATION      = S.ABBREVIATION,
    VERNACULAR_CODE   = S.VERNACULAR_CODE,
    EFFECTIVE_TIMESTAMP = S.SOURCE_TIMESTAMP,
    EXPIRATION_TIMESTAMP = S.EXPIRATION_TIMESTAMP,
    STATUS_INDICATOR  = S.STATUS_INDICATOR,
    GENDER_SPECIFICITY = S.GENDER_SPECIFICITY
FROM
    DIAGNOSIS_D D
    INNER JOIN #SOURCE_DEFINITION_WIDE S
        ON D.MASTER_ID = S.MASTER_ID
        AND D.DIMENSION_ID = S.DIMENSION_ID
        AND D.PHYSICAL DATASET = 'Existing';

```

Performance Concerns

The code presented in this chapter illustrates the functionality needed to process all of the source definition data against new and existing entries in the Definition tables of the various dimensions. It is far from optimized. You'll want to optimize this logic according to the results of your data analysis. The critical variables you'll need to isolate in order to make those design choices will be the number of rows in the Definition table, and the volatility of the data in those rows. My experience during day-to-day operations of the ETL is that the number of rows that are changed during any one run of the ETL is very small relative to the total number of rows in the table.

Chapter 12

ETL Fact Pipe

The Fact pipe shares much in common with the Definition pipe, particularly with respect to the requirement that in order to load the facts, they'll need to be married to the Reference pipe for the dimensional connections. Where the data in the Definition pipe only need to be married to the one-dimensional Reference for the single Target dimension, the facts will ultimately need to be connected to all of the dimensions, only some of which will have been sourced into the Reference pipe. Specifically, the processing covered in the Fact pipe includes

- Transforming source-only intake data into source-to-target staged data through a join to the Metadata dimension, including Codeset translation of nonstandard source values
- Generating source-to-target staged data through a join to the Metadata dimension to pick up both valued and factless facts that have been defined against any inbound datasets
- Consolidating the roles, ranks, and weights of the reference entries for each fact in order to derive a single Group Composite for each sourced dimension for each fact
- Resolving a Group ID for each generated Group Composite, either because the Group ID already exists in the dimension or because it is created to support the current load
- Setting the Data State and Datafeed dimensions to the appropriately sourced values
- Resolving unsourced dimension entries for required dimensions, including resolving the Organization dimension to the sourced Enterprise ID, the Calendar dimension to the date in the Source Timestamp, and the Unit of Measure dimensions to Factless, Numeric, or Text depending upon an inspection of the actual fact value being processed
- Resolving all remaining unsourced dimensions to the Not Applicable entry in the dimension

- Inserting new fact values into the Fact table (recalling that handling multiple Fact value columns in multiple Fact tables has been deferred to the Gamma version)
- Changing the data state of older facts that might have been superseded by any newly arriving facts, including marking these new facts as Superseded if they are older than current instances of those facts from previous loads

The high-level sequence diagram in [Figure 12.1](#) illustrates these main steps in the processing. As with the Reference and Definition pipes described in the prior two chapters, the Fact pipe begins with the metadata transformation processing.

Metadata Transformation

The fact pipe requires two different transformations against the Metadata dimension: one to match sourced fact values to the required metadata to complete the source-to-target mapping and another to obtain the definitions of the factless facts defined in the metadata that have no sourced counterpart in the data being processed. Being factless, these facts are not sourced as values. The appropriate reference data is present in the Reference pipe for building these facts, but the second metadata transformation is required since the normal INNER JOIN to the Metadata dimension will be unable to select them.

Generating Sourced Facts

The first transformation for sourced fact values is based on the standard join logic used in the Reference and Definition pipes and includes Codeset translation of fact values obtained:

```
SELECT S.JOB_ID,
       S.PHYSICAL_DATASET,
       CASE WHEN S.LOGICAL_DATASET = R.CONTEXT_KEY_02
             THEN S.LOGICAL_DATASET
             WHEN S.LOGICAL_DATASET = '~All~'
                 THEN R.CONTEXT_KEY_02
                 ELSE S.LOGICAL_DATASET
             END LOGICAL_DATASET,
       CASE WHEN S.FACT_BREAK = R.CONTEXT_KEY_03
             THEN S.FACT_BREAK
             WHEN S.FACT_BREAK = '~All~'
                 THEN R.CONTEXT_KEY_03
                 ELSE S.FACT_BREAK
             END FACT_BREAK,
       S.ENTERPRISE_ID,
       S.BREAK_ID,
       S.SOURCE_TIMESTAMP,
```

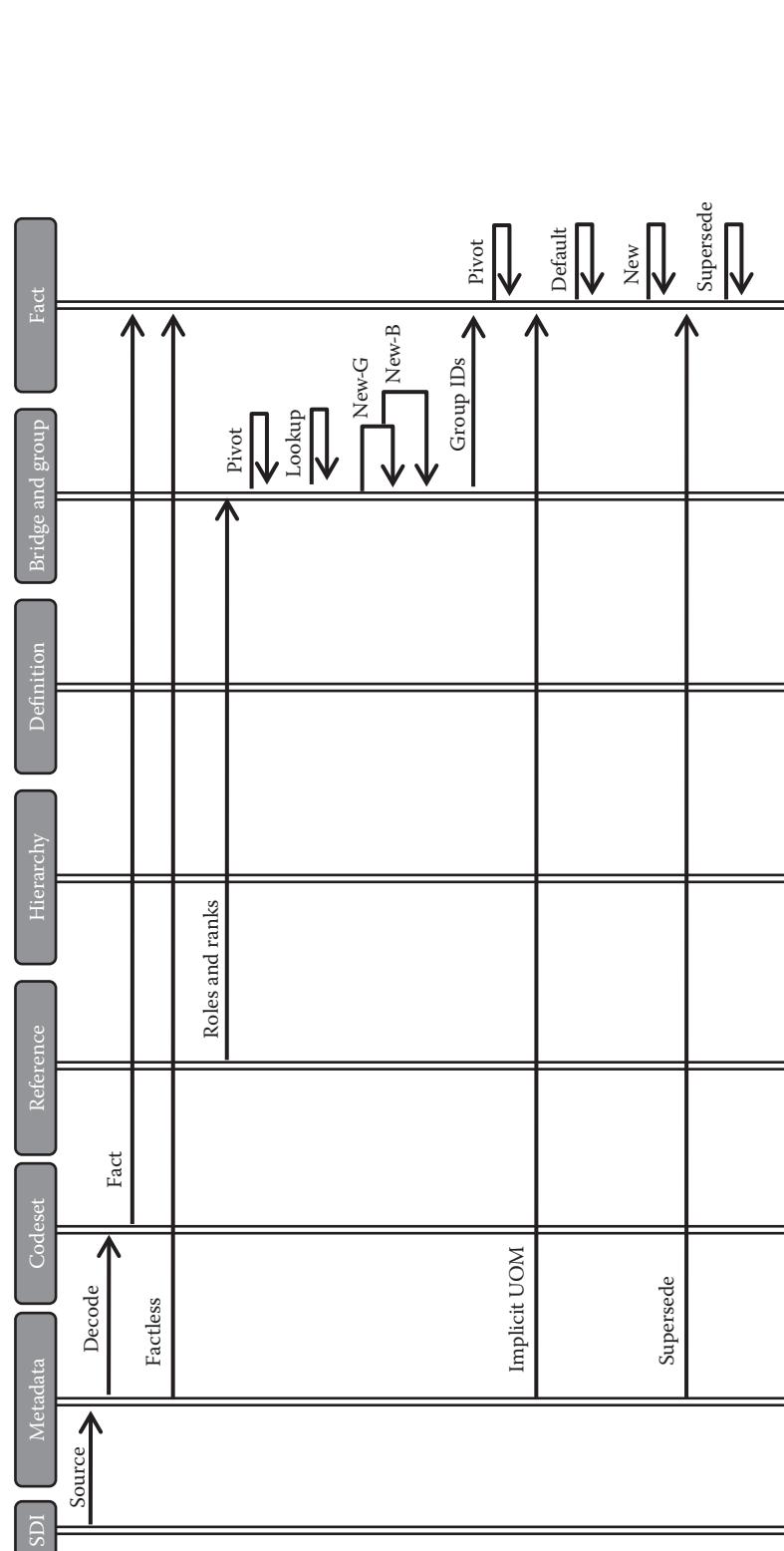


Figure 12.1 Sequence diagram for Fact pipe processing.

```

S.DATAFEED_BREAK,
S.ETL_TRANSACTION,
S.TARGET_STATE,
COALESCE(CD.DESCRIPTION, S.VALUE) AS VALUE,
D.DATASET_TYPE,
R.CONTEXT_KEY_05 [TARGET_TABLE],
R.CONTEXT_KEY_06 [TARGET_COLUMN],
D.IMPLICIT_UNIT,
D.IMPLICIT_MEASURE,
D.IMPLICIT_LANGUAGE
FROM STAGING.SOURCE S
INNER JOIN METADATA_R R
    ON S.PHYSICAL_DATASET = R.CONTEXT_KEY_01
    AND ( S.LOGICAL_DATASET = R.CONTEXT_KEY_02
        OR S.LOGICAL_DATASET = '~All~'
        OR R.CONTEXT_KEY_02 = '~All~' )
    AND ( S.FACT_BREAK = R.CONTEXT_KEY_03
        OR S.FACT_BREAK = '~All~'
        OR R.CONTEXT_KEY_03 = '~All~' )
    AND S.SOURCE_COLUMN = R.CONTEXT_KEY_04
INNER JOIN METADATA_D D
    ON R.MASTER_ID = D.MASTER_ID
    AND D.STATUS_INDICATOR = 'A'
LEFT JOIN CODESET_R CR
    INNER JOIN CODESET_D CD
        ON CR.MASTER_ID = CD.MASTER_ID
        AND S.SOURCE_TIMESTAMP
            BETWEEN CD.EFFECTIVE_TIMESTAMP AND CD.EXPIRATION_TIMESTAMP
        ON CR.CONTEXT_KEY_01 = D.CODESET_CONTEXT_KEY_01
        AND CR.CONTEXT_KEY_02 = D.CODESET_CONTEXT_KEY_02
        AND CR.CONTEXT_KEY_03 = S.VALUE
WHERE D.TARGET_TYPE = 'Fact'

```

Generating Factless Facts

Since there are no source dataset entries for these facts, the second transformation query needs to identify candidate transactions from the source data that can serve as proxies for those absent facts. Within the sourced reference data, the SDI Job ID, Physical Dataset, Logical Dataset, and Fact Break together represent the sourced fact transactions in the inbound source dataset. We don't know that any of these transactions, in particular, will require factless facts to be generated, but they serve as the list of candidates that *might* require them. These facts can't be generated by the standard metadata transformation join because, by definition, there isn't any sourced value to which to join the target metadata. As a result, this proxy transaction subset joins to the Metadata dimension in order to find any factless facts that have been defined against these Physical Dataset,

Logical Dataset, and Fact Break combinations. Any generated facts are added through a Union to the first metadata transformation output.

```

UNION
SELECT S.SDI_JOB_ID,
       S.PHYSICAL_DATASET,
       CASE WHEN S.LOGICAL_DATASET = R.CONTEXT_KEY_02
             THEN S.LOGICAL_DATASET
             WHEN S.LOGICAL_DATASET = '~All~'
                 THEN R.CONTEXT_KEY_02
                 ELSE S.LOGICAL_DATASET
             END LOGICAL_DATASET,
       CASE WHEN S.FACT_BREAK = R.CONTEXT_KEY_03
             THEN S.FACT_BREAK
             WHEN S.FACT_BREAK = '~All~'
                 THEN R.CONTEXT_KEY_03
                 ELSE S.FACT_BREAK
             END FACT_BREAK,
       S.ENTERPRISE_ID,
       S.BREAK_ID,
       S.SOURCE_TIMESTAMP,
       S.DATAFEED_BREAK,
       S.ETL_TRANSACTION,
       S.TARGET_STATE,
       '~Factless~' AS VALUE,
       D.DATASET_TYPE,
       R.CONTEXT_KEY_05 [TARGET_TABLE],
       R.CONTEXT_KEY_06 [TARGET_COLUMN],
       NULL AS IMPLICIT_UNIT,
       NULL AS IMPLICIT_MEASURE,
       NULL AS IMPLICIT_LANGUAGE
FROM
  (SELECT DISTINCT
      SDI_JOB_ID,
      PHYSICAL_DATASET,
      LOGICAL_DATASET,
      FACT_BREAK,
      ENTERPRISE_ID,
      BREAK_ID,
      DATAFEED_BREAK,
      ETL_TRANSACTION,
      TARGET_STATE,
      SOURCE_TIMESTAMP
   FROM #SOURCE1_REFERENCE ) S
INNER JOIN METADATA_R R
  ON S.PHYSICAL_DATASET = R.CONTEXT_KEY_01
  AND ( S.LOGICAL_DATASET = R.CONTEXT_KEY_02
        OR S.LOGICAL_DATASET = '~All~' )

```

```

        OR R.CONTEXT_KEY_02 = '~All~' )
    AND ( S.FACT_BREAK      = R.CONTEXT_KEY_03
        OR S.FACT_BREAK      = '~All~'
        OR R.CONTEXT_KEY_03 = '~All~' )
        AND R.CONTEXT_KEY_04 = '~Factless~'
INNER JOIN METADATA_D D
    ON R.MASTER_ID = D.MASTER_ID
    AND D.STATUS_INDICATOR = 'A'
WHERE D.TARGET_TYPE = 'Fact';

```

Note that the subquery that obtains the candidate transactions also pulls the Enterprise ID, Datafeed Break, ETL Transaction, Target State, and Source Timestamp. These extra columns don't affect the uniqueness of the candidate transactions selected; instead, they will be used later in order to conduct proper control processing while actually building the factless facts for insertion. The Fact pipe now includes all needed facts, whether sourced or factless. It won't make any difference to the rest of the processing in the Fact pipe which of these two transformation queries resulted in the Fact being present in the Fact pipe.

Staged Facts

The Facts that were generated from the earlier transformations can be staged for further processing after the foreign key portion of the wide fact entries has been resolved through the bridge and group process:

```

CREATE TABLE #SOURCE1_FACT (
    SDI_JOB_ID           INT,
    PHYSICAL_DATASET     VARCHAR(64),
    LOGICAL_DATASET      VARCHAR(64),
    DATASET_TYPE         VARCHAR(64),
    ENTERPRISE_ID        VARCHAR(64),
    BREAK_ID             INT,
    SOURCE_TIMESTAMP     DATETIME,
    DATAFEED_BREAK       VARCHAR(64),
    ETL_TRANSACTION      VARCHAR(64),
    TARGET_STATE          VARCHAR(64),
    VALUE                VARCHAR(64),
    DATASET_TYPE         VARCHAR(64),
    TARGET_TABLE          VARCHAR(64),
    TARGET_COLUMN         VARCHAR(64),
    IMPLICIT_UNIT        VARCHAR(64),
    IMPLICIT_MEASURE     VARCHAR(64),
    IMPLICIT_LANGUAGE    VARCHAR(64)
);

INSERT INTO #SOURCE1_FACT
<<< two-pass metadata transformation query HERE >>>

```

Bridges and Groups

The data in the Fact pipe will eventually be inserted into the Fact tables of the warehouse, but first, the data in the Reference pipe have to be processed into bridges and groups in order to build the dimensional references needed to populate the dimensional foreign keys that constitute the bulk of the columns in the Fact tables (Figure 12.2).

Group processing is very much like Reference processing. In your Reference pipe, you have Role, Rank, and Weight data for each Master ID that is needed here in the Fact pipe. Those values, along with the Master ID itself, will serve as a consolidated Bridge Composite. Each Bridge Composite is analogous to the Context Key columns in the Reference pipe. The Context Keys were pivoted into the needed Reference Composite so the logic for building reference entries didn't need to know the number of components needed to build any particular natural key. Here, you'll pivot the Bridge Composites into a single row per transaction-dimension in order to create a consolidated Group Composite, without having to know how many bridges are being included in any particular group, and then query the dimension Group table to see if the group already exists. If so, you'll obtain the Group ID for the group. If not, you'll turn on a Placeholder Flag indicating that the group needs to be created for the facts that need to reference it.

As you did with the Reference pipe, assign a unique sequence Group ID to each distinct Group Composite that has the Placeholder Flag turned on. You can then insert those group definitions into the appropriate dimension Group table. Those new groups are then joined back to the Reference pipe in order to insert the separate bridge entries required for those new groups. At that point, the staged groups (regardless of their Placeholder value) become the driver of the fact load.

Before you can insert facts into Fact tables, the Group IDs that serve as the foreign keys to the various dimensions need to be resolved. At this point, the Reference pipe has the needed Master IDs for all dimension entries, but there might be multiple Master IDs in the Reference pipe for any single dimension against any single Fact. This requires that the sets of Master IDs in the Reference pipe be condensed into Groups that can then be referenced by the facts themselves.

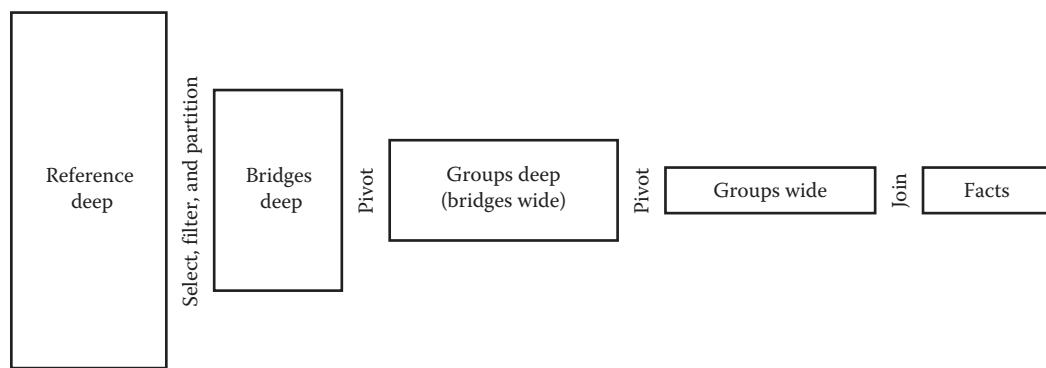


Figure 12.2 Processing flow from deep references to wide facts.

The logic of resolving bridges and groups will largely mimic the logic you used earlier to resolve Master IDs in the Reference pipe. You previously pivoted source Context Keys to define a Reference Composite value. The Reference Composite then was used to query the Reference table to get the Master ID, if found, or to turn on a Placeholder Flag, if not found. New References were then inserted for all placeholder keys. Note that since there are no dimension-specific columns in the Bridge and Group tables, the concept of Orphan processing doesn't apply to Group processing.

Bridge Staging

Here in the Fact pipe, you'll conduct a series of processing steps that are analogous to what you did back in the Reference pipe. The first step involves restaging the bridge data from the Reference pipe into a subset staging table that includes only fact-related reference data rows and in which an ordered bridge composite can be formatted:

```
CREATE TABLE #SOURCE1_BRIDGE (
    SDI_JOB_ID             INT,
    PHYSICAL_DATASET        VARCHAR(64),
    LOGICAL_DATASET         VARCHAR(64),
    DATASET_TYPE            VARCHAR(64),
    BREAK_ID                INT,
    TARGET_DIMENSION         VARCHAR(64),
    BRIDGE_ROLE              VARCHAR(64),
    BRIDGE_RANK              INT,
    BRIDGE_WEIGHT             FLOAT(3,1),
    ORDERRED_BRIDGE          VARCHAR(74),
    GROUP_ID                 INT,
    MASTER_ID                 INT,
    PLACEHOLDER_FLAG          CHAR(1)
);
```

The `INSERT` to this narrower staging table includes conditional logic that takes advantage of the single-bridge default Groups that were created for each Master ID back in the Reference pipe:

```
INSERT #SOURCE1_BRIDGE
SELECT SDI_JOB_ID,
    PHYSICAL_DATASET,
    LOGICAL_DATASET,
    BREAK_ID,
    TARGET_DIMENSION,
    BRIDGE_ROLE,
    BRIDGE_RANK,
    BRIDGE_WEIGHT,
    'BRIDGE_COMPOSITE' ||
```

```

COLUMN_NUMBER()
OVER(PARTITION BY SDI_JOB_ID, PHYSICAL_DATASET, LOGICAL_DATASET,
      BREAK_ID, TARGET_DIMENSION
      ORDER BY BRIDGE_ROLE, BRIDGE_RANK DESC) AS ORDERED_BRIDGE

MASTER_ID
  || '+' || BRIDGE_ROLE
  || '+' || BRIDGE_RANK
  || '+' || BRIDGE_WEIGHT AS BRIDGE_COMPOSITE,
CASE WHEN
      BRIDGE_ROLE = TARGET_DIMENSION
      AND BRIDGE_RANK = 1
      AND BRIDGE_WEIGHT = 1.0
      THEN MASTER_ID ELSE NULL END AS GROUP_ID,
MASTER_ID,
CASE WHEN
      BRIDGE_ROLE = TARGET_DIMENSION
      AND BRIDGE_RANK = 1
      AND BRIDGE_WEIGHT = 1.0
      THEN 'N' ELSE 'Y' END AS PLACEHOLDER_FLAG
FROM #SOURCE1_REFERENCE S
WHERE S.DATASET_TYPE = 'Fact'
AND S.TARGET_TYPE = 'Reference';

```

By directly assigning the default Group ID from the Master ID, it becomes possible to assign a Group to a fact–dimension interaction without the need to actually join to the Target dimension. Since the vast majority of Group references against your dimensions are likely to be single bridged, this logic will save a great deal of resources and improve the overall performance of your ETL job stream. In fact, even when you start using more multiple-bridge groups in the future as your informatics requirements and data mature, the extremely large Subject and Interaction dimensions will remain the dimensions that are least likely to require multiple-bridged groups. In my experience, this one piece of logic can reduce your access to the Group tables during fact processing by 80%–90%, a particularly valuable savings when you process tens of millions of facts at a time, each using four to seven dimension groups.

The establishment of a relative row number for the bridges with each group was accomplished during the building of the Bridge staging table through the PARTITION clause. This row number, which was appended to the “BRIDGE_COMPOSITE” literal to create an Ordered Bridge column, will drive the necessary pivot into a Group Composite. This final pivot will be into a wider Group staging table:

```

CREATE TABLE #SOURCE1_GROUP (
  SDI_JOB_ID          INT,
  PHYSICAL_DATASET    VARCHAR(64),
  LOGICAL_DATASET     VARCHAR(64),
  BREAK_ID            INT,
  TARGET_DIMENSION    VARCHAR(64),
  GROUP_COMPOSITE    VARCHAR(1000),
  GROUP_ID            INT,
  PLACEHOLDER_FLAG   CHAR(1)
);

```

Group Staging

Because of the default single-bridge Group IDs, the `INSERT` into the pivoted Group staging table need only include the subset of the Bridge staging data that still have no Master ID assigned to it:

```
INSERT #SOURCE1_GROUP
SELECT SDI_JOB_ID,
       PHYSICAL_DATASET,
       LOGICAL_DATASET,
       DATASET_TYPE,
       BREAK_ID,
       TARGET_DIMENSION,
       BRIDGE_COMPOSITE1
          || COALESCE('||' || BRIDGE_COMPOSITE2, '')
          || COALESCE('||' || BRIDGE_COMPOSITE3, '')
          || COALESCE('||' || BRIDGE_COMPOSITE4, '')
          || COALESCE('||' || BRIDGE_COMPOSITE5, '')
                           AS GROUP_COMPOSITE,
       GROUP_ID,
       PLACEHOLDER_FLAG
  FROM (#SOURCE1_BRIDGE
 WHERE S.PLACEHOLDER_FLAG = 'Y' ) U
PIVOT
(
    MIN(BRIDGE_COMPOSITE)
  FOR ORDERED_BRIDGE IN (
      BRIDGE_COMPOSITE1,
      BRIDGE_COMPOSITE2,
      BRIDGE_COMPOSITE3,
      BRIDGE_COMPOSITE4,
      BRIDGE_COMPOSITE5
  )
) P;
```

This sample code illustrates the `PIVOT` into the Group Composite using up to five Bridge Composites. In reality, you'll need to include enough Bridge Composites in this Pivot to support the maximum number of Bridges you expect to need in your largest groups. In my experience, most can easily be built with these five Bridge Composites, but you'll undoubtedly identify sources where one or more dimensions require much more than five entries. For example, large surgical teams can require dozens of Caregiver entries, or financial Encounter summaries might include dozens of diagnoses or procedures. If the requirement for the number of bridge entries per group starts to exceed the number you allow for in your ETL `PIVOT` statements, you'll be forced to enhance and retest your code. I use 50 bridges as my design limit, and I've never needed to exceed that number.

Fortunately, exceeding your planned number will never happen at run time. The number of bridges required to define a fact is embedded in the metadata you define during source analysis. If you need to define metadata that exceeds your previously chosen design threshold for group size, you can immediately initiate the needed code enhancements. Those enhancements must be placed into production prior to beginning to load the data for which your metadata identified the problem. In my experience, if you allow for 50 bridges, you should almost never encounter that issue.

Also keep in mind that the size of the Group Composite column in the database tables also constrains the potential results of this Group pivot, although less directly than the sheer number of bridges. Since the Role is a VARCHAR column, the Group Composite will comfortably handle many more bridges than might be calculated by simply using the column sizes as guidance. There are several strategies that will help you fit more bridges into each Group Composite, including selecting shorter Role values or choosing to omit the Weight from the Bridge Composite when the weight value is zero. Since bridge scale typically doesn't become an issue for a long time after implementation (i.e., It's often a second or third production year issue), I don't recommend trying to maximize Group Composite content until after the first production implementation.

Group Lookups

It's finally time for you to do your lookup of the Groups required for the facts in the Fact pipe. You've developed a lot of group-related code up to this point, so perhaps the actual final lookup will seem anticlimactic. That extra processing serves some key purposes, particularly with respect to reducing the actual number of groups that must be looked up at this point. These lookups are only to the multibridge groups in each dimension that represent a small fraction of the groups referenced by facts.

The Group lookup code is fanned out across the different dimensions since the target groups are in their own dimension tables. Using the Diagnosis dimension for the example, the code involves a simple JOIN to the Group table to retrieve the Group ID and turn off the Placeholder flag:

```
UPDATE #SOURCE1_GROUP
SET GROUP_ID = G.GROUP_ID,
PLACEHOLDER_FLAG = 'N'
FROM #SOURCE1_GROUP S
INNER JOIN DIAGNOSIS_G G
ON S.TARGET_DIMENSION = 'Diagnosis'
AND G.GROUP_COMPOSITE = S.GROUP_COMPOSITE;
```

If this lookup join seems uninteresting, that's okay. The Group tables are larger than the Definition tables in each dimension. You don't want them to

require special processing that would consume resources unnecessarily. I've had clients that tried to get fancy at this point, but I've never felt satisfied with their results.

In one case, the client merged all groups from all dimensions into a single Group table, adding a column for Dimension Name. The stated rationale was that the primary purpose of the Group table is for ETL lookups, so why not merge the data so the code that does the lookup wouldn't need to be fanned out? The code ends up using a single JOIN to the generic Group table that satisfies the lookup for all dimensions:

```
UPDATE #SOURCE1_GROUP
    SET GROUP_ID = G.GROUP_ID,
        PLACEHOLDER_FLAG = 'N'
  FROM #SOURCE1_GROUP S
INNER JOIN COMPREHENSIVE_G G
    ON G.DIMENSION_NAME = S.TARGET_DIMENSION
   AND G.GROUP_COMPOSITE = S.GROUP_COMPOSITE;
```

The code looks the same as the fanned-out code you've created, and the single set of code satisfies the requirement without having to create as many as 20 or more separate lookup queries. There's a certain logical elegance to the unifying approach, but I still don't recommend pursuing that design strategy.

My primary concern is the size of the Group tables and the likelihood that multibridged groups will be used in any particular dimension. The two largest dimensions in the data warehouse will typically be the Subject and Interaction dimensions. These dimensions are also the *least* likely to use multibridged groups. There simply aren't as many multipatient or multiencounter facts as there are almost any other multigroup-dimensional fact. Because of this, you'll rarely actually have to do a lookup of a multibridged group against these two dimensions. Most of the lookups involve the many smaller dimensions. The combined group design alternative creates a situation where the Group table is huge, including all Subject and Interaction groups, and the lookups that would have been against small individual group tables are now all against the one huge table. I simply don't see the benefit, so I argue against the consolidating approach.

My secondary concern about the consolidated group approach involves a desire to see the fanned-out dimension lookup executed in parallel when that is a capability of your ETL operating environment. I'd rather run 20 or more small parallel joins than one huge stand-alone join. Creating the fanned-out version of the join isn't difficult, and that code is remarkably stable over time. The only time the code has to be touched is when new dimensions are added to the warehouse, something that need only happen over an extended multiyear timeframe. In another case, a client left the Group table alone in each dimension, but tried to get very fancy in avoiding the need to create all of the fanned-out lookup joins.

They instead used a single massive JOIN statement that performed a LEFT JOIN against each Group table:

```
UPDATE #SOURCE1_GROUP
    SET GROUP_ID = COALESCE(DG.GROUP_ID, PG.GROUP_ID, TG.GROUP_ID ... SG.GROUP_ID),
        PLACEHOLDER_FLAG = 'N'
FROM #SOURCE1_GROUP S
LEFT JOIN DIAGNOSIS_G DG
    ON S.TARGET_DIMENSION = 'Diagnosis'
    AND DG.GROUP_COMPOSITE = S.GROUP_COMPOSITE
LEFT JOIN PROCEDURE_G PG
    ON S.TARGET_DIMENSION = 'Procedure'
    AND PG.GROUP_COMPOSITE = S.GROUP_COMPOSITE
LEFT JOIN TREATMENT_G TG
    ON S.TARGET_DIMENSION = 'Treatment'
    AND DG.GROUP_COMPOSITE = S.GROUP_COMPOSITE
    O
    O
    O
LEFT JOIN STRUCTURE_G SG
    ON S.TARGET_DIMENSION = 'Structure'
    AND SG.GROUP_COMPOSITE = S.GROUP_COMPOSITE;
```

My main concern with this approach is the complexity of the code. Why create and maintain a very complicated query when a set of small independent queries will do? This approach also precludes options for having the lookup for each dimension executed in parallel. I also haven't ever noticed any improvements in the performance of this query versus its more independent counterparts. For all of these reasons, I discourage this design alternative.

New Group Surrogates

With the lookup of Groups completed, you'll next need to assign new surrogate values to the Group IDs of the remaining undefined Groups. To accomplish this, you can insert the distinct Group Composites from all of the dimensions into the same New Surrogate table that was used throughout the Reference pipe to assign new identifiers:

```
DELETE FROM NEW_SURROGATES WHERE PROCESS = 'Beta Group IDs';

INSERT INTO NEW_SURROGATES (PROCESS, DIMENSION, COMPOSITE)
SELECT DISTINCT 'Beta Group IDs', TARGET_DIMENSION, GROUP_COMPOSITE
    FROM #SOURCE_GROUP
    WHERE PLACEHOLDER_FLAG = 'Y'
```

I find that client support teams are usually surprised by how few new groups need to be created on a day-to-day basis in production. There's a jump in new groups whenever a new type of fact that uses novel roles is introduced into the load. Likewise, a load of new definitions in a dimension will result in an increase

in new groups, even for long-used roles, against those new definitions. Those jumps typically smooth out within a few days or weeks of the new types of data being introduced. Once the warehouse is placed into production, and the data in the dimensions stabilize, the frequency of new groups during ETL executions falls to very small numbers in most cases.

Distribute Surrogates to Group IDs

The newly assigned Group IDs need to be distributed across all of the source transactions that make use of those groups. You can accomplish this with a simple UPDATE statement back to the Group staging table:

```
UPDATE #SOURCE1_GROUP
  SET GROUP_ID = NS.SURROGATE
  FROM #SOURCE1_GROUP S
 INNER JOIN NEW_SURROGATES NS
    ON NS.COMPOSITE = S.GROUP_COMPOSITE
   AND NS.DIMENSION = S.TARGET_DIMENSION;
```

The Group staging table contains rows only for the multipart groups that have been sourced and is an incomplete picture of the set of dimension groups needed for the entire range of sourced facts.

Distribute Group IDs to Bridges

All of the Group IDs, including those found on table lookups and those generated for newly defined groups, must be redistributed back to the transaction-level bridge entries in the Bridge staging table:

```
UPDATE #SOURCE1_BRIDGE
  SET GROUP_ID = G.GROUP_ID
  FROM #SOURCE1_GROUP G
 INNER JOIN #SOURCE1_BRIDGE B
    ON B.SDI_JOB_ID          = G.SDI_JOB_ID
   AND B.PHYSICAL_DATASET    = G.PHYSICAL_DATASET
   AND B.LOGICAL_DATASET     = G.LOGICAL_DATASET
   AND B.BREAK_ID            = G.BREAK_ID
   AND B.TARGET_DIMENSION    = G.TARGET_DIMENSION;
```

The Bridge staging table now contains all of the Group IDs for all sourced dimensional data needed to create facts.

Insert New Groups

Before moving on to build and INSERT new facts—the nominal main purpose of this Fact pipe—you need to actually insert the new Group

definitions into the Group table of each dimension. The `INSERT` statement is fanned out to each dimension, as usual:

```
INSERT INTO ACCOUNTING_G (GROUP_ID, GROUP_COMPOSITE)
SELECT GROUP_ID,
       GROUP_COMPOSITE
  FROM #SOURCE1_GROUP
 WHERE TARGET_DIMENSION = 'Accounting'
   AND PLACEHOLDER_FLAG = 'Y';
```

Alternatively, the inserts to the dimension Group tables can be accomplished directly from the New Surrogates table:

```
INSERT INTO ACCOUNTING_G (GROUP_ID, GROUP_COMPOSITE)
SELECT GROUP_ID,
       GROUP_COMPOSITE
  FROM NEW_SURROGATES
 WHERE PROCESS = 'Beta Group IDs'
   AND DIMENSION = 'Accounting';
```

Either way, the new Groups are created in the Group tables of the dimensions.

Insert New Bridges

You also need to insert all of the related bridges into the dimension Bridge tables. The fanned-out `INSERT` statement can be driven off the Bridge staging table:

```
INSERT INTO ACCOUNTING_B (GROUP_ID, MASTER_ID, ROLE, RANK, WEIGHT)
SELECT GROUP_ID,
       MASTER_ID,
       BRIDGE_ROLE,
       BRIDGE_RANK,
       BRIDGE_WEIGHT
  FROM #SOURCE1_BRIDGE
 WHERE TARGET_DIMENSION = 'Accounting'
   AND PLACEHOLDER_FLAG = 'Y';
```

Inserting these new bridges completes the Bridge and Group processing of the Fact pipe, leaving only the facts themselves to be processed.

Build Facts

While the Reference pipe had multiple rows per dimension per fact, the Bridge staging table now has only one row per dimension per fact, and that row has the required Group ID. To clarify, stating that the Bridge staging table has one row per dimension per transaction is true only if you filter that bridge table to look

only at the Ordered Bridge entries for the first bridge per dimension. The group staging table that actually would have had one row per transaction per fact can't be used for this purpose because only multibridged groups were transferred into that staging table, and that is not adequate for building facts. Instead, you use the Bridge staging table.

By pivoting all of the Group IDs in the staging table by transaction, you'll have all of the dimension foreign keys for each pending fact in a single row. You can then join the waiting staged fact values in order to actually create the Fact table entries. There are a few additional checks and balances that need to be done before the actual insertion to the Fact table, but most of these additional steps are data controls to assure each new fact is complete even when many aspects of the dimensionality of the fact are not sourced.

Bridge Pivot

The pivot of the bridge data serves as the initialization of the staging table for the wide fact set of foreign keys for processing. The query results in a single result row for each fact that needs to be inserted to any of the Fact tables:

```
INSERT INTO #FACT_WIDE
(
    SDI_JOB_ID
    ,BREAK_ID
    ,FACT_BREAK_ID
    ,SOURCE_DATETIME
    ,STATUS_INDICATOR
    ,ACCOUNTING_GROUP_ID
    ,ANNOTATION_GROUP_ID
    ,CALENDAR_GROUP_ID
    ,CAREGIVER_GROUP_ID
    ,CLOCK_GROUP_ID
    ,DATAFEED_GROUP_ID
    ,DIAGNOSIS_GROUP_ID
    ,FACILITY_GROUP_ID
    ,GEOPOLITICS_GROUP_ID
    ,INTERACTION_GROUP_ID
    ,MATERIAL_GROUP_ID
    ,METADATA_MASTER_ID
    ,OPERATION_MASTER_ID
    ,ORGANIZATION_GROUP_ID
    ,PATHOLOGY_GROUP_ID
    ,PATIENT_GROUP_ID
    ,PROCEDURE_GROUP_ID
    ,QUALITY_GROUP_ID
    ,SYSTEM_GROUP_ID
    ,UOM_GROUP_ID
    ,VALUE
)
SELECT
    P.SDI_JOB_ID                               AS SDI_JOB_ID,
    P.PHYSICAL_DATASET                         AS PHYSICAL_DATASET,
    P.LOGICAL_DATASET                          AS LOGICAL_DATASET,
    P.FACT_BREAK                                AS FACT_BREAK,
    P.BREAK_ID                                  AS BREAK_ID,
    NULL                                       AS SOURCE_DATETIME,
    NULL                                       AS STATUS_INDICATOR,
    P.ACCOUNTING_GROUP_ID                      AS ACCOUNTING_GROUP_ID,
```

```

P.ANNOTATION_GROUP_ID          AS ANNOTATION_GROUP_ID,
P.CALENDAR_GROUP_ID            AS CALENDAR_GROUP_ID,
P.CAREGIVER_GROUP_ID           AS CAREGIVER_GROUP_ID,
P.CLOCK_GROUP_ID                AS CLOCK_GROUP_ID,
P.DATAFEED_MASTER_ID           AS DATAFEED_MASTER_ID,
P.DIAGNOSIS_GROUP_ID           AS DIAGNOSIS_GROUP_ID,
P.FACILITY_GROUP_ID             AS FACILITY_GROUP_ID,
P.GEOPOLITICS_GROUP_ID         AS GEOPOLITICS_GROUP_ID,
P.INTERACTION_GROUP_ID          AS INTERACTION_GROUP_ID,
P.MATERIAL_GROUP_ID              AS MATERIAL_GROUP_ID,
P.METADATA_MASTER_ID            AS METADATA_MASTER_ID,
P.OPERATION_MASTER_ID           AS OPERATION_MASTER_ID,
P.ORGANIZATION_GROUP_ID         AS ORGANIZATION_GROUP_ID,
P.PATHOLOGY_GROUP_ID             AS PATHOLOGY_GROUP_ID,
P.PATIENT_GROUP_ID               AS PATIENT_GROUP_ID,
P.PROCEDURE_GROUP_ID             AS PROCEDURE_GROUP_ID,
P.QUALITY_GROUP_ID                 AS QUALITY_GROUP_ID,
P.SYSTEM_GROUP_ID                  AS SYSTEM_GROUP_ID,
P.UOM_GROUP_ID                     AS UOM_GROUP_ID,
NULL                                AS VALUE
FROM
(SELECT DISTINCT
B.SDI_JOB_ID,
B.PHYSICAL_DATASET,
B.LOGICAL_DATASET,
BFACT_BREAK,
BBREAK_ID,
CASE WHEN B.TARGET_DIMENSION_NAME IN ('Metadata', 'Operation', 'Datafeed')
      THEN B.TARGET_DIMENSION + '_MASTER_ID'
      ELSE B.TARGET_DIMENSION + '_GROUP_ID'
      END    AS COLUMN_NAME,
B.GROUP_ID      AS COLUMN_VALUE
FROM #SOURCE1_BRIDGE B
WHERE ORDERED_BRIDGE = 'BRIDGE_COMPOSITE1' ) U
PIVOT (MAX(COLUMN_VALUE)
FOR COLUMN_NAME IN (
ACCOUNTING_GROUP_ID,
ANNOTATION_GROUP_ID,
CALENDAR_GROUP_ID,
CAREGIVER_GROUP_ID,
CLOCK_GROUP_ID,
DIAGNOSIS_GROUP_ID,
FACILITY_GROUP_ID,
GEOPOLITICS_GROUP_ID,
INTERACTION_GROUP_ID,
MATERIAL_GROUP_ID,
METADATA_MASTER_ID,
OPERATION_MASTER_ID,
ORGANIZATION_GROUP_ID,
PATHOLOGY_GROUP_ID,
PATIENT_GROUP_ID,
PROCEDURE_GROUP_ID,
QUALITY_GROUP_ID,
SYSTEM_GROUP_ID,
UOM_GROUP_ID
)
) AS P;

```

The main challenge in the earlier query is that the PIVOT clause needs to be provided with a unique column name for each of the dimension identifiers that will be generated in the wide view. Most of those identifiers will be Group IDs, while a few will be Master IDs. A simple case statement is used to append the appropriate group or master identifier name onto the end of the Target dimension.

Value Alignment

At this point, the new fact rows have been initialized in the staging table without the corresponding fact values, while the fact values are staged separately. It's time to bring those two staging tables into an alignment that finalizes the staging of fact data:

```
UPDATE #FACT_WIDE
    SET VALUE          = V.VALUE,
        SOURCE_DATETIME = V.SOURCE_TIMESTAMP,
        STATUS_INDICATOR = 'Active'
  FROM #FACT_WIDE F
 INNER JOIN #SOURCE1_FACT V
    ON S.SDI_JOB_ID           = V.SDI_JOB_ID
   AND S.PHYSICAL_DATASET     = V.PHYSICAL_DATASET
   AND S.LOGICAL_DATASET      = V.LOGICAL_DATASET
   AND S.FACT_BREAK           = V.FACT_BREAK
   AND S.BREAK_ID              = V.BREAK_ID;
```

The facts have now been prepared as much as possible using data obtained from the source datasets. Still missing are values that need to be initialized because they are necessary for the facts to be complete for insertion, but they weren't provided by any source.

Finalize Dimensions

With the fact row built, it's now possible to review each fact for any dimensionality that has not been sourced (Figure 12.3). Dimensions not sourced will remain as NULL in the fact rows that were just built. Since you don't want any NULL foreign keys in any of your facts, those columns need to be set to something before you actually insert them into the Fact table.

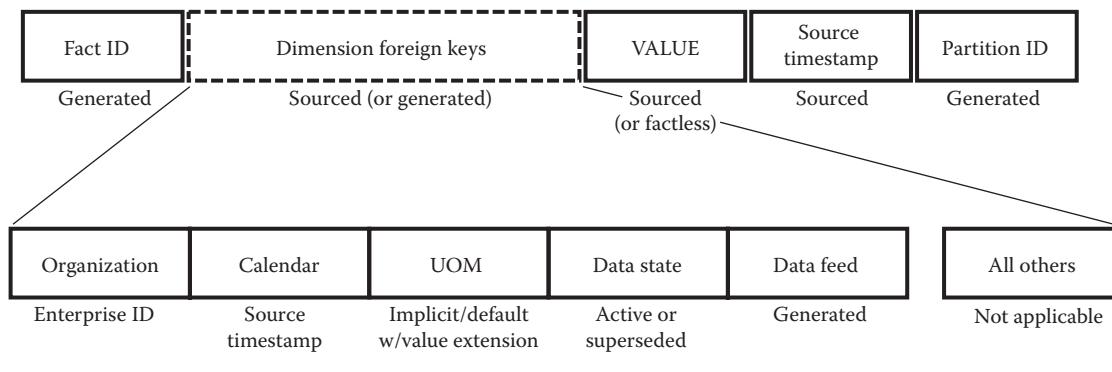


Figure 12.3 Finalizing columns of fact build.

Unit of Measure

Proper interpretation of a value in the Fact table requires that the Unit of Measure (UOM) dimension be properly set for that value. Two scenarios are possible at this point: (1) The Group ID is set because an explicit UOM was sourced, and a refinement to delimit the actual fact value might be needed, or (2) the Group ID in the wide fact row is still NULL and needs to be resolved. These two scenarios are mutually exclusive.

If the UOM Group ID in a fact row is not NULL, the source explicitly identified the unit of measure for the fact. The UOM is a Unit, a Measure, or a Language. If you've loaded Value rows into the UOM Dimension for the UOM in the fact, you'll need to check to see whether the particular sourced Value has a matching Value row in that UOM:

```
UPDATE #FACT_WIDE
  SET UOM_GROUP_ID = R2.MASTER_ID
  FROM #FACT_WIDE_PIPE F
  INNER JOIN UOM_B B
    ON F.UOM_GROUP_ID = B.GROUP_ID
    AND B.BRIDGE_ROLE = 'Value'
  INNER JOIN UOM_D D
    ON B.GROUP_ID IS D.MASTER_ID
    AND D.MASTER_ID = D.DEFINITION_ID
    AND D.SUBDIMENSION IN ('Unit', 'Measure', 'Measure-Language', 'Unit-Language')
  INNER JOIN UOM_R R
    ON D.MASTER_ID = R.MASTER_ID
  INNER JOIN UOM_R R2
    ON R2.REFERENCE_COMPOSITE = CONCAT(R.REFERENCE_COMPOSITE, ' | ', F.VALUE)
  INNER JOIN UOM_C C2
    ON R2.CONTEXT_ID = C2.CONTEXT_ID
    AND C2.CONTEXT_NAME = 'Value'
 WHERE UOM_GROUP_ID IS NOT NULL;
```

This update to the fact pipe only affects rows where there is a Value row in the UOM that is defined as more specific than the UOM that was received from the data source. It also takes advantage of a few important characteristics of the standard design pattern for the dimension. First, it requires that the Master ID and the Group ID of the UOM data be the same, meaning that no complex group is present for the fact. You don't want to apply the update to the UOM foreign key if the UOM group was modified anywhere in this ETL because of an error in the data (e.g., unexpected text in a quantitative value, or an out of range value received). In these modified cases, the UOM Group ID remains unchanged. Second, the update takes advantage of the common structuring of Reference Composite values in the Reference table. Regardless of which of the four possible contexts has been used to define the sourced UOM, the Reference Composite for the prospective Value row would simply append a single pipe and the Value. That combined Reference Composite can only belong to the desired Value row, but if the UOM Group ID is still NULL, the Group ID must be set either implicitly through the metadata or by interrogating the source fact value.

If the UOM Group ID is NULL in the wide fact row, the two remaining options are to (1) use an implicit UOM that might have been defined by the source analyst in the Metadata dimension or (2) default the UOM based on a rudimentary inspection of the content of the value column. Obtaining the UOM from the metadata is a new capability in the Beta version.

Implicit UOM

The Alpha version of the Fact loader provided two different mechanisms for assigning a unit of measure (UOM) to each fact: (1) The UOM could be explicitly sourced as part of the SDI dataset, or (2) the Fact loader, on seeing a NULL entry in the UOM dimension identifier in the fact after the final reference pivot, would default the UOM to either Factless, Numeric, or Text based on an inspection of the value column. This approach guaranteed that every fact would have a unit of measure, but was less than satisfactory if you want a specific unit of measure for data that didn't have an explicit UOM defined in the source. Since most source data don't have an associated explicit unit of measure, most data were loaded in the Alpha version using default units of measure.

In some cases, you can get around this issue by sourcing a UOM column as a literal text value in the SDI query. Using some form of case statement, a forced value could actually be made fairly sophisticated, and these values would then result in the ETL treating the UOM as explicitly defined. The cost, though, is an increase in the complexity of the SDI queries and in the analysis that would be needed to keep those literal values correctly aligned with the units stored in the UOM dimension. For example, as long as the alignment is maintained, a source query could create a Unit column containing "mmHg" for millibars of mercury and a Measure column containing "Systolic Blood Pressure." Metadata would have to be added for these new columns in the ETL load in order to properly map them to the explicit entry in the UOM dimension. As far as the ETL is concerned, the UOM columns were in the source dataset.

Instead of needing to modify the source intake queries to try to provide for UOM entries not actually available in the source datasets, you can expand UOM functionality to include the concept of an *implicit* UOM in the metadata. The implicit UOM gets applied to a fact if there is no explicit UOM identifier provided by the source. The default UOM created by inspecting the value column only gets used if no implicit UOM is provided in the metadata for the fact. There are now three opportunities to set a fact's unit of measure: (1) explicit in the source, (2) implicit in the metadata, or (3) defaulted by the Fact loader.

If, when setting up the metadata for a particular fact, the analyst knows the desired unit of measure, then it is coded as part of the fact's metadata without any need to add derived or forced source columns to the SDI query. In this way, a source loading Systolic Blood Pressure correctly assigns that UOM, while the Alpha version would have placed the value in the Numeric unit based only on an inspection of the fact value as being a number.

Note that the implicit UOM supplied in the Metadata definition for a fact will apply to all of the facts aligned to that Metadata ID that don't load with an explicit UOM. If your fact types are very specific, the implicit UOM can be very effective. For example, if the Metadata defines a very specific Oral Temperature fact, the implicit unit can be effectively assigned as Degrees Celsius. As your sourced facts or metadata become more generic, it becomes harder to use the implicit UOM feature. Generally, you will use the implicit UOM feature to populate a UOM for a fact that your data analyst has determined to be appropriate for a fact, but otherwise is unavailable in the source dataset. Since the trend in automated data is to include explicit units where available, you'll find that you make the most use of the implicit UOM feature when the units are Text, and you want to specify a more specific semantic measure or expected language. Text data rarely arrive at the warehouse with UOM data assigned. Implicit UOM is a tool for gaining some control over that wealth of data that otherwise simply falls into the highly generic text UOM.

Assigning the Implicit UOM

The metadata-based implicit UOM is taken from the Metadata for the fact if the implicit UOM columns are populated. This requires resolving the implicit UOM data in the Metadata dimension, when present, for all source datasets:

```
UPDATE #FACT_WIDE
  SET UOM_GROUP_ID = COALESCE(R2.MASTER_ID, R.MASTER_ID)
  FROM #FACT_WIDE_PIPE F
  INNER JOIN UOM_R R
    ON R.REFERENCE_OOMPOSITE =
      CONCAT(F.IMPLICIT_UNIT,
              ISNULL(CONCAT(' | ', F.IMPLICIT_MEASURE), ''),
              ISNULL(CONCAT(' | ', F.IMPLICIT_LANGUAGE), ''))
  LEFT JOIN UOM_R R2
    ON R.REFERENCE_OOMPOSITE =
      CONCAT(F.IMPLICIT_UNIT,
              ISNULL(CONCAT(' | ', F.IMPLICIT_MEASURE), ''),
              ISNULL(CONCAT(' | ', F.IMPLICIT_LANGUAGE), ''),
              ' | ' F.VALUE)
  WHERE F.UOM_GROUP_ID IS NULL
    AND F.VALUE IS NOT NULL;
```

Note that the *implicit* UOM never overrides a sourced value, so it is an effective way to provide a default unit of measure for data sources in which a UOM entry is defined but not mandatory. It also checks for a Value row in the UOM dimension for the specified implicit UOM, substituting that Group ID, if found. If the logic for implicit UOM entries doesn't resolve a UOM for the fact value,

then a final *default* UOM is assigned by looking directly at the fact value in order to assign one of three possible values:

1. If the fact value is Factless, then the Factless unit is assigned.
2. If the fact value is Numeric, then the Numeric unit is assigned.
3. Otherwise, the UOM is simply set to the Text unit.

```
UPDATE #FACT_WIDE
SET UOM_GROUP_ID =
CASE WHEN F.VALUE = '~Factless~' THEN
    (SELECT MASTER_ID FROM UOM_R WHERE REFERENCE_COMPOSITE = 'Factless')
    ELSE WHEN NUMBER(F.VALUE) IS TRUE THEN
        (SELECT MASTER_ID FROM UOM_R WHERE REFERENCE_COMPOSITE = 'Numeric')
        ELSE
            (SELECT MASTER_ID FROM UOM_R WHERE REFERENCE_COMPOSITE = 'Text')
        END
FROM #FACT_WIDE F;
```

In this manner, the UOM of a Fact will never be NULL, because the most functionally relevant UOM will be assigned based on a logical COALESCE of *explicit* source, *implicit* metadata, or *default* data typing assignments.

Calendar and Clock

All facts must be dimensionalized against the Calendar dimension, and many will also be dimensionalized against the Clock dimension. If the Calendar Group ID in a fact row is not NULL, then no further action is needed in these two dimensions, but if the Calendar Group ID is NULL, it must be set at this point. The calendar entry you use will be the date component of the Source Timestamp associated with the fact row. If you set the Calendar dimension in this way, you should also check the Clock dimension. If it is NULL, set it using the time portion of the Source Timestamp. If the Calendar entry wasn't NULL, meaning it had been sourced explicitly, then allow the Clock entry to remain NULL. Put another way, you only default the Clock if you are already defaulting the Calendar.

The updates required to set these dimensions are very basic. Your first update should be to the Clock Group ID since that update is also dependent upon the Calendar Group ID still being NULL:

```
UPDATE #FACT_WIDE F
SET CLOCK_GROUP_ID = (
    SELECT R.MASTER_ID
    FROM CLOCK_R
    WHERE REFERENCE_COMPOSITE = PARSE(F.SOURCE_TIMESTAMP)
)
WHERE CALENDAR_GROUP_ID IS NULL
    AND CLOCK_GROUP_ID IS NULL;
```

You can then update the Calendar Group ID, as needed:

```
UPDATE #FACT_WIDE F
  SET CALENDAR_GROUP_ID = (
    SELECT R.MASTER_ID
      FROM CALENDAR_R
     WHERE REFERENCE_COMPOSITE = TRUNC(F.SOURCE_TIMESTAMP)
  )
 WHERE CALENDAR_GROUP_ID IS NULL;
```

It might seem unusual to have source datasets that don't explicitly associate a Calendar or Clock entry with each fact, but it happens often enough to be useful. Snapshot facts often don't have a date other than the Source Timestamp simply because the designers of the source systems didn't include these timestamps. One option is to source the Effective Timestamp as a sourced column, with its own metadata mapping it to the Calendar and Clock dimensions. I don't recommend this as an approach because it is very wasteful of resources in the Reference pipe and increases the scale of the main pivot of the Reference pipe data. If your intent is to dimensionalize a Fact against the Source Timestamp, I recommend allowing this cleanup logic to do the assignments for you. Mapping the timestamp column from the source twice into every transaction would potentially add hundreds of thousands, or even millions, of extra rows of data to the Reference pipe during execution that simply wouldn't be needed.

Organization

All facts must be dimensionalized against the Organization dimension. If no source data were provided for the Organization, the Group ID for that dimension will still be NULL at this point. Those rows need to be updated using the Enterprise ID from the source dataset as the default organizational setting:

```
UPDATE #FACT_WIDE
  SET ORGANIZATION_GROUP_ID = R.MASTER_ID
  FROM #FACT_WIDE_PIPE F
 INNER JOIN ORGANIZATION_R R
   ON R.REFERENCE_OOMPOSITE = F.ENTERPRISE_ID
 INNER JOIN ORGANIZATION_C C
   ON R.CONTEXT_ID = C.CONTEXT_ID
   AND C.CONTEXT_NAME = 'Enterprise'
 WHERE F.ORGANIZATION_GROUP_ID IS NULL;
```

The Gamma version will alter the way the Organization dimension is assigned for a default value, based on the security requirements that are defined for the warehouse as implementation moves forward. You'll conduct more analysis on your security requirements as the implementation progresses, but the default

assignment of the Enterprise ID provides at least an initial level of control for anticipating which organization owns, or is responsible for, each fact.

Optional Dimensions

All remaining dimensions that can be sourced but that remain NULL in any Fact should be set to point to the Not Applicable system row in each relevant dimension. The update to accomplish this must be fanned out for each dimension group or master identifier in the fact row:

```
UPDATE #FACT_WIDE
    SET DIAGNOSIS_GROUP_ID = 1
    FROM #FACT_WIDE_PIPE
    WHERE DIAGNOSIS_GROUP_ID IS NULL;
```

There will be circumstances where you initialize a dimension to something other than the Not Applicable entry, but those variations will not be implemented until the Gamma version. If you have a short-term need to be able to use another value, I recommend adding that data to the explicitly sourced data so the entry you desire will be assigned by the source, and this default logic will not be involved. In the Gamma version, when you add more sophisticated logic to the defaulting of these keys in the fact row, you can go back and stop sourcing those values.

Set Control Dimensions

The warehouse control dimensions are defined as part of the implementation of the warehouse and are typically not under the control of the source jobs that provide data to the ETL streams.

Data State

The data state for a fact can be defined as part of the source data intake process through the optional Target State parameter of the standard SDI output table. That source column is typically left NULL by most SDI jobs, resulting in the Active state being used by default:

```
UPDATE #FACT_WIDE
    SET DATA_STATE_MASTER_ID = R.MASTER_ID
    FROM #FACT_WIDE_PIPE F
    INNER JOIN DATA_STATE_R R
        ON R.REFERENCE_OOMPOSITE = COALESCE(F.TARGET_DATA_STATE, 'Active')
    WHERE F.DATA STATE MASTER ID IS NULL;
```

If sourced, the target data state must be present in the Data State dimension, and its Master ID will be used to populate the Data State Master ID in the fact.

Data state will receive much more attention in the Gamma version. Until then, allowing the value to default to Active will be the most typical action taken when sourcing fact data.

Datafeed

The Datafeed dimension is set according to internal design constraints built into the ETL pipes. The source data intake process has very little input or control over how Datafeeds get defined. That doesn't mean that the source intake has no impact on datafeed details. In particular, the SDI intake format includes a Datafeed Break column that can extend the standard definition of datafeed controls. If the Datafeed Break is NULL, the number of rows in the Datafeed dimension is completely determined by the internal controls.

If a Datafeed Break is provided by the source intake, the number of Datafeed rows grows, multiplying by the number of distinct values that occur in the Datafeed Break for the physical–logical dataset in which the Datafeed Break is sourced. Providing a Datafeed Break is a means to increase the level of detail against which source data and control counts are tracked in the ETL. Since the Datafeed dimension isn't typically queried by users, I recommend challenging any use of the Datafeed Break for a source intake in order to determine that the value is actually only useful for data control. My experience is that many of the Datafeed Break assignments might have been considered for normal functional dimensionalization because the values represented something that an analytical user of the warehouse might want to include in a query. The choice is implementation specific, but unless the value in the Datafeed Break is a systems-level artifact, I recommend using dimensionalization instead.

Insert Fact Values

The only column not initialized in each Fact up to this point is the Partition Key. Early in the Beta version, I typically leave this column NULL, but this results in the table not being partitioned by the database management system. If you've identified a partitioning strategy at this point, go ahead and set the Partition Key before you insert your new facts. Typically you'll insert a bunch of data, perhaps 10–50 million facts, without any partitioning so you can monitor performance and develop an effective partition strategy to meet the needs of your actual data.

A common generic partition strategy that I've used successfully is to set the Partition Key to a calendar period that will help split the data into time-sensitive partitions. The size of an optimal partition will vary based on the span of time over which you are loading data into the Fact table. To keep the concept generic, I typically place the Source Timestamp for the fact, truncated to either the year alone or the year–month, into the Partition Key. This results in separate database

partitions for each year or month of Fact data. Since many analytical queries involve data for recent months, or perhaps the last couple of years, this greatly limits the portion of the Fact table that might need to be accessed to satisfy a particular query (presuming that you are loading data that span many years).

At this point, you are ready to insert your new facts into the Fact table. All of the necessary data have been sourced or generated and can now be inserted directly into the Fact table:

```
INSERT INTO FACT
SELECT ACCOUNTING_GROUP_ID
      ,ANNOTATION_GROUP_ID
      ,CALENDAR_GROUP_ID
      ,CAREGIVER_GROUP_ID
      ,CLOCK_GROUP_ID
      ,DATAFEED_GROUP_ID
      ,DIAGNOSIS_GROUP_ID
      ,FACILITY_GROUP_ID
      ,GEOPOLITICS_GROUP_ID
      ,INTERACTION_GROUP_ID
      ,MATERIAL_GROUP_ID
      ,METADATA_MASTER_ID
      ,OPERATION_MASTER_ID
      ,ORGANIZATION_GROUP_ID
      ,PATHOLOGY_GROUP_ID
      ,PATIENT_GROUP_ID
      ,PROCEDURE_GROUP_ID
      ,QUALITY_GROUP_ID
      ,SYSTEM_GROUP_ID
      ,UOM_GROUP_ID
      ,VALUE,
      ,SOURCE_TIMESTAMP,
      ,NULL AS PARTITION_KEY
)
FROM #FACT_WIDE
WHERE TARGET_TABLE = 'Fact';
```

If you've included multiple Fact tables in your design, you'll have to condition the inserts on the Target Table, and repeat the necessary Insert for each table (as a fanned-out structure, just like with your dimension maintenance).

Superseding Facts

As a result of adding all of the new facts received in this ETL job, there will be many previously loaded facts that are obsolete and need to be updated in order to indicate that they have been Superseded by a more recent active value. The logic for this is straightforward, but is not completely generic. There are several standard updates that will work well based on information in the

Metadata dimension. However, there will also be situations that arise in your analysis that don't fit one of these standard updates, so you'll need to build additional code to handle your nonstandard scenarios. The list I offer here has evolved over considerable time and tends to handle many of the more common situations.

Metadata Review

Let's start by reviewing the metadata entries that are relevant to the superseding of facts. The following data elements were defined in Chapter 7 (Table 12.1). These parameters support your ability to keep older versions of factors from getting in the way of an analytical user's ability to query facts that are the most current. Previous facts along the identified dimensions will be marked Superseded when newer Active facts are created. These columns will be NULL if no superseding is intended by the source data analyst.

The general update query for superseding facts takes a list of Fact IDs from an inner query that always returns a list of Fact IDs to be superseded:

```
UPDATE #FACT_WIDE
SET DATA_STATE_MASTER_ID =
    (SELECT MASTER_ID FROM DATA_STATE_R WHERE REFERENCE_COMPOSITE = 'Superseded')
WHERE F.FACT_ID IN
    (SELECT FACT_ID FROM
        (SELECT DISTINCT FACT_ID, ACTIVE_RANK FROM
            (
                << Embed Supersede Ranking Query Here >>
            )
        ) X
    WHERE X.ACTIVE_RANK > 1);
```

The UPDATE statement changes the data state for each fact to Superseded. The first-level query selects all Fact IDs that have an Active Rank greater than one from a deeper query. The innermost query returns Fact IDs with Active Ranks,

Table 12.1 Fact Superseding Metadata Columns

<i>Definition Column</i>	<i>Purpose/Definition</i>
SUPERSEDING_ORIENTATION	Subject, interaction, or organization
SUPERSEDING_DIMENSION	The warehouse dimension, within the specified orientation, against which superseding logic will identify matching facts
SUPERSEDING_ROLE	The warehouse bridge role, within the specified dimension, against which superseding logic will identify matching facts
SUPERSEDING_RANK	The warehouse bridge rank, within the specified bridge role, against which superseding logic will identify matching facts
SUPERSEDING_PERIOD	Ever, year, quarter, month, or day

where the rankings are derived by partitioning active facts over the relevant superseding control data in the metadata. Since the Active Rank partitioning is ordered by descending rank, the Active Rank of one is always the most recent fact that should remain Active, which is why the higher-level query filters out rows with an Active Rank of one. Because the superseding control data in the Metadata can vary considerably, the innermost embedded superseding logic must be implemented as a collection of queries, each working with different values for the control variables. As your requirements for superseding facts change over time, you'll need to add embedded queries to the list.

An example embedded query that identifies facts so there is only ever one active fact for a Subject dimension entry is as follows:

```

SELECT FACT_ID,
       ROW_NUMBER() OVER(PARTITION BY FACT SUBJECT_GROUP_ID, FACT.METADATA_MASTER_ID
                          ORDER BY FACT.SOURCE_TIMESTAMP DESC) AS ACTIVE_RANK
  FROM FACT
 INNER JOIN METADATA_D META
    ON FACT.METADATA_MASTER_ID = META.MASTER_ID
   AND META.SUPESEDING_ORIENTATION      = 'Subject'
   AND META.SUPESEDING_DIMENSION        IS NULL
   AND META.SUPESEDING_ROLE            IS NULL
   AND META.SUPESEDING_RANK           IS NULL
   AND META.SUPERSEDING_PERIOD        = 'Ever'
 INNER JOIN DATA_STATE_D DS
    ON DS.MASTER_ID = FACT.DATA_STATE_MASTER_ID
   AND DS.ABBREVIATION = 'Active'

```

This embedded query shares four characteristics with all other superseding embedded queries:

1. Only looks at Active facts
2. Always sorts the Source Timestamp in descending order
3. Always joins the Fact and Metadata tables to obtain the superseding control data
4. Always includes the Metadata Master ID among the partitioning columns

Three characteristics that *differ* across the various embedded queries are as follows:

1. Filtering of the superseding control data in the Metadata definition table
2. Inclusion of other data among the partitioning columns
3. Joining of additional dimensions to the fact in order to obtain needed partitioning columns

In the example earlier, the metadata parameters include the Orientation (e.g., “Subject”) and Period (e.g., “Ever”), and the Subject Group ID is included in the partitioning data. This results in a ranked ordering of facts for each Subject-Metadata combination for which the metadata include those control parameters.

Examples of facts where a subject should only ever have one active value might include a birth or death fact, or a residential address. Older versions are retained as superseded facts for users who are interested in history, but a basic query of active facts would only ever return a single value per subject for those metadata entries.

The Superseding of older Active facts is the last stage of this ETL Fact pipe. It finalizes the use of data in the Reference and Definition pipes and brings the main processing of the ETL subsystem to its conclusion. Most of the data risks that need to be managed by the ETL subsystem in the Beta version have been handled by these standard pipes. Chapter 13 will provide for some final cleanup and control as well as a validation strategy. It's not uncommon to allow some early-adopting users to begin querying the warehouse at this point. I suggest that those users not be allowed self-service access until more controls are provided in the Gamma version, but with some support from the warehouse team, some very valuable usage can begin with what has been provided so far. Finalizing the Beta version will help ensure that those early queries are as successful as possible pending the finalizing of all data controls in the Gamma version.

Chapter 13

Finalizing Beta

With the processing of Reference, Definition, and Fact data completed, you can turn your attention to finalizing the Beta version. The data remaining to be loaded at this point can be described as internal control data for the warehouse, including the Control Layers and Control Points entries in the Datafeed dimension initialized in Chapter 9, and any audit trail data that you might choose to add to the Fact table in support of updates accomplished in the last three chapters. I also recommend implementing an ETL workflow that validates all of the data that have been loaded, not only as a verification and validation technique within the warehouse implementation project but also as a permanent run-time verification and validation of the loads occurring into the future.

Audit Trail Facts

Keeping an audit trail of updates to a database is typically considered an information technology function of little interest to owners and users of the data. The database management system on which you implement your warehouse likely keeps one or more log files of transactions that would be used by your support staff if you experienced problems with your implementation at the technical data level. The purpose of loading audit trail facts into your Fact table is to support user query capabilities for the data, over and above those technical issues. There are circumstances where your users will benefit from, or be interested in, being able to query a history of what has taken place in the warehouse updates over time, particularly with respect to information that would be completely lost without some form of audit trail.

There are three situations in which information is lost within the ETL logic described in the last three chapters:

1. Broken keys in the Reference pipe that resulted in one or more facts being pointed to the Unknown row in a dimension because of a missing required Context Key
2. Prior values of dimension attributes that were updated in the Definition pipe without the use of slowly-changing dimension logic
3. New values of dimension attributes in the Definition pipe that were incompatible with the data type of the target column in the Definition table

The second situation can be avoided by using slowly-changing value logic on more data attributes, but this could impact storage and performance of the dimension. Highly volatile data could explode the size of a Definition table very quickly. The third situation can be avoided by using character definitions for more columns, but this would sacrifice some query performance and limit capability to use standard filtering conditions, such as data-type-specific logic (e.g., date calculations). Even if reduced in these ways, all three situations are likely to remain, even at smaller occurrence rates, making an audit trail advisable for capturing these data values and making them available to users.

A warehouse audit trail involves inserting one new fact for each condition to be audited. I usually do not include audit trail capability in my Beta version of the warehouse, but I'm describing it in this chapter because it represents a possible control over the situations in which data coming through the ETL subsystem don't make it into the warehouse. Your governance group needs to decide how serious each situation should be considered to be. If they have concerns over the completeness of the warehouse data, including the audit trail in the Beta version can help solidify their confidence in the new warehouse. If their confidence is already high, they might prefer to defer audit trail capability to the Gamma version. The data needed to insert audit facts are already in the ETL staging tables from the Reference and Definition pipes. Allow them to decide whether they want an audit trail and which subset of data should be included.

Each distinct entry in the Reference pipe with the Unknown Flag set to *True* will be inserted as an audit fact, with the value of the Reference Composite as the fact value. Alternatively, an audit fact can be inserted for each Context Key associated with the unknown value. That audit fact will dimensionalize to the target dimension against the Unknown system row, the Calendar dimension using the ETL run date, the Datafeed dimension against the ETL Job ID, and the Metadata dimension against the reference data for Context Key 01. In the Gamma version, you should implement hierachic capability for your metadata, as well as metadata definitions for sourcing Reference Composite values. When you do, you should switch these audit facts to use that Metadata row instead of the one for Context Key 01.

The audit trail for definition attributes involves using the deep Definition pipe data joined to the associated Reference pipe data to insert one audit trail fact for each value that was inserted or updated in the Definition tables. Alternatively, you can add an Audit Flag to the Definition table of the Metadata dimension if your governance group decides to limit the audit trail to a subset of the definition columns. Each audit fact should be dimensionalized to the associated definition row in the target dimension, the Metadata entry under which the attribute value was received, the Calendar dimension using the Source Timestamp, the Datafeed dimension using the Datafeed Master ID found in the entry's Reference pipe data, and the Unit of Measure (UOM) dimension using the implicit UOM entry in the metadata.

An implemented audit trail can be a powerful component of your warehouse implementation. You should resist the temptation to shift these data into a separate Fact table for audit data. The rationale for a separate fact table is usually based on the notion that users aren't interested in auditing and that only support people will look at the data. There's a tendency not to want to "clutter up" the main fact table with audit data. If that's the case, I suggest not implementing the audit trail at all because if it isn't integrated into all of the other data in your warehouse, your users will never look at it. If it's worth doing, look for ways that an audit trail can add value for your users by integrating it into their semantic layer for querying.

The most obvious way that the audit trail adds user value is by providing visibility into data values that would have been discarded by the ETL without the audit. Being able to query all of the broken keys that have been received against a context can help users improve data quality by identifying valid key structures that weren't recognized or identified during data source design or systemic errors in source systems that cause those structures to sometimes supply invalid data. Likewise, being able to query invalid attribute values allows data sourcing problems to be identified and targeted by corrective action. Note that the most common attribute value problem is a source value that should have been a valid date, but wasn't. The next most common is textual data being received for a numeric definition column. Being able to see and analyze these data problems supports a data quality program for continuous improvement and increases overall user confidence in the warehouse.

The auditing of routine changed values in definition attributes adds value in different ways depending on the scope of your slowly-changing dimension strategy. In essence, an audit trail makes the slowly-changing data capability in your Definition pipe less necessary. For low volatility situations where users are likely to use the "as of" capability of the dimension in queries, using the slowly-changing dimension capability makes perfect sense, but for situations where some of the data in the dimension are more volatile or where users aren't expected to need older versions of data, an audit trail can provide the historical data without the need to modify the Definition table. This can have a significant impact on your very large dimensions, such as Subject or Interaction. When a

dimension involves tens of millions of entries, even a single slowly-changing column can explode the data to hundreds of millions of rows very quickly. For subject attributes that inevitably change over time but are of little interest in clinical queries or interaction attributes that change constantly during a patient encounter, slowly-changing attributes can impact the performance of a dimension without substantially improving query usefulness. Queries of these data typically involve only current values even when historical values are present in the dimension. Shifting those data into the audit trail so it is available when needed without impacting the dimension provides an effective alternative.

Since the slowly-changing logic in the Definition pipe stores the same information as the audit trail, one option is only to create audit entries for attribute values that are not controlled as slowly-changing. These are the only attributes for which data are actually lost if they are updated without an audit trail in place. If, in your support role for your production warehouse, you decide in the future to start using slowly-changing data controls for an attribute for which you didn't in the past, the audit trail can be used to retrospectively construct the slowly changed variants in the dimension as an initialization. If you decide to stop using slowly-changing controls for an attribute, the slowly changed variants in the dimension can be used to retrospectively construct the necessary audit trail entries before they are collapsed to a single row in the dimension. In this manner, all data values are available, toggling between values in the Definition table as slowly changed variants or as audit entries in the Fact table. Being able to change your mind later without losing data increases the power and value of the audit trail.

Datafeed Dimension

While the audit trail provides a capability for querying data values that might otherwise not be available, the Datafeed dimension provides a capability to query values according to the characteristics of the sourcing and ETL process that loaded them. Like the audit trail data, it is important to think of data feed information as something that users will want to query and use. If not, the Datafeed dimension will seem very technical—like an information technology artifact—and be of interest only to support staff. That support usage can be very important, but it isn't the primary purpose of the dimension. Users who are interested in characteristics of the data sources should be able to integrate that perspective into their standard queries (e.g., “Are lab results that report contaminated sample errors more or less common from outsourced labs versus our own in-house labs?”). The Datafeed dimension provides that extra source-oriented perspective of the data.

Loading the Datafeed dimension at this point is very straightforward. The staging tables for the dimension were initialized during early ETL processing (in Chapter 9) because the Master IDs for the data were needed throughout the

processing in the Reference, Definition, and Fact pipes. As a result, the data now need only be inserted into the dimension from the Data Control pipe to finalize those data connections. Each Datafeed entry involves inserting both a Reference entry and a Definition entry using data in the Data Control staging table. The first entry created should be for the ETL job entry using the ETL Job ID still contained in the New Surrogates table. There aren't any data that point to that entry, but they serve as the natural hierarchy anchor for all of the source, layer, and point data processed by the job.

The SDI source context entries can be inserted from the data control layers staging table using all the rows that include a NULL value for the Target dimension. All of the other entries in that table can be inserted into the ETL Layer context. Collectively, these entries resolve all of the foreign key values in Definition and Fact rows that have been loaded during the job.

The Data Control Points staging table includes any count values that you added anywhere in your ETL processing, but because those values aren't used in the processing itself, Master IDs have not yet been assigned. In order to insert them into the Datafeed dimension, they must be passed through the New Surrogates table to obtain new Master IDs. Rows where the control count is zero (only possible if one or more of your count queries included an outer join to the Data Control pipe) can be omitted from the load. Lastly, the ETL Job ID entry in the New Surrogates table can be deleted to signal that all warehouse updates are now complete. The absence of the entry in the table will allow the next ETL job to execute when submitted. Your ETL updates are now complete.

Verification and Validation

The data warehouse is a complex system. In order for your organizations to trust and use the system, you need to be able to test it completely and effectively. It's not enough that the warehouse be correct; it must be *provably* correct. This won't be easy. As a software system, the warehouse can be subject to a host of software verification and validation techniques that have matured in the software industry in recent decades, but that only starts to solve the problem. The bigger challenge is one of organizational psychology. Users expect the warehouse to produce exactly the same results they achieve when they look at their data in the original source systems. The problem is that the data in the source systems in healthcare is often very messy, or even wrong. Since you don't want those errors reproduced in the warehouse, your mappings must provide an ability to trace all transformations back to the original source data, as well as transparency into the rules that might have altered the data. The Gamma version will introduce controls and governance in this area, so for the Beta version, you should just concentrate on providing *transparency* and *traceability* as a means to verify and validate your implementation.

Structural Verification

The main level of verification that I recommend for locking down the Beta version is based on the defined structure of the warehouse and its dimensions. Most of the errors I look for are very rare, but will have severe consequences in the generic ETL pipes if left uncorrected. To find these errors, I write and execute queries that specifically look for structural problems, and I return messages in those queries when I find them. I define each message to include a message ID, message text, and message category. The values are arbitrary and allow flexibility in how refined a test strategy to build into the ETL environment.

How the messages are stored or handled varies by implementation based on the range of database and ETL tools that you are using and how you've integrated them into your job scheduling functionality. Generally, I try to integrate them into the operating environment so the job scheduler sees a failure if one or more messages are returned. I usually condition these job failures on the severity of the message (which I usually indicate with the last character of the message ID), or the category of the message. Not all messages indicate a condition that might cause a failure during ETL processing, so the results of these queries should be designed to meet local needs. Think of these queries as a quality control step that can be executed as both the first and last step of your ETL environment. The first execution makes sure that nothing has been broken in the warehouse during the day by the support team running any kind of manual updates, and the last execution makes sure that no structural errors occurred during the ETL execution. Within that model, you can be quite creative in the conditions for which you choose to write queries.

Context Tables

The dimension Context tables are small, but critically important. The main structural failure you need to check for is the absence of the context for the system rows in each dimension:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       'System Context (0) missing from Diagnosis dimension' AS MESSAGE_TEXT,
       'Configuration' AS MESSAGE_TYPE
  FROM (
    SELECT MIN(CONTEXT_ID) AS MIN_CONTEXT_ID
      FROM DIAGNOSIS_C
     )
 WHERE MIN_CONTEXT_ID <> 0;
```

This query must be repeated for each dimension individually, and the errors are extremely unlikely, but since much of the ETL processing is dependent on the presence of the six system rows, it's important to be confident of their structural integrity.

The other potential structural error that would affect users at query time includes mismatches between the required Reference Label entries and the associated optional Display Label entries. If a display label is populated without an associated reference label, the context definition is incorrect:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('Unexpected value of ', 
              DISPLAY_LABEL_01,
              ' in Display Label 01 in Diagnosis Context', CONTEXT_NAME)
                           AS MESSAGE_TEXT,
              'Configuration' AS MESSAGE_TYPE
FROM DIAGNOSIS_C
WHERE DISPLAY_LABEL_01 IS NOT NULL
  AND REFERENCE_LABEL_01 IS NULL;
```

This logic must be repeated for each of the 10 labels in each dimension. The reverse situation of a reference label being populated without an associated display label is not an error, but should be corrected by updating the display label with the value in the reference label:

```
UPDATE DIAGNOSIS_C
  SET DISPLAY_LABEL_01 = REFERENCE_LABEL_01
FROM DIAGNOSIS_C
WHERE DISPLAY_LABEL_01 IS NULL
  AND REFERENCE_LABEL_01 IS NOT NULL;
```

Alternatively, a database trigger could be defined that returned the reference label if the associated display label were queried and found to be NULL.

Reference Tables

The Reference tables require high integrity because they drive much of the processing logic of the ETL pipes. One problematic area is duplicate entries that would create hidden Cartesian joins during queries. They can be avoided by searching for any Reference Composite that occurs more than once in the same Context ID:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('Diagnosis dimension includes ',
              COUNT(*),
              ' non-unique Reference Composites')
                           AS MESSAGE_TEXT,
              'ETL Load' AS MESSAGE_TYPE
FROM (
  SELECT CONTEXT_ID, REFERENCE_COMPOSITE, COUNT(*)
    FROM DIAGNOSIS_R
   GROUP BY CONTEXT_ID, REFERENCE_COMPOSITE HAVING COUNT(*) > 1
)
```

Situations in which this condition is found should not occur and are extremely unlikely to occur once you've moved your Beta ETL code past unit testing. One option that I do not recommend to avoid this problem is forcing these values to be unique at the database level. While that control would prevent the error this code is looking for, it would be at the cost of having an ETL job (in this case, the Reference pipe) fail on a database constraint. A failure in these generic jobs prevents all of the data for the day from being loaded. I allow the data to load and use the query to spot the problem so I can correct it. I might have to change or delete (for reprocessing) the data associated with the duplicate key, but the rest of the ETL loads will complete successfully. Considering how rare I expect this error to be, I prefer to carry that burden on the support side. In 10 years of building data warehouses in healthcare, I've not yet seen this error occur in production.

A more likely Reference error, and one having its root cause in the design of the metadata for a new source load, is a mismatch between the count of components expected in a reference composite versus count of components actually sourced. If the metadata defines a source key as having multiple parts, it's important that the actual keys sourced always have the same number of parts. This can be tested by looking at the presence of reference labels in the Context table versus Context Keys in the Reference table. First, you can identify unexpected Context Keys by looking for values that have been sourced even though the associated Reference Label is NULL:

```

SELECT 'ERR0001E' AS MESSAGE_ID,
       CONCAT('Diagnosis context ',CONTEXT_NAME,
              ' has ', COUNT(*),
              ' reference entries with unexpected Context Keys')
                  AS MESSAGE_TEXT,
       'Configuration' AS MESSAGE_TYPE
  FROM DIAGNOSIS_C C
 INNER JOIN DIAGNOSIS_R R ON (R.CONTEXT_ID = C.CONTEXT_ID)
 WHERE (REFERENCE_LABEL_01 IS NULL AND CONTEXT_KEY_01 IS NOT NULL)
       OR (REFERENCE_LABEL_02 IS NULL AND CONTEXT_KEY_02 IS NOT NULL)
       OR (REFERENCE_LABEL_03 IS NULL AND CONTEXT_KEY_03 IS NOT NULL)
       OR (REFERENCE_LABEL_04 IS NULL AND CONTEXT_KEY_04 IS NOT NULL)
       OR (REFERENCE_LABEL_05 IS NULL AND CONTEXT_KEY_05 IS NOT NULL)
       OR (REFERENCE_LABEL_06 IS NULL AND CONTEXT_KEY_06 IS NOT NULL)
       OR (REFERENCE_LABEL_07 IS NULL AND CONTEXT_KEY_07 IS NOT NULL)
       OR (REFERENCE_LABEL_08 IS NULL AND CONTEXT_KEY_08 IS NOT NULL)
       OR (REFERENCE_LABEL_09 IS NULL AND CONTEXT_KEY_09 IS NOT NULL)
       OR (REFERENCE_LABEL_10 IS NULL AND CONTEXT_KEY_10 IS NOT NULL)
 GROUP BY C.CONTEXT_NAME;

```

Second, you can look for missing source components by looking for NULL Context Keys where the Reference Label was populated:

```
SELECT 'ERR0001E' AS MESSAGE_ID,
       CONCAT('Diagnosis context ', CONTEXT_NAME,
              ' has ', COUNT(*),
              ' reference entries with missing Context Keys')
                  AS MESSAGE_TEXT,
       'Configuration' AS MESSAGE_TYPE
  FROM DIAGNOSIS_C C
 INNER JOIN DIAGNOSIS_R R ON (R.CONTEXT_ID = C.CONTEXT_ID)
 WHERE (REFERENCE_LABEL_01 IS NOT NULL AND CONTEXT_KEY_01 IS NULL)
    OR (REFERENCE_LABEL_02 IS NOT NULL AND CONTEXT_KEY_02 IS NULL)
    OR (REFERENCE_LABEL_03 IS NOT NULL AND CONTEXT_KEY_03 IS NULL)
    OR (REFERENCE_LABEL_04 IS NOT NULL AND CONTEXT_KEY_04 IS NULL)
    OR (REFERENCE_LABEL_05 IS NOT NULL AND CONTEXT_KEY_05 IS NULL)
    OR (REFERENCE_LABEL_06 IS NOT NULL AND CONTEXT_KEY_06 IS NULL)
    OR (REFERENCE_LABEL_07 IS NOT NULL AND CONTEXT_KEY_07 IS NULL)
    OR (REFERENCE_LABEL_08 IS NOT NULL AND CONTEXT_KEY_08 IS NULL)
    OR (REFERENCE_LABEL_09 IS NOT NULL AND CONTEXT_KEY_09 IS NULL)
    OR (REFERENCE_LABEL_10 IS NOT NULL AND CONTEXT_KEY_10 IS NULL)
 GROUP BY C.CONTEXT_NAME;
```

The seriousness of these particular errors needs to be evaluated on a case-by-case basis. These errors typically turn out to have been sourced correctly, with the configuration data being incorrect. This commonly occurs when a multipart key from a source system requires different variations that involve fewer key components. The solution is to add the new context entries, update the “erroneous” Reference entries to point to the correct new context, and reevaluate the metadata for that source load to see if the newly evaluated data better represent what is expected from that source.

Definition Tables

The structural integrity of the Definition tables depends on the presence of the required six rows of the System subdimension. The simplest test is to check that there are exactly six rows in that subdimension:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('System subdimension of Diagnosis includes ',
              COUNT(*), ' rows (expected 6 rows)')
                  AS MESSAGE_TEXT,
       'Configuration' AS MESSAGE_TYPE
  FROM DIAGNOSIS_D
 WHERE SUBDIMENSION = 'System'
 GROUP BY SUBDIMENSION HAVING COUNT(*) <> 6
```

Alternatively, you can check each specific row individually to assure that each is present in the table:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       'Not Applicable (1) row is missing from System subdimension of Diagnosis' AS MESSAGE_TEXT,
       'Configuration' AS MESSAGE_TYPE
  FROM (
    SELECT R.REFERENCE_COMPOSITE
      FROM DIAGNOSIS_C C
      LEFT JOIN DIAGNOSIS_R R ON (C.CONTEXT_ID = R.CONTEXT_ID)
     WHERE C.CONTEXT_ID = 0
       AND R.MASTER_ID = 1
  ) D
 WHERE D.REFERENCE_COMPOSITE IS NULL
```

This query needs to be repeated for all of the six system rows, and the set of six queries must be repeated for each dimension. Errors are extremely rare, but must be corrected if found.

Bridge Tables

An integrity issue to be tested in the Bridge tables is the balancing of the weight values across the multiple entries in a group. If they don't sum to one, any aggregation carried out across the group will produce erroneous results in a query:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('Diagnosis dimension includes ', COUNT(*), ' groups with unbalanced Weight values (not summed to 1.0)') AS MESSAGE_TEXT,
       'ETL Load' AS MESSAGE_TYPE
  FROM (
    SELECT GROUP_ID, SUM(WEIGHT)
      FROM DIAGNOSIS_B
     GROUP BY GROUP_ID HAVING SUM(WEIGHT) <> 1
  )
```

The values are obtained in the sourcing metadata in the Beta version, so errors will be rare and easily corrected since those settings are within the warehouse team's direct control. However, the Gamma version will introduce direct sourcing of weight values, making an error of this type more likely. While easily corrected, users of the data are extremely unlikely to be able to spot this error in their queries, making a structural test like this very important.

Group Tables

Users don't directly access the Group tables in their queries, but the structural integrity of the data, and the way it maps to the corresponding Bridge table entries, is critical to the correct mapping of data during both ETL and querying. Although an error in this table can occur only through a logic error in the Fact pipe while building bridges and groups, the relationship is so important

that I recommend a test to assure that the data in the Group table matches the data in the Bridge table, particularly with respect to delimiters in the Group Composite:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('Diagnosis group ', G.GROUP_ID, ' includes ',
              1 + LEN(DR.REFERENCE_COMPOSITE)
              -LEN(REPLACE(DR.REFERENCE_COMPOSITE,NATURAL_HIERARCHY_INTERNAL_DELIMITER,''))
              ' components in composite, but ', COUNT(*) , ' bridge entries.')
              AS MESSAGE_TEXT,
       'ETL Load' AS MESSAGE_TYPE
  FROM DIAGNOSIS_G G
 INNER JOIN DIAGNOSIS_B B ON (B.GROUP_ID = G.GROUP_ID)
 GROUP BY G.GROUP_ID HAVING COUNT(*) <> 1 + LEN(DR.REFERENCE_COMPOSITE)
              -LEN(REPLACE(DR.REFERENCE_COMPOSITE,NATURAL_HIERARCHY_INTERNAL_DELIMITER,''))
```

This query identifies any group where the number of delimiters is incorrect. The correct number of delimiters should always be one less than the number of bridge entries defined for the group. When a mismatch occurs, the damage can be significant. The source of the error is usually incorrect definition of metadata, most likely a redundant use of the same role and rank within a single fact definition. Since some of the deep data in the Fact pipe goes through reduction using a DISTINCT clause, it's possible that the data used to build the Group Composite value could end up being different than the data used to insert the Bridge entries. Good testing during development virtually prevents this situation, but the risk warrants regular testing.

Fact Table

The main risk in the Fact table is that one or more of the foreign keys to the dimensions will be NULL, causing the data to be dropped in queries that depend on inner joins. The test looks for any facts that contain NULL entries:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('Fact table contains ',
              COUNT(*),
              ' Null references to Diagnosis dimension')
              AS MESSAGE_TEXT,
       'ETL Load' AS MESSAGE_TYPE
  FROM FACT
 WHERE DIAGNOSIS_GROUP_ID IS NULL
```

This query needs to be repeated for each dimension. Alternatively, you can set the NULL values to point to the Not Applicable entry in the System subdimension:

```
UPDATE FACT
   SET DIAGNOSIS_GROUP_ID = 1
  FROM FACT
 WHERE DIAGNOSIS_GROUP_ID IS NULL;
```

I recommend not defaulting NULL keys to a value until you have completely diagnosed why the NULL key occurs in the first place. A NULL key indicates a breakdown of your finalization logic in the ETL Fact Pipe and that error should be corrected before reinitializing these NULL values.

Metadata

Most of the errors you'll ever see in your warehouse will be traceable back to something being incomplete or wrong in your metadata. It's the metadata that provides the level of abstraction that enables your generic ETL subsystem to function. That's a great strength of this design, but also its main vulnerability if defects leak into the data. You should invest time making your metadata as mistake-proof as possible.

To identify conditions for testing, it helps to imagine conditions that could be predictably wrong with the metadata and then test for those conditions. An example is when the metadata for a new source dataset involves data identified by keys that haven't previously been seen in the warehouse. It's possible that an analyst might define the metadata for loading this new source key, but forget to add the necessary context to the dimension's Context table:

```
SELECT 'ERR0001D' AS MESSAGE_ID,
       CONCAT('Target context ', E.TARGET_CONTEXT,
              ' in metadata (', E.MASTER_ID,
              ') not defined in DIAGNOSIS_C') AS MESSAGE_TEXT,
       'Metadata' AS MESSAGE_TYPE
  FROM (
    SELECT D.MASTER_ID, D.TARGET_CONTEXT
      FROM METADATA_D D
     LEFT JOIN DIAGNOSIS_C C ON (C.CONTEXT_NAME = D.TARGET_CONTEXT)
    WHERE D.TARGET_TYPE = 'Reference' ) E
 WHERE C.CONTEXT_NAME IS NULL;
```

This error doesn't show up in the metadata itself, but does cause the subsequent ETL Reference pipe to be unable to resolve the keys during processing. Because ETL processing is usually based on inner join conditions, the data for these keys does not load, and definition and fact data associated with those keys are dropped.

Fortunately, errors in metadata tend to show up during unit testing of new source loads, so users are rarely impacted. The more you mistake proof your metadata, the less likely these operational conditions are to occur. Over the years, several of my clients have implemented actual metadata maintenance applications (usually web based) that allow analysts to add or maintain metadata in the warehouse. All the test conditions and constraints become embedded in the metadata application. I'm not against this approach, but it has to be done carefully, and it can't catch all of the problems because some of the problems

emerge from combinations of data conditions that might not be visible at the maintenance transaction level against a single row of metadata. The previous context example would be caught by this approach. The application forbids a metadata row from referencing a dimension context if that context isn't found on the Context table.

Whether written as queries to be run during operation or as conditions built into a smarter application, the mistake-proofing of your metadata should receive high priority as you wrap up your Beta implementation.

Preparing for Gamma

Although the Beta version doesn't include certain advanced functions and features that you'll implement in the Gamma version, and it lacks many of the security, privacy, and performance controls that you've also deferred to the Gamma version, it is still a fairly full-functioned data warehouse. The vast majority of queries that somebody would want to write, particularly those that correspond to historical reporting against your data sources, can already be supported. Because of the lack of controls, you'll want to keep your support staff integrally involved in any early use of the Beta version by your user community. Within those constraints, many of your users will be very satisfied when given access to this Beta version while you move on to implement the Gamma version.

GAMMA VERSION



The Gamma version of the warehouse builds on the Beta version, adding functions and features for performance and security. The Gamma version is a functionally complete system, typically still lacking much of the data that need to be loaded in order to consider it production ready. A central challenge is to make the system operational, so little-to-no human intervention is required in its day-to-day operation. This requires that privacy controls be finalized so users can freely access the warehouse once it is placed into production. The Gamma version is typically ready 3–4 months after the Beta version is completed, although the systems factors interacting with the overall environment introduce more variability into the plan as more factors emerge as beyond the direct control of the development team.

Chapter 14

Finalizing ETL Workflows

The Gamma version is the final iteration of your development spiral. The Beta version didn't include all of the functionality that you will ultimately want in your warehouse. The effective boundary between functions included in Beta and those delayed until Gamma was variable, and different organizations will make different choices about when to deploy individual features. The key differentiator was keeping the Beta version small enough to be implemented in only a few months, and that wouldn't have been possible if you tried to include everything. The functions and features included in this chapter are the ones that my own experience has shown can be delayed until later in your project. They typically involve adding more detailed features to the core functions developed in Beta, features that were omitted earlier to enable meeting essential Beta version goals. Those features and functions have to be added now because as you move toward production you'll lose the ability to reinitialize the warehouse and reload data as you add functions and features, and many of these features would require conversion of data already in the warehouse if implemented later.

Most of the changes that are required to finalize the ETL workflows will involve dropping or altering some of the simplifying assumptions you made previously to keep the scope of the Beta version more manageable. These include the following:

- *Alternatively sourced keys:* Sometimes, the data being sourced into the warehouse will have been sourced from the warehouse itself rather than from some organizational system or external dataset. In these cases, the notion of a natural key existing in the data source breaks down. The Gamma version needs to be able to allow you to source the data using the various surrogates used to store the data within the warehouse. This capability will be critical when you start producing and storing metrics where you've used data in the warehouse to derive values for those metrics.
- *Sourced metadata:* Sometimes, the parameters that need to process the source data won't be known in advance, so they won't be available to be

placed in the Metadata dimension during configuration. Those values might actually be part of the source data itself and will need to be sourced just like any other data, except that the source data will control the flow of your ETL in ways that couldn't have happened in the earlier versions.

- *Standard data editing:* Knowing that healthcare data can be error prone or incomplete, you'll add value to your ETL loads by using the metadata associated with each source-target data mapping to evaluate inbound data against the requirements, indicating to users any problems or issues found.
- *Value-level unit of measure (UOM):* Some queries will need to filter the actual value of facts in the Fact table, which is the least efficient place to access data in the warehouse. For some kind of facts (e.g., facts with a small range of values or the most commonly used values for widely varying facts), you will dimensionalize the actual values into the UOM dimension in order to promote ease of use and the greater efficiencies of using a dimension over the Fact tables, even when the fact value itself is the subject of your query or filter.
- *Undetermined dimensionality:* Data in the Beta version that was dimensionalized to the Undetermined row in a System subdimension need to be redimensionalized to a legitimate value in the dimension.
- *ETL transactions:* The range of activity supported in the Beta version ETL is very narrow: Insert new data, update dimensions if present. In the Gamma version, you'll expand to include more standard transaction types for inserting, updating, initializing, or even deleting data. You'll also include an ability to define organizationally specific transaction types as a form of "user exit" in the ETL process.
- *Target states:* In the Beta version, data are always added to the Active data state. You will remove that limitation from the Gamma version to allow the data to be loaded into any desired state, including Deleted, Merged, Inactive, Corrected, Quarantined or Superseded. You'll also allow for organizationally-specific additional states to be defined.
- *Superseded facts:* The Beta version prevented duplicate facts from being loaded in a Fact table, but it didn't check for multiple variations or versions of the same facts. In Gamma, you'll add the capability to allow a newly arriving fact to trigger a change in state of earlier facts from Active to Superseded making it much easier for users to query only active facts.

These capabilities can actually be added to the ETL at any time. I've seen aspects of these functions included earlier in the Beta version because they were deemed important for some initial loading. I've also seen some of these features deferred for years because they weren't needed in the near term, or the available work-arounds were considered sufficient. Every organization I've worked with has made different choices at this stage. Because your ETL is very generic, and each function is typically isolated to very specific places in the code and jobs, it doesn't require a major effort to add the functions later. In cases where

traditional nongeneric ETL is being developed, the choice of timing can be more critical. Since the traditional ETL architecture might involve hundreds or thousands of distinct ETL jobs, each containing the logic required for that particular ETL load, the choice of including or deferring particular features becomes a critical success factor since new features that require altering a thousand ETL jobs typically don't ever get implemented. In your case, these changes can be deferred indefinitely, if necessary, without too much disruption. Implement each as needed, although most of my clients have seen benefits in implementing these features earlier rather than later.

Alternatively Sourced Keys

A central theme of the generic architecture of the data warehouse is the functional separation of the source system natural keys that identify data from the surrogate key identifiers generated within the warehouse for identifying that data. This separation allows the same data arriving from multiple source systems using different natural keys to be ultimately identified and treated as the same data, correctly using any arriving natural key in a source dataset to map the correct surrogate identifier. Additionally, the design allows for a generic structure for those natural keys to have up to 10 components differently sourced in a way that the number of components isn't a parameter that needs to be known to the generic ETL flows that are processing those multipart natural keys. In the Beta version, the ETL code processes a 10-part natural key in exactly the same way as a 1-part key.

As you build out the ETL subsystem in the Gamma version of the warehouse, you need to allow two alternatives to the general Beta processing in order to support some requirements that you weren't concerned with in the Beta version: (1) treating multipart source keys as though they are one-part source keys in order to improve ETL performance and (2) allowing the source key to be one of the warehouse surrogate values when the data being loaded in the ETL originated in the warehouse itself. The implementations of these two alternatives are largely the same, but their rationales are different; so each alternative will be explored separately.

Sourcing Compound Natural Keys

The first alternative allows the developer of a source data intake (SDI) query to take advantage of the knowledge about the source data that are available at the time of the creation of the SDI job: knowledge that combinations of source columns being extracted will be combined to create compound natural keys. Developers can now do something they wanted to do back in the Beta version that I stopped them from doing. It's not that I can't see some value in this change, it's that I disagreed with the reasons why developers often want to do it.

Under what circumstances would I agree that sourcing the data as a single compound natural key is a good idea? It is a good idea only when the volume of the data being processed by the ETL subsystem grows to the point where performance degradation starts to be observed or expected. The volume of the data being processed in the ETL workflows is a function of both the number of rows being processed in the source and the number of columns being sourced from each row. When the source includes several multipart natural keys, it's typical for a few columns to be used as high-order components of multiple keys, exaggerating their impact on resources used by the load. These conditions are most often met when loading historical data into the Gamma version, and since historical loads are often one-shot implementations, the risks introduced by consolidating the compound key in the SDI job can be minimized.

I recall a situation where I wanted to load several years of historical laboratory results into the Gamma version of a warehouse. My source dataset included over 700 million results, going back 10 years. There were many natural keys within these data, but three in particular bothered me: the natural identifier for the laboratory test ordered involved the concatenation of three different source columns, the identifier for the lab test performed (or resulted) involved four source columns (two of which matched the first two in the ordered procedure), and the patient location involved four source columns. That was seven source columns that would become seven SDI dataset rows (or nine rows after the metadata transformation in the start-up of the generic ETL). I knew those rows were destined to become just three rows in the Reference pipe after the Context Keys were pivoted into the Reference Composite. With six extra wasted rows across 700 million source transactions, I was looking at 4.2 *billion* extra rows in memory during my Gamma load for just this dataset. The ETL Reference pipe would eliminate those extra rows after pivoting all of the component keys into the required Reference Composite values, but a great deal of storage and processing resource would be consumed in the process. The same circumstances were occurring across many of the Gamma loads that we were developing.

Instead of requiring all of those resources, you can allow the SDI query to preconcatenate any desired natural keys into single derived column, and source those columns as any other column might be sourced in the SDI query. It doesn't matter what the name of the derived source column is, as long as it is mapped to the Reference Composite in the metadata. Using this approach, the source dataset contains only one row for each natural key, regardless of how many source data components went into making up that compound key. This approach requires that the developer of the SDI query be knowledgeable regarding the target context against which that key will be loaded, because the end result is that the Reference Composite value must look like it would have looked if the source columns had simply been extracted and mapped to the various Context Key columns in the metadata. The metadata must now contain

an entry for the Reference Composite source column instead of the various Context Key values. If the metadata entry isn't provided, the new sourced Reference Composite value will not be able to move into the Reference pipe when the ETL executes.

Although the alternative of sourcing the Reference Composite directly is always available, there are circumstances where it either doesn't really mitigate the performance risk or it creates new risks. If the components of the natural key still have to be sourced by the SDI query because they are being mapped to columns in the Definition pipe, some of the savings of eliminating the source rows for those individual columns will be lost. Also, the new risk is that the standard codeset translation for source columns can't be used to clean up values in the individual components as the data are transformed against the metadata. This alternative shouldn't be used if either of those two risks apply but is relatively safe otherwise.

Sourcing Warehouse Surrogates

The second alternative for sourcing keys involves allowing the warehouse itself to serve as a source dataset. This can involve sourcing data in a dimension or in one of the Fact tables. The keys that might be needed, depending upon what is being sourced, include the Master ID, Definition ID, or Group ID in any dimension, or a Fact ID in a Fact table. If sourced, these columns can be given any name in the generated SDI dataset as long as the metadata maps these columns to the correct Master ID, Definition ID, Group ID, or Fact ID as target columns.

You didn't need to source warehouse surrogate columns as keys in the Beta version because you didn't have any situations where you were generating new source data by looking into the warehouse. Here in the Gamma version, you'll have several opportunities, including the creation of metric data for dashboards and scorecards within the warehouse and for managing the superseding or cleanup of data values in the quality assurance toolset. These new functions will be discussed later in this chapter. For now, you just need to be aware that when you source data from the warehouse, you'll often want to use the warehouse surrogate keys as source columns. Sourcing surrogates eliminates the need for an SDI query to extract dimension data with joins all the way through the dimension to the Context table, possibly involving multiple bridge entry pathways. That approach could end up sourcing dozens of columns of information, only to have the ETL Reference pipe convert all of that data back into the original surrogate keys during the ETL load. Sourcing the surrogates themselves avoid the need to extract all of that natural key data, define the metadata needed to remap it back into the warehouse, and transform it into the (already known) surrogate keys in the ETL Reference pipe. Sourcing surrogate keys is a simple and efficient alternative, and it doesn't require any changes in the way the SDI query is developed.

Alternatives in the Reference Pipe

The easiest way to incorporate the newly sourced Reference Composite or surrogate keys into the Reference pipe is to add them to the pivot command that turns the deep values for the existing Context Keys into a single wide row from which a Reference Composite is being derived.

```
PIVOT
(
    MIN(VALUE)
    FOR TARGET_COLUMN IN (
        CONTEXT_KEY_01,
        CONTEXT_KEY_02,
        CONTEXT_KEY_03,
        CONTEXT_KEY_04,
        CONTEXT_KEY_05,
        CONTEXT_KEY_06,
        CONTEXT_KEY_07,
        CONTEXT_KEY_08,
        CONTEXT_KEY_09,
        CONTEXT_KEY_10,
        REFERENCE_COMPOSITE,
        MASTER_ID,
        DEFINITION_ID,
        GROUP_IDS
    )
) P
```

This revised query results in the wide Reference pipe rows having the new key values available for processing. Additional logic can be added to the Reference and Fact pipes to use those values as needed during the continuing load. For example, when the Master ID or Definition ID is sourced, the Reference pipe doesn't need to do the normal lookup to identify the appropriate Definition row for a Reference Composite. In the Fact pipe, a lookup to the Group table isn't necessary against data for which the Group ID was supplied. Because these lookups tend to be resource intensive, ETL runs where these keys are sourced can save processing time.

While the expanded query is logically correct, it contains a performance flaw that can be improved upon. The source system keys being received in the staging table will never use all of the columns specified in the pivot command, so there is some inefficiency in including all of them in the same query and pivot. If the source supplies any of the new alternate values, there will be no Context Keys, so a pivot command is actually unnecessary. Based on this distinction, you can split the single query into separate queries, only one of which requires a pivot. The results of the multiple queries will be combined through a UNION clause to provide the correct staged output ([Figure 14.1](#)).

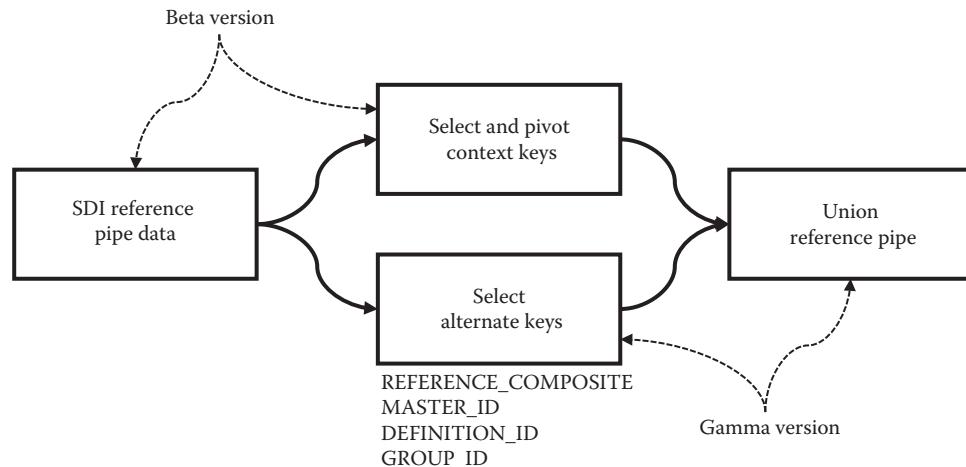


Figure 14.1 Alternate key sourcing pathways, Beta versus Gamma.

The original pivot-based query needs to have a NULL selected for each of the new alternate keys that is not being passed through the PIVOT command. This allows a common output result to be obtained from both pivoted and nonpivoted subqueries. The new additional queries—one for each new alternate key—each select the desired key and force NULL for all of the other keys. The Reference Composite select looks like the following:

```
UNION
SELECT ...
    P2.SOURCE_VALUE AS REFERENCE_COMPOSITE,
    NULL AS MASTER_ID,
    NULL AS DEFINITION_ID,
    NULL AS GROUP_ID
FROM
(
    << Metadata Transformation HERE >>
    AND S.TARGET_COLUMN = 'REFERENCE_COMPOSITE'
) P2
```

The other three alternate keys have the same structure, only differing in which of the four new keys they are selecting.

```
UNION
SELECT ...
    NULL AS REFERENCE_COMPOSITE,
    P3.SOURCE_VALUE AS MASTER_ID,
    NULL AS DEFINITION_ID,
    NULL AS GROUP_ID
FROM
(
    << Metadata Transformation HERE >>
    AND S.TARGET_COLUMN = 'MASTER_ID'
) P3
```

```

UNION
SELECT ...
    NULL AS REFERENCE_COMPOSITE,
    NULL AS MASTER_ID,
    P4.SOURCE_VALUE AS DEFINITION_ID,
    NULL AS GROUP_ID
FROM
(
    << Metadata Transformation HERE >>
    AND S.TARGET_COLUMN = 'DEFINITION_ID'
) P4

UNION
SELECT ...
    NULL AS REFERENCE_COMPOSITE,
    NULL AS MASTER_ID,
    NULL AS DEFINITION_ID,
    P5.SOURCE_VALUE AS GROUP_ID
FROM
(
    << Metadata Transformation HERE >>
    AND S.TARGET_COLUMN = 'GROUP_ID'
) P5

```

This selection of the alternate keys can also be accomplished with a single subquery that includes appropriate CASE statements for the assignment of selected column values.

```

SELECT ...
CASE WHEN PA.TARGET_COLUMN = 'REFERENCE_COMPOSITE'
    THEN PA.SOURCE_VALUE ELSE NULL END REFERENCE_COMPOSITE,
CASE WHEN PA.TARGET_COLUMN = 'MASTER_ID'
    THEN PA.SOURCE_VALUE ELSE NULL END MASTER_ID,
CASE WHEN PA.TARGET_COLUMN = 'DEFINITION_ID'
    THEN PA.SOURCE_VALUE ELSE NULL END DEFINITION_ID,
CASE WHEN PA.TARGET_COLUMN = 'GROUP_ID'
    THEN PA.SOURCE_VALUE ELSE NULL END GROUP_ID
FROM
(
    << Metadata Transformation HERE >>
    AND S.SOURCE_COLUMN IN
        ('REFERENCE_COMPOSITE', 'MASTER_ID', 'DEFINITION_ID', 'GROUP_ID')
) PA

```

At this point, you've completely altered the Reference pipe to allow for the sourcing of the multiple alternate warehouse keys that are needed to source the data into the warehouse from the warehouse itself. Just as in the Beta version, there is now a single wide row in the Reference pipe for each transactional dimension key. The code to take advantage of these additional values still needs to be added. Some of that code will be straightforward, like using the Group ID in the source to dimensionalize a fact rather than going through the bridge and group

building process. Other code will be less straightforward because the reason for the alternate key is to conduct processing that wasn't considered in the Beta version, such as using the Master ID to delete a dimension entry.

Note that this discussion hasn't dealt with the sourcing of the Fact ID as an alternate key. That key isn't used in the Reference pipe, so it wasn't included in the code revisions. The sourcing of Fact ID will be used later in the Fact pipe when you add functions for modifying or superseding facts.

Sourced Metadata

One major weakness in the Beta version implementation is the rigidity with which all of the processing parameters that are needed to load data must be defined in advance in the Metadata dimension. To the extent that a data source might require high variability over those parameters, the metadata that needs to be defined to handle all of the various alternatives is extensive. Worst among these are situations where the role that a dimension bridge entry might play in a fact, or the relationship that might be defined between ancestor and descendant entries in a hierarchy, needs to vary across or within the source dataset. In the Beta version, these situations require explicit metadata to be defined for each possibility and usually involve defining many logical datasets or fact breaks in that metadata to keep all the permutations straight. It works, but it's more cumbersome than it needs to be. Here in Gamma, we fix this problem by allowing values in the metadata to be sourced in the SDI files just like any other data. The multiple iterations of metadata required in the Beta version collapse into fewer metadata rows that expect the variability in operating parameters to be supplied by the source intake files. This makes the resulting metadata more generic and allows more flexibility for sourcing complicated datasets without having to build specific metadata for every logical permutation of data types in those datasets.

The basic logic of this functional expansion is simple: Allow SDI queries to source values for metadata parameters, ensuring that those source columns are mapped to the appropriate transactional subsets of the ETL in which those values will be used. For example, suppose a source for a flat hierarchy includes the relation value that describes the edge between the ancestor and descendent vertices. The flat hierarchy processing in the Beta version already pivots the two Master IDs into a single row when creating the flat hierarchy entries. That pivot now includes additional metadata columns, including relation, so the sourced relation would be available for the flat hierarchy entry. The ETL code is modified to use the sourced value, if present, or the original metadata entry for relation, if the sourced value is not present.

Likewise, an SDI query for a master load might need to treat the target subdimension of an entry as a sourced column if the source dataset contains information that will populate different subdimension rows in a dimension.

In the Beta version, a complete set of metadata would have been defined as a different logical dataset for every possible subdimension, just to allow the needed variability. By adding the subdimension column to the pivot of data in the Reference pipe, the sourced subdimension value would be used when creating new rows in a Definition table, either as a master load or an orphan creation. If a subdimension value were not sourced, the value in the metadata row would be used.

Any of the columns in the metadata Definition table can be sourced. The actual implementation strategy in the ETL is always the same: Find the source data pivot that combines the data related to the parameter being sourced and add the newly sourced column to that pivot. Four main SQL pivot queries are needed in the ETL processing, three already present in the Beta version and one new query.

- *Reference pipe*: The pivot that takes the Context Keys from deep to wide can be modified to allow metadata related to the dimension definition to be aligned with the master data needed to maintain dimension entries.
- *Reference pipe, hierarchy*: The pivot that combines ancestor and descendent references into wider hierarchy transactions can be modified to add sourced metadata columns.
- *Definition pipe*: The pivot that takes dimension definition columns from deep to wide can be modified to include any sourced metadata columns that end up in a Definition table of a dimension.
- *Fact pipe, bridges*: A *new* pivot query will be needed to take the parameters for a single-dimension bridge from deep to wide, allowing the source to provide roles, ranks, and weights. The Beta version didn't need this query because none of the required columns could be sourced.

In the data being selected in these pivot queries, we use a COALESCE or CASE statement to cause the sourced value to be used, if present, instead of any metadata value. In this way, sourced metadata parameters always *override* any stored metadata values, but no particular metadata value is *required* to be sourced.

Standard Data Editing

The generic nature of the ETL process for loading the data warehouse offers opportunities to anticipate problems in the data that can be solved or mitigated by adding some standard editing and transformation logic at certain points in the ETL jobs. The amount of resource you'll invest in these types of functions will depend upon whether or not your source data are found to have patterns of data quality problems that could be addressed in the following ways.

Value Trimming and Cleanup

Source value columns are typically expected to contain valid characters and neither start with leading spaces nor have a trail of ending spaces. Most of the time, this expectation is met; but when it isn't, it can cause significant problems that can be difficult to track down. I recommend trimming leading and trailing spaces from all inbound source columns. I also recommend stripping undesired ASCII values from inbound source values, unless their loss would change the functional meaning of the values involved. Undesired ASCII values are any that don't display when queried or can't be typed by a user writing a filter in a query. These characters make querying and using impacted data very difficult and, since most of them typically originate from data transmission errors, they can be removed without impeding the effective use of the data.

Timestamp Handling

It is common for a fact to be dimensionalized to both Calendar and Clock dimensions using a source data value that was originally a timestamp in the source data. I recommend writing a function that can separate the date and time portion of timestamps, leaving only the desired component in the source value as it continues through the ETL processing. This function can be used in the ETL Reference pipe to cleanse Context Keys destined for the Calendar and Clock dimensions. If you pass the function a timestamp value along with the Target Context from the Calendar or Clock dimension, the function can return the appropriate year, quarter, month, day, meridian, hour, minute, or second for continued processing. By providing for this conversion within the ETL stream, the SDI queries don't have to do the splitting themselves, and the timestamp source column needs only to be sourced once regardless of how many times, and in what fashion, it is used in the processing of an ETL data source.

Data Type Checking

Every fact is eventually dimensionalized to the UOM dimension. If that assignment is explicit through the data source or is implicit through the metadata, there is a chance that the actual source value received from the source will not match the data type implied by the associated UOM for the value. Of particular concern are UOM that are identified as quantitative in the UOM dimension but arrive as nonquantitative values from the data sources. Left unchecked, these nonnumeric values will cause query failures whenever they are included in any algebraic aggregating query.

To overcome this condition, the logic in your Fact pipe can be expanded to check the data type of the fact value against the requirements of the associated UOM. If the UOM entry indicates that the scale of the UOM is quantitative, but the fact value doesn't pass a type check for numeric, the UOM for the fact has to

be modified. This typically entails creating a new group in the UOM dimension, if it hasn't already been created by a previous failure of a data type check for the same UOM entry. The new group includes all of the bridges of the original group, plus a new entry that places the Master ID of the Unexpected Text entry in the UOM dimension (which was created during the initialization of the warehouse), into the Value role with a weight of zero. The original bridge entry that was in the Value role is altered to be Expected (which leaves the weight unaltered). In this manner, the fact has been updated so its UOM is now Unexpected Text, and its expected UOM is whatever the original sourced or implicit UOM was. Since the Unexpected Text unit is defined with a class of Narrative the fact will no longer be included in any algebraic aggregation queries.

For this data control to have the desired user impact, it is critical that a coding convention for checking these results always be included in aggregating queries, most likely in your semantic layer definitions for your business intelligence environment. This requirement will impact any aggregating query against a fact value, such as,

```
SELECT SUM(VALUE) FROM FACT;
```

A query like this will fail if one or more values in the result set aren't a valid number. This query should already be filtering the dimensions enough to only be including values that are expected to be numeric, so nominal textual facts are being excluded. An example could be a query looking to aggregate financial charges against an encounter over a specific period of time. The expectation of the user creating the query would be that all of the charges resulting from that query would be valid. To ensure that is the case, these queries should always include a join to the UOM dimension to ensure that the fact value is actually in a unit that has been defined as quantitative.

```
SELECT SUM(F.VALUE)
  FROM FACT F
INNER JOIN UOM_B B ON (B.GROUP_ID = F.UOM_GROUP_ID AND B.ROLE = 'Value')
INNER JOIN UOM_D D ON (D.MASTER_ID = B.MASTER_ID
                      AND D.STATUS_INDICATOR = 'A'
                      AND D.CLASS = 'Quantitative');
```

It is also a helpful convention to build a version of the query (with the same dimensional constraints and filters for determining the appropriate facts to be considered) that counts the number of facts that were excluded from the first query because the value of those facts was not numeric.

```
SELECT COUNT(*)
  FROM FACT F
INNER JOIN UOM_B B ON (B.GROUP_ID = F.UOM_GROUP_ID AND B.ROLE = 'Expected')
INNER JOIN UOM_D D ON (D.MASTER_ID = B.MASTER_ID
                      AND D.STATUS_INDICATOR = 'A'
                      AND D.CLASS = 'Quantitative');
```

If this query returns a zero count, the user knows that no invalid values had to be omitted from the summed aggregation. For counts greater than zero, the user could query those values to determine any functional impact on what they are trying to do with their aggregation results.

```
SELECT F.VALUE, COUNT(*)
  FROM FACT F
  INNER JOIN UOM_B B ON (B.GROUP_ID = F.UOM_GROUP_ID AND B.ROLE = 'Expected')
  INNER JOIN UOM_D D ON (D.MASTER_ID = B.MASTER_ID
                         AND D.STATUS_INDICATOR = 'A'
                         AND D.CLASS = 'Quantitative')
 GROUP BY F.VALUE
 ORDER BY COUNT(*) DESC;
```

It is very common to receive values for facts that are defined as quantitative but don't actually pass the numeric test in the ETL. This can certainly happen because an actual error occurs in the data value, but it often occurs in the source system for legitimate reasons. We view the nonnumeric data as *technically* wrong but accept it as *legitimate* because the source system intended to send the data the way it did. The most common example of this type of error occurs in laboratory results where some quantity is expected as the result value, but the actual value received includes a less than (<) or greater than (>) sign. The inclusion of the sign invalidates the value as a number in a technical sense, although the *functional* interpretation of the fact is still valid. As a result, when querying facts in such a way that aggregation isn't an issue, I recommend displaying the expected UOM instead of the value UOM if present.

```
SELECT F.VALUE,
       COALESCE(D2.DESCRIPTION, D1.DESCRIPTION) AS UOM
  FROM FACT F
  INNER JOIN UOM_B B1 ON (B1.GROUP_ID = F.UOM_GROUP_ID AND B1.ROLE = 'Value')
  INNER JOIN UOM_D D1 ON (D1.MASTER_ID = B1.MASTER_ID AND D1.STATUS_INDICATOR = 'A')
  LEFT JOIN UOM_B B2
    INNER JOIN UOM_D D2 ON (D2.MASTER_ID = B2.MASTER_ID AND D2.STATUS_INDICATOR = 'A')
   ON (B2.GROUP_ID = F.UOM_GROUP_ID AND B2.ROLE = 'Expected');
```

There will be many cases where an expected UOM is injected by the ETL against values that users might not have considered. A common example is the dosage fact value associated with a medication order or administration. Dosage is usually defined as a numeric quantity in source systems. Your data analyst most likely mapped it into the metadata using an implicit numeric unit. Since many actual dosage facts arrive from sources containing both the dosage quantity and the dosage units in the same source value (e.g., 5 mg, 2 tabs), they will fail the type check in the ETL and end up in the Unexpected Text UOM.

Consider that while many dosage facts might be arriving with these data-type problems, there might be other dosage facts arriving from other sources that don't have these problems. There will be dosage facts in the Unexpected Text UOM like 5 mg, as well as dosage facts in the numeric UOM that are

dimensionalized to the “mg” UOM. I consider this scenario a positive value-adding opportunity for the warehouse, so it is important not to have the analyst recognize this situation in advance and define the incoming facts as being in a narrative or textual unit. Defining the metadata implicit UOM as *narrative* accepts and institutionalizes the problem, while defining it as *numeric* creates visibility into the problem so your data governance group can address it.

Pending instructions or approval from your data governance function, you can write a spider update function that looks at all of the facts in the Unexpected Text UOM. For all of those values that are numbers followed by small texts (e.g., 350 mg), you can parse out just the text and see if that value identifies anything in the UOM dimension. If it does, you can update the fact value to include just the number component of the original value and update the fact’s UOM dimensionality to point instead to the newly recognized UOM entry in the Value role. The original UOM group that contained the Unexpected Text reference no longer applies to that fact, so the updated fact becomes available for aggregation operations along with a more explicit and precise UOM for display.

Values that had more complicated text than just the UOM name or abbreviation will remain in the Unexpected Text UOM. These values represent your ongoing data cleansing opportunities. Some will have alternative values that can be better cleansed by adding a few new aliases to the UOM dimension for the associated units. Others might be more complicated, requiring new sourcing or parsing strategies to be put in place. The main theme we’re focused on here is that these conditions, even if they can’t be corrected right away, are highly visible in the warehouse. Your data governance and data quality functions have all of these raw materials to work with in instituting improvement or corrective actions.

Range Checking

A follow-on edit to the data type check in the ETL is the numeric range check. For fact values that are defined as numeric and have passed the data type check for actually being numeric, you can add a range check to assure that the fact values that are received fall within an acceptable range of values as determined by the source data analyst. To implement this check, add two new columns to your UOM Definition table: low-range value and high-range value. Populate these values for any UOM, where you want to invoke a range check in the ETL. The values can be used together to establish a defined range or individually to specify an expected low or high value. The edits in the ETL then take on a similar form as the data type check for numeric values. If the low-range value is not NULL, ensure the fact value is greater than or equal to the specification. If the high-range value is not NULL, ensure the fact value is less than or equal to the specification. When a validation fails, you should update the dimensional reference to UOM in the same manner as you did for the Unexpected Text entry, except in this case, the resulting Value bridge entry should point to the Range Violation

row in the UOM dimension. Alternatively, you can define a Low-Range Violation UOM and a High-Range Violation UOM if you would prefer to have more specific visibility into the two separate conditions. Note that these new system-level UOM entries are defined as quantitative, so the values can still be included in aggregations. If a user wants to exclude the out-of-range values from a query, they can be filtered on the UOM.

```
SELECT SUM(F.VALUE)
  FROM FACT F
INNER JOIN UOM_B B ON (B.GROUP_ID = F.UOM_GROUP_ID AND B.ROLE = 'Value')
INNER JOIN UOM_D D ON (D.MASTER_ID = B.MASTER_ID
                        AND D.STATUS_INDICATOR = 'A'
                        AND D.CLASS = 'Quantitative'
                        AND D.UNIT <> 'Range Violation');
```

As with the data type check earlier, it is a helpful convention to build a version of the query (with the same dimensional constraints and filters for determining the appropriate facts to be considered) that counts the number of facts that were excluded from the first query because the value of those facts was out of range. This second query counts the number of out-of-range values that were omitted from the main user query and sums their values to determine the impact on the reported sum in the first query.

```
SELECT COUNT(*), SUM(F.VALUE)
  FROM FACT F
INNER JOIN UOM_B B ON (B.GROUP_ID = F.UOM_GROUP_ID AND B.ROLE = 'Value')
INNER JOIN UOM_D D ON (D.MASTER_ID = B.MASTER_ID
                        AND D.STATUS_INDICATOR = 'A'
                        AND D.UNIT = 'Range Violation');
```

If the count or sum values are small relative to the result of the broader query, users can choose to ignore the differences or at least allow them in their analysis. If the values are large, users can query the underlying data in order to determine a more appropriate course of action. For general queries of range-related data, the same COALESCE that allowed the original UOM in the expected role to appear in queries instead of the error-related Value role will resolve these unit displays as well.

Value-Level UOM

In the Beta version Fact pipe, a fact that didn't have a UOM explicitly assigned in the source might have an implicit UOM assigned through the metadata for the fact. Additionally, if the UOM dimension contained a Value entry for the explicit or implicit UOM, the fact would be automatically realigned to take advantage of that specific Value entry in the dimension even if the source data didn't include it. The logic in the Beta version depended upon the Value entries in UOM being

present when the new facts arrived, thereby introducing a synchronization and alignment problem as new Value entries were added or old Value entries deleted, from the UOM dimension. Over time, facts end up not pointing to their correct entry. It's time to fix that problem.

The analysis of new source data typically involves profiling the data that occurs in each column of potentially new input and developing an understanding of common characteristics that occur in the data. One common characteristic is that certain values appear much more often than others in certain columns. In this discussion, we're interested in values that occur frequently in fact columns. As you analyze data for your new warehouse, you'll invariably see certain types of facts where the range of values is extremely diverse and is sometimes dominated by a relatively few values. If, instead, you see that the fact is completely covered by only a few values, those values probably belong in a dimension, not in the Fact table. Here, we're concerned with facts that have unlimited distinct values over time but a subset of those values that tends to dominate the profile of the source column. For example, when you load tens of millions of laboratory results, you will find that the values of those facts vary considerably and widely. The two most common lab result values are "Negative" and "Yellow," and these two values sometimes account for 20%–30% of the facts of that type. You'll want to be able to include, exclude, or count these values in a query without having to access the Fact table.

There are typically many other scenarios offering the same basic opportunity. You'll have facts that are counts where most of the values are zero or one. You'll see dollar amounts where a handful of values represents the bulk of the data (i.e., usually \$0 and \$1, but also including the most common room and procedure charges). These are precisely the situations in which you can take advantage of the value-level UOM processing to make the access of these distinctive values more efficient. To use value-level UOM entries, each desired value must be defined as a row in the UOM dimension. Value rows are generally loaded through standard ETL processing, although it isn't uncommon for the source intake for the new Value rows to consist of a value-profiling query to identify commonly occurring fact values.

Suppose you have the following two UOM: a Result Text being managed as a measure row in the dimension and a count that is being managed as a unit (with no semantic measure implications) (Table 14.1).

Presumably your source data analyst is mapping the source data for some of your facts to these entries in the UOM dimension, most likely through the implicit UOM feature of the fact metadata. Over time, your support team notes

Table 14.1 Definition Entries for a Unit and Unit-Measure

<i>Subdimension</i>	<i>Scale</i>	<i>Class</i>	<i>Unit</i>	<i>Measure</i>	<i>Language</i>	<i>Value</i>
Measure	Narrative	Narrative	Text	Result		
Unit	Quantitative	Quantity	Count			

Table 14.2 Definition Entries for Aligned Unit/Measure and Value

<i>Subdimension</i>	<i>Scale</i>	<i>Class</i>	<i>Unit</i>	<i>Measure</i>	<i>Language</i>	<i>Value</i>
Measure	Narrative	Narrative	Text	Result		
Value	Narrative	Narrative	Text	Result		Negative
Value	Narrative	Narrative	Text	Result		Yellow
Unit	Quantitative	Quantity	Count			
Value	Quantitative	Quantity	Count			0
Value	Quantitative	Quantity	Count			1

that, of the facts that are using these two UOM entries, there are two dominant values for each of those facts. On a case-by-case basis, your interpretation of whether or not a set of values is dominant will vary based on your local requirements, particularly related to how you expect your users to query the data in question; but a working scenario of 5% of the values making up 25%–40% (or more) of the values is reasonable as a starting point for planning. Those identified values should be added to the UOM dimension as Value entries, typically by the warehouse support team (Table 14.2).

With the four Value rows added to the dimension, when new facts arrive that target, one of the two original rows, those facts will be altered to point to the appropriate Value row if the actual value in the fact matches the value in the associated Value row. A new Result Text fact in the Fact table will now point to one of the three UOM entries: (1) the “Negative” Value row, (2) the “Yellow” Value row, or (3) the measure row. A new count fact in the Fact table will now point to one of the three UOM entries: (1) the “0” Value row, (2) the “1” value, or (3) the Unit row.

Having these distinctions in the UOM dimension allows queries to take better advantage of the dimension to improve performance. It becomes possible to include or exclude certain values from a query by filtering on the UOM value column rather than on the Fact value column. For quantitative data, the distinction also makes algebraic aggregation easier. Suppose you want to take a sum of the count facts. Without Value rows, the query would perform a sum on the Fact value column. With the two Value rows, you can easily exclude the zero values from your query since they won’t affect the total. You can then take the sum of all fact values still pointing to the unit row and add a count of the facts pointing to the value “1” row. Presuming that the zero and one values are the vast majority of the facts, the query has now limited its access to the Fact value column to only those that actually need to be accessed in order to summarize their values. Taken to the extreme, if all of the values of count facts are placed in the UOM value model, the facts could be summed without actually touching the Fact table. Since the Fact table might have billions of rows in it, you will see significant performance improvements for certain fact-value-based queries and aggregations.

The Beta version already supports the addition of Value rows to the UOM dimension through standard Reference and Definition pipe processing. The Fact pipe already takes advantage of Value rows in assigning UOM dimensionality to new facts. However, the Beta version lacks an integrating function to realign fact-to-UOM dimensionality as Value rows are added from the UOM dimension. The Fact table might already contain facts that match the new Value rows but were processed through the ETL before the new Value row was added to the warehouse. You add functionality in the Gamma version that automatically realigns those legacy facts when new Value rows are added or removed from the UOM dimension.

For example, suppose when you add your “Negative” and “Yellow” values to the Result Text UOM, there are already tens of millions of facts pointing to the Result Text measure. Any number of those facts might contain the values “Negative” or “Yellow” and could now be pointing to the wrong entry in the UOM dimension. You need to add a utility that will correct this problem any time you add a Value row to a UOM. The logic is basically the same as the Beta version logic for assigning value identifiers to facts, except that the new logic is applied to *existing* facts on the Fact table rather than to new facts in the Fact pipe. An UPDATE statement against the Fact table will do, provided a list of newly added Value rows is available. This is why this new logic is added to the Fact pipe, using a UOM-filtered Reference pipe input to drive processing.

You will also need a utility for the less common situation where a UOM Value row is removed from the dimension. In this situation, the facts that are pointing to the UOM Value row being deleted must be realigned to point to the appropriate Unit or Measure row. Again, an UPDATE statement against the Fact table can accomplish this. Using the staged Reference pipe to identify Value rows that are being deleted, join the UOM Value rows to the Fact table through the “Value” role. The result set will include the facts that need to be updated. Join those facts back to the UOM dimension for the unit or measure, updating each fact’s UOM Group ID with the Master ID of the new target UOM entry.

The frequency with which you add or delete Value rows in the UOM dimension depends upon several factors, most addressing a desire to gain some form of efficiency during certain queries. Value entries are particularly helpful when there are fact values that will be used as query filters in some form, such as when functionally neutral values are excluded (e.g., Value \neq “0” or Value \neq “Negative”) or specific values are being sought (e.g., Value = “350” or Value = “Yellow”). Given the size of the Fact table, filtering on the value column can impact the performance of a query very quickly compared to being able to filter the value within the UOM dimension. By extension, Value entries offer alternatives to queries that might be looking to count instances of facts by value, such as in counting administrations of a medication by the dosage of the orders. Since medications tend to only be administered with a modest range of dosage values, it’s reasonable to completely populate Value rows in the UOM dimension

for every expected dosage, making it possible to query or count facts by dosage without ever actually needing to access the Fact table.

Undetermined Dimensionality

Not all source datasets contain natural keys for all of the dimensions that you will want your facts to reference. Many datasets have a limited view of data that include only information needed by the application system that creates or maintains the dataset. When this occurs, you have three options: (1) load the facts without the desired dimensionality, (2) add enough logic to your source intake job to be able to access and source the needed natural keys, or (3) source and load the facts with the dimensional reference as Undetermined and resolve the dimension reference later. Of these options, the first is undesirable because you won't want to load facts that lack desired dimensionality, although it happens all the time when the needed source data are unavailable. The second option of pursuing more data through extensive joins in the sourcing query works, but at a high cost in terms of increased complexity and the need for testing of the expanded source intake queries. It violates our desire both to keep the source intake queries as simple as possible and not to repeat logic across multiple SDI jobs that can be institutionalized in a more generic way. The third option is the best choice because it keeps the SDI query as simple as possible and allows common generic logic to be developed within your ETL process flow.

An example of where Undetermined processing is often needed in healthcare is in the source datasets that contain financial data, particularly charges. I've loaded many financial charge datasets that included encounter or account identifiers but not patient identifiers. As a result, the natural key for the Interaction dimension was available, but the dataset lacked an identifier for the Subject dimension. In every environment I've worked in, it hasn't been difficult to find one or more tables that could be joined to my source to derive a patient identifier for each associated encounter or account. My concern is that the logic to do so is complicated and might be needed in many SDI jobs for many data sources. The nature of this generic ETL architecture is that we'd like to satisfy common requirements like this one at a single appropriate point in the ETL stream, not in each individual sourcing query that might need to search out additional keys.

Instead of taking on the extra burden of extensive join logic in each SDI selection query, you can get around that problem in this example by sourcing a Patient ID as "Undetermined" in the system context. Because the System subdimension is always present in every dimension, this logic can be used for any dimensionality. You end up with many facts dimensionalized to the Undetermined subject, so now you'll implement the additional logic

to complete your desired dimensionality. Since each dimension could have facts dimensionalized as Undetermined, your resolution logic will need to be fanned out across the dimensions. Most of these entries can be resolved with a fairly simple lookup to other common facts that will include the desired dimensionality. Some dimensions might need more than one adjustment update query if the logic needed gets more complicated. You'll have to determine that through your source analysis and comparison of the dimensionality of closely related facts.

In the case of my financial facts lacking a patient identifier for the Subject dimension, the simplest resolution is typically to find other facts for the same encounter that have been successfully dimensionalized to the correct patient for the encounter. The first step is to find all the facts with Undetermined Subject dimensionality and obtain the Master ID of their primary Interaction dimension group.

```
SELECT FACT_ID AS UNDETERMINED_FACT_ID, IB.MASTER_ID AS INTERACTION_MASTER_ID
  FROM FACT F
  INNER JOIN INTERACTION_B IB
    ON (F.INTERACTION_GROUP_ID = IB.GROUP_ID AND IB.WEIGHT = 1 AND IB.MASTER_ID > 6)
 WHERE F.SUBJECT_GROUP_ID = 3;
```

The query takes advantage of the fact that the Undetermined row in each dimension has a Master ID of three, and it presumes that if the Interaction group has multiple bridges, the desired encounter or account would have a weight of one. It also assures that the Interaction dimensionality is not pointing to a System row (e.g., unknown, Master ID = 2). This query is then used as a subquery for a new query that will look for the appropriate patient identifiers for these facts, expecting that there is only one dimensionalized patient for each encounter.

```
SELECT UNDETERMINED_FACT_ID,
       MAX(MASTER_ID)          AS NEW SUBJECT_MASTER_ID,
       COUNT(*)                 AS SUBJECT_COUNT
  FROM (
    SELECT DISTINCT FL.UNDETERMINED_FACT_ID, SD.MASTER_ID
      FROM FACT F
      INNER JOIN SUBJECT_B SB ON (F.SUBJECT_GROUP_ID = SB.GROUP_ID)
      INNER JOIN SUBJECT_D SD ON (SB.MASTER_ID = SD.MASTER_ID
                                  AND SD.SUBDIMENSION = 'Person')
      INNER JOIN INTERACTION_B IB ON (F.INTERACTION_GROUP_ID = IB.GROUP_ID)
      INNER JOIN
        (
          SELECT FACT_ID AS UNDETERMINED_FACT_ID,
                 IB2.MASTER_ID AS INTERACTION_MASTER_ID
            FROM FACT F
            INNER JOIN INTERACTION_B IB2
              ON (F.INTERACTION_GROUP_ID = IB2.GROUP_ID
                  AND IB2.WEIGHT = 1 AND IB2.MASTER_ID > 6)
             WHERE F.SUBJECT_GROUP_ID = 3
        ) FL ON IB.MASTER_ID = FL.INTERACTION_MASTER_ID )
 GROUP BY FL.UNDETERMINED_FACT_ID;
```

This query looks up all DISTINCT Subject dimension entries in the Person subdimension that are associated with any fact for the Interaction identifiers found in the subquery. The expectation is that only one Subject entry will be found for each Interaction, but the query includes a COUNT clause just in case that condition is not true for some encounters. If the count returned is one, then the fact indicated in the Undetermined Fact ID should be updated to replace its Subject Group ID with the New Subject Master ID. If the count returned is greater than one, the Subject Group ID in the fact should be updated to reference the Ambiguous entry (Master ID = 4) in the System subdimension. Since the encounter is associated with facts for multiple patients, this algorithm won't resolve the Undetermined entry, but the Ambiguous reference will trigger corrective action in the control processing described in Chapter 15.

Every situation in which an Undetermined entry will be referenced by a fact could be different, but the actual number of algorithmic differences that I've seen is fairly small compared to what could conceivably happen, and the code for all those variations can usually be managed as a single late ETL workflow. The code in my missing-patient example will work for any fact missing a patient for a known encounter. If differences occur, additional filters might be required to tailor queries to handle specific cases. This example would result in an Ambiguous assignment if an encounter includes any multi-Subject facts because of the presence of two patients. Multi-Subject facts are rare, but they do occur. For example, transplant and blood donor facts often have two subjects: a donor and a recipient. To resolve these ambiguities, you might choose to exclude one of those roles from the query in order to allow the facts to resolve to one of those patients.

You'll also see situations where an Undetermined reference doesn't resolve because no subjects are returned for an Interaction. It's possible that no subjects are found because you haven't yet loaded any facts with legitimate Subject references. This is extremely rare, so I would question any situations where that appears to happen. It's far more likely that you've only got facts that resolve to something other than the patient in the Subject dimension. For example, you might have only lab results that dimensionalize to Specimen subdimension entries, so your query didn't see them. In those cases, you will need to expand your query to include joins through the Subject hierarchy to obtain an ancestor entry in the Person subdimension.

```

INNER JOIN SUBJECT_B SB ON (F.SUBJECT_GROUP_ID = SB.GROUP_ID)
    LEFT JOIN SUBJECT_D SD ON (SB.MASTER_ID = SD.MASTER_ID
                                AND SD.SUBDIMENSION = 'Person')
    LEFT JOIN SUBJECT_H SH ON (SH.DESCENTENT_MASTER_ID = SB.MASTER_ID
                                AND SH.RELATION = 'accessioned against')
INNER JOIN INTERACTION_B IB ON (F.INTERACTION_GROUP_ID = IB.GROUP_ID)
INNER JOIN
    ( subquery ) FL ON IB.MASTER_ID = FL.INTERACTION_MASTER_ID
WHERE SB.MASTER_ID IS NOT NULL OR SB.ANCESTOR_MASTER_ID IS NOT NULL
  
```

This query takes advantage of functional knowledge that the facts in question will be either directly dimensionalized to a Person entry in subject or to a Specimen entry that has been “accessioned against” a Person entry. Since every dimension entry is its own ancestor through the Self hierarchy entries, adding the hierachic connection solves the specimen-only problem without impacting other simpler resolutions.

The queries that resolve Undetermined dimensionality aren't necessarily simple in content or structure. The purpose of placing them at the end of the generic ETL stream is to prevent them from being added to many of the SDI jobs that are needed to feed the ETL. In the meantime, the expectation is that these queries will be used and reused across multiple source scenarios that exhibit the same Undetermined dimensionality, even if the original causes of the omissions in the source data differ. The logic of assigning a patient to an encounter will prove very stable across a large number of source dataset processes. Placing that code in the Gamma version once is preferable to placing that code into dozens, or hundreds, of SDI queries.

You need to develop one additional piece of generic capability for resolving Undetermined references that involves adding some fuzzy logic to some of the algorithms. The aforementioned example involved finding a patient to assign to a charge for a known encounter. Suppose it had been the reverse: a known patient but an Undetermined encounter. The overall logic would be the same: If there's only one encounter that fits the criteria, assign it to the fact; but if there are multiple encounters, assign the fact as Ambiguous. This problem arises when trying to determine matches when the cardinality of the criteria might be expansive (i.e., an encounter is only for one patient, but a patient might have many encounters). A fact might arrive for a time slot when no encounter is active for the patient. This happens all the time with charges, bills, payments, and lab results. Fortunately, these facts typically arrive fully dimensionalized; but when a fact includes an Undetermined encounter, the logic gets fuzzy. I typically look for encounters for the patient within a selected time period before and after the fact. The amount of time I use to make the time period fuzzy might vary from a few hours to a few days depending upon the type of fact for which I'm trying to resolve the Undetermined reference. If I make the fuzzy time period wide, I'll get multiple encounters for some patients more often, and the facts will be altered to reference the Ambiguous row. I prefer this approach to the alternative of using a small time period and, as a result, periodically updating facts with the wrong encounter. As you develop controls for the Ambiguous mappings in Chapter 15, you'll adjust the desired fuzziness of these algorithms based on your experience in working with the data governance function to resolve the ambiguities. If Ambiguous turns out to be a frequent false positive, you'll tighten your fuzzy logic. Allow your data governance function to drive any capabilities or tools that you develop in this area. They will decide on an appropriate balance between Alpha risk (i.e., referencing the wrong encounter) and Beta risk (i.e., referencing Ambiguous when a valid encounter was available).

ETL Transactions

The standard data layout in all of the SDI intake datasets includes a column for each SDI job to specify the ETL transaction to be processed against any particular Break ID in the source data. In the Beta version, we ignored that column and treated all arriving data as new fact data to be added to the Fact table or as master data to be added or changed in the dimensions. In the Gamma version, we drop that assumption and allow other transaction types to be processed. We'll continue to default the transaction type to "Add/Update" if it arrives as a NULL value from the SDI source, in order to provide backward compatibility to already existing source datasets. I recommend leaving the column NULL in the sourcing jobs if the "Add/Update" is being used in order to save memory, since that value is by far the most common and there are usually millions of rows of data passing through the ETL pipes during any particular execution.

The actual coding changes that are required to support additional transaction types will vary depending upon what transactions are implemented and how your organization chooses to define their purpose. While the list of transactions varies, there are some fairly common transaction types that I recommend, which are as follows:

- *Add dimension master*: This transaction will add a new entry into a dimension, or it will autoadopt an existing orphan entry; but it will not update that entry if it already exists. This transaction is useful for supplementing data that have a more reliable source but aren't considered primary enough to cause it to override the data already in the warehouse. For example, when a full ICD-9 master table is loaded into the Diagnosis dimension after the organization's local ICD-9 master list has been loaded, the intention of the transaction is to acquire additional codes not present in the organization's master list without necessarily impacting local descriptions of existing codes that were previously loaded.
- *Update dimension master*: This transaction will update existing dimension entries but will neither add new entries nor autoadopt orphan entries.
- *Initialize dimension master*: This transaction will autoadopt existing orphan entries or update values in existing rows that are currently Null. It will not create new dimension Definition entries.
- *Delete dimension master*: This transaction will change the Status Indicator of an existing dimension entry to Deleted. It will not physically delete the entry.
- *Add fact*: This transaction will only insert new facts.
- *Update fact*: This transaction will only apply updates to an existing fact, including a new fact value or different dimensionality. The transaction requires that a Fact ID be provided in the source data.
- *Delete fact*: This transaction will delete an existing fact from the Fact table by changing its data state to Deleted. The transaction requires that a Fact ID be provided in the source data.

The reality is that I have only rarely seen clients implement any of these additional transaction types into their ETL workflows. Circumstances where these transactions would be invoked are relatively rare, and most clients prefer to intervene in these situations on a more manual basis. It is very easy to apply very specific update logic directly to the warehouse database, making it less likely that you'll commit the resources to permanently establish this additional functionality. I've included it here for a sense of completeness, acknowledging that I don't have universal or generic code available for implementing these options. Most of the changes required to implement these transactions in the ETL code impact the Definition or Fact pipes. The add/update logic already in those pipes becomes one option in a branch condition that is based on the transaction to be processed. The change always *logically* takes place at that point in the pipe, and the new logic for each transaction is added as a separate branch. You might have to *physically* implement some of these branches differently depending upon how you've architected any use of your database management system bulk loader. Bulk loading can alter the physical order in which logical transactions have to be executed.

From time to time, you'll also discover places in the generic ETL logic where you will want to put your own organization-specific processing or data handling. To do so, you should "invent" a transaction type that can be evaluated by a condition in the ETL flow at your desired point. By sourcing transactions with the invented Target Transaction, you can invoke your own custom logic without having to make additional extensive alterations to the ETL generic code.

Target States

As with the ETL transactions, the standard data layout in all of the SDI datasets includes a column for each SDI job that specifies the Target State to be processed against any particular Break ID in the source data. Data in the Beta version are always added to the Active data state. You ignored the column and treated all arriving data as Active data. That limitation is removed from the Gamma version to allow data to be loaded into any desired state, including Deleted, Merged, Inactive, Corrected, Quarantined, or Superseded. You should also allow for organizationally specific additional states to be defined.

You'll continue to default the target state to "Active" if it arrives as a NULL value from the SDI source, in order to provide backward compatibility to already existing source datasets. Like the Target Transaction, to save memory, I recommend leaving the Target State column NULL in the sourcing jobs if the Active state is desired, since that value is by far the most common, and there are usually millions of rows of data passing through the ETL pipes during any particular execution.

The coding change required to support additional Target States takes place in the Fact pipe. In Beta, we set the Data State identifier to always point to the Active state in the Data State dimension. The required new logic should obtain the Master ID for the Target State in the source data to set the Data State dimension identifier in each fact.

Superseded Facts

There are many circumstances where a new fact supersedes a fact already in the warehouse. This situation probably already exists in some of the data already loaded in the Beta version, but all those facts are still in the Active state because the Beta version didn't include any logic to identify facts that were being superseded. While users can be taught to query the warehouse so only the most current version of any particular fact is returned in a query, the logic of doing so can be quite complicated. Many users lack the skill necessary to build these queries consistently. For this reason, the warehouse ETL needs to support this requirement by automating the recognition and handling of new facts that supersede old facts. The result of the new logic will change the data state of the older facts to Superseded, so users will not see these facts in their queries if all facts are filtered on the Active data state. The data will remain in the Superseded state in order to allow queries to include these facts, when desired.

Unfortunately, I've never found an absolutely generic way to find all facts that need to be superseded. Instead, I've identified patterns among these facts that have worked for me over time and collectively handled the majority of the situations in which superseding logic is needed. There are columns available in the Metadata dimension to support these scenarios, although we didn't use them in the Beta version. The values of these columns imply a query of the Fact table, the results of which would be a collection of facts that need to be changed to the Superseded data state. Some examples include the following:

- A patient should have only one active home address fact.
- An encounter should have only one active admission fact.
- A performed laboratory test should have only one active result fact.
- An encounter should have only one active attending physician assignment fact.

The first metadata column that helps define this logic is the Superseding Orientation. This value specifies the range over which the superseding logic will

be evaluated. The orientation for the supersede logic can be any dimension but is typically one of the following:

- *Subject*: There should be only one active fact per Subject, typically a Patient.
- *Interaction*: There should be only one active fact per Interaction, typically an Encounter.
- *Caregiver*: There should be only one active fact per caregiver, typically a Provider.
- *Operation*: There should be only one active fact per Operation, typically an Order.

Supersede orientations in the other dimensions can be implemented using the same algorithms, but I rarely see requirements for anything other than these examples.

The second metadata column involved in the superseding logic is the Superseding Period. This value determines the time period over which an active fact will supersede earlier versions. Most superseding involves only having one active fact of any given type, and this is characterized by specifying “Ever” in the period. Alternatively, the Period might be set to “Year” or “Month” or “Day” if the intent is to allow one active fact during any of these grains of time.

The superseding metadata for the one-per-Subject and one-per-Interaction examples would be as follows:

<i>Metadata Column</i>	<i>Value</i>
SUPERSEDING_ORIENTATION	Subject
SUPERSEDING_PERIOD	Ever

<i>Metadata Column</i>	<i>Value</i>
SUPERSEDING_ORIENTATION	Interaction
SUPERSEDING_PERIOD	Ever

Three other metadata columns support more complicated superseding logic, and they are needed for the aforementioned active attending provider example. The Superseding Dimension, Role, and Rank allow a specific entry to be included in a dimension other than the Superseding Orientation dimension. Here’s an example set of values for the single attending provider assignment.

Metadata Column	Value
SUPERSEDING_ORIENTATION	Interaction
SUPERSEDING_PERIOD	Ever
SUPERSEDING_DIMENSION	Caregiver
SUPERSEDING_ROLE	Attending
SUPERSEDING_RANK	1

These values presume that the metadata against which this logic is being defined could have assigned more provider roles than just the attending provider, so the specifics are encoded in the metadata. Had the metadata been specific to attending provider assignments, the Dimension, Role, and Rank would have been unnecessary; although they would work just fine as long as they match the corresponding data values used to dimensionalize the fact.

The metadata for superseding the laboratory result would be as follows:

Metadata Column	Value
SUPERSEDING_ORIENTATION	Operation
SUPERSEDING_PERIOD	Ever
SUPERSEDING_DIMENSION	Procedure
SUPERSEDING_ROLE	Resulted
SUPERSEDING_RANK	

These data presume that the order information for the laboratory test was dimensionalized in the Operation dimension and that the laboratory test in question was dimensionalized to the Resulted role.

The data columns available in the Metadata dimension to support the superseding of facts have evolved over time and can handle the bulk of the various superseded fact opportunities that present themselves in biomedical data. They don't handle all situations yet, however. The primary limitation is that the logic of superseding takes place within a single fact type: one Metadata dimension row. I haven't yet developed a generic way for the arrival of one type of fact to indicate that another type of fact should be superseded (i.e., a *final* lab result supersedes a *preliminary* lab result when the two facts are defined using different metadata). Most of these cases get resolved by improving the dimensionality of the mappings so the two types of facts become recognized as one fact type (i.e., the status of preliminary or final is moved into the Quality dimension). Once collapsed to a single metadata definition, the superseding logic correctly handles the status-based superseding between preliminary and final results.

I'm experimenting with the Metadata hierarchy as a possible means for expanding the generic functional architecture. One metadata entry could be defined as superseding another, so cross-fact superseding could be enabled. A hierarchy could eventually also provide many other fact-to-fact extensions to the model or even cross-dimensional editing and control. For example, a common editing criterion during fact loads is to ensure that one or more events actually occurs or that certain events occur in an expected order (e.g., lab order, sample collected, sample accessioned, preliminary lab result, final lab result, corrected lab result; or allergy assessment, medication order, medication administration, medication reconciliation). A metadata hierarchy, with relationships defined for each criterion of interest, could enable these features in a fully generic way. For now, you'll have to build these capabilities directly into the relevant ETL pipes, most of them in the Fact pipe.

Continuous Functional Evolution

Depending upon which Gamma version features you've chosen to implement, your data warehouse ETL is now functionally complete. Features that you've chosen to defer can be added later, each at the appropriate location in one of the generic ETL pipes. You've probably also implemented considerable functionality into your ETL that is very specific to your organization and its requirements. These might include user exits through the ETL transaction process, organizationally specific Data State values, or unique fact superseding logic routines for situations not handled by the standard metadata-based capability. These customizations are normal in an evolving architecture, and they serve as the source for future expansions of the generic architecture.

All of the Gamma features of the warehouse, and most of the Beta features, came about as I've observed the tailoring and customizations made to data warehouses that looked most originally like today's early Alpha versions. It is by monitoring your customizations that you'll uncover further opportunities for expanding the generic capabilities of your warehouse. In this chapter, patterns I've previously recognized became functional capabilities in the ETL. Other patterns involve different ways users look at data in their queries, and you'll implement new database views in Chapter 17 to support them more generically. In the next chapter, you'll see a series of data controls that have emerged and evolved in the same way. The evolution of your warehouse should be driven by the actual functional needs and behaviors of your users. It is continuous because the more you implement, the more they'll adapt and use new features in creative and unanticipated ways.

Chapter 15

Establishing Data Controls

The functioning warehouse is not quite ready for production, but it can be used for queries in a variety of meaningful ways. The warehouse is probably already being queried by a variety of users who are working directly with the development team, but they aren't yet able to access the warehouse on their own. Still needed are a series of technical and organizational controls that will ensure appropriate use in production. The technical controls include capabilities to secure, monitor, and tune the operational warehouse so it provides sound, high-performance functioning. The organizational controls introduce data administration and data governance capabilities that emphasize proper HIPAA compliance and source data error correction. Until these final features are put in place, the Gamma warehouse can only be directly used by the development team. Data controls will enable a turnover to production for stand-alone user access.

Finalizing Warehouse Design

The database you created in the Alpha version of the warehouse has remained mostly unchanged. Until recently, the actual performance of the database hasn't been a key issue in our discussions. In the Beta version, we were more interested in getting the generic ETL architecture into place. As we begin implementing controls around the data, some of your attention will turn to finalizing the database design. I'm not a database administrator (DBA), so I won't be able to tell you exactly how to finalize your design. I have watched many clients work on their designs, so my comments here are based on my observations.

Database Statistics

At the end of Beta, you probably had a few hundred million facts loaded into the warehouse, each typically using five or six functional dimensions. That is usually

just enough data to be able to notice the performance differences between the Alpha and Beta versions. Most performance issues fall into one of two categories: ETL load timings or query execution times. As with any star-schema-architected data warehouse, the strategies for improving these two aspects of performance will differ because they emphasize different parts of the design. One common step I've seen everywhere is database statistics. Clients who update database statistics more often have far fewer performance concerns. I've seen queries that run for 90 minutes reduced to 10 seconds when statistics are updated.

Application Layer Issues

Database performance is also greatly influenced by the toolsets you use to load or access the data. If you are using an ETL tool suite, as well as a semantic layer for business intelligence access, those tools will impact database performance. The common situation I see is that these tools are sometimes very good at pushing the logic of any Structured Query Language (SQL) down to the database level for processing, while at other times, they rely more on application-level logic than necessary. If a query and subquery combination is pushed down to the database, the performance will be much better than if the first query is executed to obtain a result set at the application layer, followed by a second query against that result set in the application layer. Star-schema designs are often dependent upon stratified query strategies (in order to obtain and combine data) that might be orthogonal in terms of dimensionality. A query to identify a cohort of patients is usually a subquery to a broader query of clinical or operational conditions. If the connection between those two queries is made in the application layer, performance will always be impacted. This means that you need to completely understand your toolset capabilities in order to properly finalize your database design.

To mediate the performance risks introduced by these tools, I typically use a test strategy that involves taking any SQL that is performing poorly at an application level (e.g., ETL tool, business intelligence layer) and execute it at the database level. If the execution is equally poor at the database level, I know I'm dealing with a database challenge. If the execution is significantly better at the database level, I know I'm dealing with an application layer challenge. Each needs to be improved, but the steps I'd take toward these improvements will vary, and they can overlap in important ways.

At the application layer, I've seen issues arise in three basic areas: (a) the application's database connection or pipe, (b) the application's configuration options, and (c) the application's semantic mapping of the database. The first two issues require you to work with people who know the tools well. What does the database connection support that might improve performance? Does the configuration of the tool provide options that might impact performance? Some tools include settings that determine when and where different aspects of queries are performed, such as subquery execution or aggregation and distinct logic.

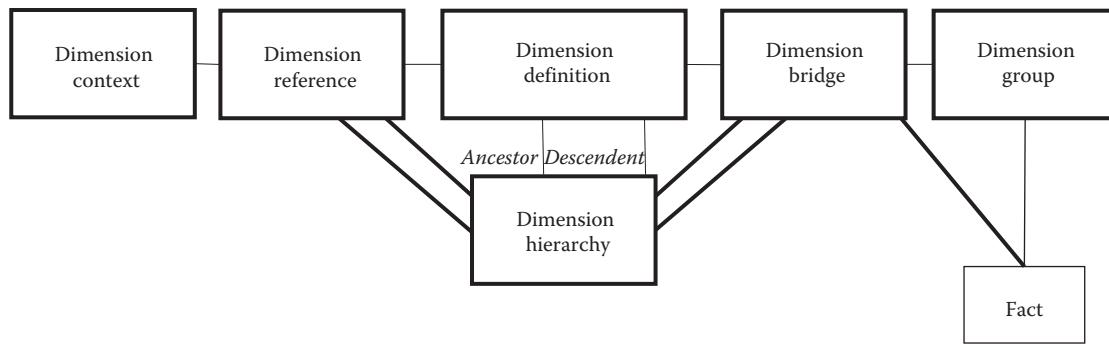


Figure 15.1 Alternate semantic pathways through dimension design pattern.

When possible, configure the tools to complete as much as possible down in the database layer. Many operations in the star schema interrogate millions of rows of data to obtain only hundreds of rows of results. My clients have found that bringing those millions of rows up to the application layer is a leading cause of performance degradation.

Fixing the semantic mapping issue involves making sure that every option for navigating the database has been included in the semantic layer. A common example is when the design pattern for a dimension is modeled too literally in the semantic layer. If the Fact table has only a semantic relationship to the Group table, every operation that passes from a fact to a dimension will need to access the Group table (Figure 15.1). The semantic layer also needs to provide a direct pathway from the Fact table to any Bridge table, making it unnecessary to access a Group table to get from a Fact to a Bridge entry.

Likewise, if the design pattern is implemented too literally, the Hierarchy table will have only ancestor and descendent pathways to the Definition table. Using the Hierarchy in any query will always entail two joins to the Definition table even if no properties are being pulled from that table. Instead, ensure that the semantic layer also provides two pathways between Hierarchy and Bridge and between Hierarchy and Reference. These additional pathways provide a means to minimize the number of joins generated by your tool's semantic layer, thereby improving performance of most of the more common queries.

Indexing and Partitioning

The biggest part of finalizing your database design is in the indexing and partitioning of your data. Once again, because I'm not a DBA, I can't provide you with the definitive model for your indexes and partitions. Every database management system provides different capabilities in support of these database functions, so you'll need to involve your organization's DBAs as well as those of your database vendor. In recent years, the design discussions I've participated in centered on just a few main issues, the functional characteristics of which I've been able to understand even though I don't necessarily know all of the DBA approaches for dealing with them.

The first issue I usually hear being debated is the use of the Master ID as the primary access point to the Definition tables even though the Definition ID is the primary key of the table. This usually translates into indexes that aid in the connecting of information in each dimension using Master IDs, particularly in getting from the Reference table to the Definition table during ETL loads.

The second issue that is debated involves the deepest grain of the star actually being the Bridge tables rather than the Fact tables as we might expect. It is the Bridge table that connects a Group ID to a Master ID, and that connection is central to any dimensional lookup from the fact side of the model. The indexing at that level needs to be bidirectionally functional. When filtering on dimension entries, a query needs to easily find Group IDs for a known Master ID. When establishing dimensional labels for facts, a query needs to easily find Master IDs for known Group IDs.

A third issue that is debated during design finalization involves my model's use of fairly large text columns as my primary search criteria: the Reference Composite in the Reference tables and the Group Composite in the Group tables. These two columns are the core enabler of the generic ETL architecture, yet many DBAs express concern at their use. An alternative I've seen implemented in several client sites is to define these columns as smaller hash columns to improve both storage utilization and query performance. I have no objection to these changes. I consider my model to be a logical database, so I'm not too concerned when these small physical changes are made that don't impact my functionality. There is usually the question of whether hash values are to be stored *instead of*, or *in addition to*, the original text values. Having the text values available does offer some support during testing since the actual values that go into a hash column are lost. Debugging of Group problems is easier when the actual generated Group Composite can be seen in the database. However, since I strongly discourage users from querying the Group table, my objections to having only a hashed Group Composite are minimal. Conversely, I see users query the Reference Composite in the Reference table all the time, so if a hash value is put in that table for performance reasons, I suggest doing it as an additional column.

Outrigger Tables

Another issue that is often raised during database design reviews involves the potential for the use of outrigger tables in my dimension design pattern. For the purposes of this discussion, an outrigger table moves the data from a table it's currently in to the outrigger table in order to improve performance. In this case, the only outriggers that would make sense are those that would be paired with the Definition table in each dimension. You could use an outrigger table to physically separate aspects of your definitions that are different enough to justify the performance concerns if the data were left together. Areas of concern in an outrigger discussion usually involve specialized columns in the Definition table that are NULL too often for large subsets of data in the dimension or

requirements for certain columns to be controlled as slowly-changing values in a manner that would create excessive or undesired volatility in the primary Definition table.

The Subject dimension is a place where you might choose to use an outrigger table. It's highly probable that you built your Subject dimension around a Person subdimension during the Beta version in order to load all of your patients into the dimension. Suppose you discover during your analysis that many of the facts that you've dimensionalized to Person entries are also available in different forms that are more specific, actually dimensionalizing to the specimen taken from the patient. You decide to add a Specimen subdimension to your Subject dimension to handle this more precise dimensionalization of those facts, so you connect each Specimen to its associated Person entry using a flat hierarchy. This kind of analysis is perfectly routine, and the option of adding a new subdimension to a dimension is exactly what the dimensional design pattern is for.

Upon further analysis, you find that the data you will be receiving for each specimen are large and unstructured, so you decide that you would rather not put those added columns in the Definition table. Instead, you choose to establish an outrigger for Specimen as depicted in Figure 15.2. The Subject Specimen Outrigger table serves as an outrigger to the Subject Definition table. As an outrigger, this new table allows the new columns that are specific to Specimen to be defined separately from the original Definition table. Each row has the same Master ID as the original table but stores data using its own Outrigger Specimen ID. The outrigger also supports slowly-changing values in the Specimen columns, independent of the same functionality in the original Definition table.

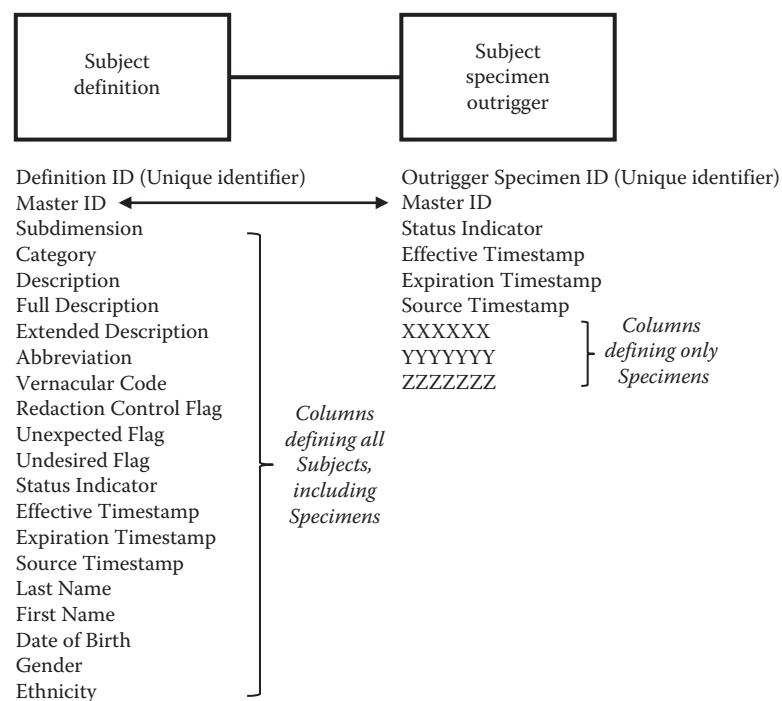


Figure 15.2 Subject definition with sample outrigger.

Because every database management system behaves differently, it can be difficult to precisely predict the impact of using an outrigger table in the design. A query of the outrigger can be accomplished as a direct JOIN in cases where a user knows that the subjects of interest are Specimen or through a more general LEFT JOIN that combines the data to look like it would have looked if the outrigger hadn't been defined.

```
SELECT *
  FROM SUBJECT_D D
  LEFT JOIN SUBJECT_SPECIMEN_O O
    ON O.MASTER_ID = D.MASTER_ID
   AND D.SOURCE_TIMESTAMP BETWEEN O.EFFECTIVE_TIMESTAMP AND O.EXPIRATION_TIMESTAMP;
```

Depending upon the grain of slowly-changing values on both sides of the definition–outrigger JOIN, the number of rows returned in certain queries will be different based on whether you joined the tables on the effective timestamp of the definition or the effective timestamp of the outrigger. If you choose this design strategy, analyze your data and requirements carefully so you are clear on which join logic you want to use. Now, suppose your continuing analysis uncovers the expectation that there will be millions of specimen rows in the dimension and that there are already several columns in the Definition table that will now be NULL in all the Specimen rows because they are needed only in a Person definition. You might decide to move this Person-specific data into its own outrigger as depicted in Figure 15.3.

It is precisely this form of slippery slope with outriggers that frightens me. I'm always afraid that outriggers will turn a star schema back into a normalized relational database, and that I'll lose the user-centered easy access that the star schema and dimensional modeling are built around. I discourage their use in data warehouse design. No matter how many outriggers you define, you'll always

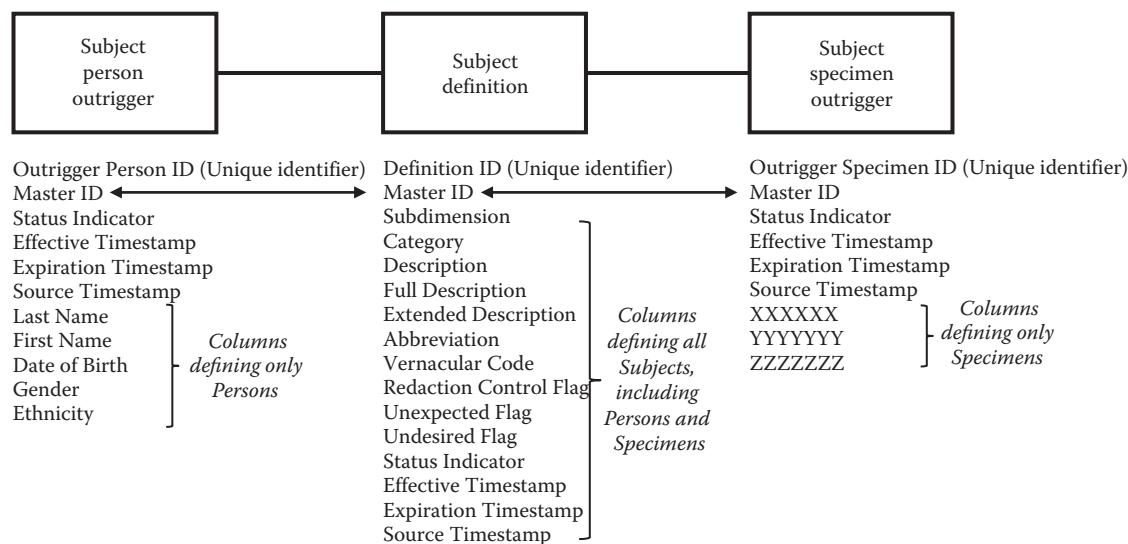


Figure 15.3 Subject Definition with Person and Specimen Outriggers.

be left with a view that recombines them for queries, and that view will embed all of the problems you thought you were solving by moving toward outrigger designs. Database managers can handle NULL values well, so building outriggers to avoid NULL values seems outdated to me. If an outrigger is justified by volatility, particularly volatility of large unstructured columns, those columns belong in the fact table, not in an outrigger in the dimension. Generally, I've abandoned ever using outrigger tables because they simply don't solve the problems that they were once good at solving, largely because the database management systems have already solved those problems for us in recent years. Outriggers are a historical legacy of database designers trying to outsmart the database. They just don't work like they used to, so I strongly recommend not using them.

If you choose to pursue outriggers despite my warnings, you'll be able to do so without any damage to the generic architecture of the warehouse. All database targeting is through a Target Table in the metadata, so you'll have no trouble specifying outrigger tables there and allowing the generic ETL architecture to place the data there. It basically involves adding a decision point to the ETL Definition pipe, at roughly any point in the logic that would target a Definition table, to branch the processing based on the value Target Table. New outrigger tables will then generally require a coding modification at those conditional points to add logic for any new table. Other than that extra maintenance requirement, outriggers pose no threat to the generic architecture developed during the Beta version.

Fact Tables

I've saved the biggest design controversy for last: the number and makeup of Fact tables. The design so far has a single Fact table containing a single value column. Somewhere along the way through the Alpha and Beta versions, a data analyst or DBA has invariably questioned that design choice, perhaps even insisting that the warehouse needs to be changed to contain multiple fact tables. The reasons vary, but they all eventually come down to performance concerns. The notion that a single Fact table with a single value column can be used to store all of the facts in a data warehouse seems improbable to many people, including some very experienced data warehouse specialists.

The alternatives discussed always include whether to allow for multiple fact tables or whether to allow some of those fact tables to have more than one value column. My objection to these alternatives has nothing to do with performance. It has everything to do with usability. Putting everything in the same Fact value column makes it very easy for users to find the data when they don't know what to look for. If you don't know what you are looking for, it's hard to know where to look when the data are spread out across numerous, sometimes hundreds, of fact tables. For facts measured in U.S. dollars, what's the largest amount in any given calendar month? What is the standard deviation of all facts measured in degrees Celsius? How many textual facts include one or more phrases from the

Human Disease Ontology? Queries like these are trivial to implement against a single Fact table with a single value column, but they require expert skill to implement against many traditional multi-fact-table warehouse models. My bias is to support usability and only sacrifice or compromise usability when the performance risks become severe. Since database management systems have gotten pretty good in recent years, that threshold is almost never reached.

The design we've been implementing does support multiple fact tables, as well as fact tables with multiple columns. You didn't use any of those features in the Alpha or Beta versions because they would have slowed you down and you hadn't gained the experience with the generic ETL architecture to effectively implement alternative models. The features are implicit in your implementation at this point, so you can complete that coding now if needed. Your ETL code already places data according to the values of the Target Table and Target Column in the Metadata dimension. By convention, you populated that metadata with the same values for the one Fact table and its one value column. You can drop that convention and implement other Fact table configurations, as long as you abide by the basic dimensional constraints of the overall warehouse design. Those constraints require that the dimensionality of all value columns in a Fact table be the same. This isn't a concern for single-valued Fact tables, but it starts to impact you if you try to widen any fact tables to include multiple values. For the functional dimensions, this isn't usually a problem, but it can become a problem in the control dimensions. In particular, the value columns in a single row of a Fact table must be in the same unit of measure and data state.

The unit of measure constraint poses the biggest challenge for multicollected facts. In a vital signs fact table containing systolic blood pressure, diastolic blood pressure, oral temperature, tympanic temperature, respiration, O₂ saturation, and FLACC pain score, you immediately run into the problem that there are multiple units of measure among your fact values. You might think they could be stored as a single fact row, but they would actually be split into multiple rows based on their units of measure. The fact row with blood pressure readings would have two values, and the rest of the value columns would have to be NULL. Even between the two blood pressure readings, it's possible that one of those values arrived as an invalid numeric value and the ETL placed it into the Unexpected Text UOM. As a result, even the two blood pressure readings would be in different rows.

Data State also poses problems for multivalued fact rows. Suppose you loaded a two-column fact row with Systolic and Diastolic Blood Pressure readings. They are loaded in an active state. A subsequent ETL source loads a new fact that includes only a Diastolic Blood Pressure reading. If the metadata for the new fact includes Supersede control for older values of the fact, there will be a problem, because the older reading can't be marked as Superseded without impacting the data state of the Systolic reading that hasn't been updated. That older fact has to be split into two facts, each with one of the original blood pressure readings so the older Diastolic reading can be marked as Superseded.

The net effect of all these is that, as units of measure and data state processing become more diverse and complex, you end up with fact rows that each contain a single value column no matter how many columns you design into the fact table. One column contains a value, and all the others end up as NULL. You have a single-valued fact table again, except that you now require users to figure out which column contains the data they actually want. Typically, these scenarios functionally bring you back to the single-valued fact table that you already implemented in the Beta version.

Among the arguments I hear for multicolumned fact tables, there is one that is distinct from the others. Some analysts and DBAs express concern about the value column in the fact table being character based. Many users want to be able to query or filter on specific columns that match their data type of interest, and whether or not such a move improves performance depends on the capabilities of your database management system, and the way it automatically casts data from one data type to another. The argument is that having the data already placed in appropriate data-typed columns in the database would improve operational performance compared to having to recast the data at query time.

To implement the desired multicolored functionality, you add the desired new columns to the Fact table, as shown to the left in Figure 15.4. All sourced data and metadata continue to target the value column in the ETL. Within the ETL Fact pipe, just before the insert into the database, add logic for each new column to check the validity of the data type in the value column in order to populate the new column if the type matches. If the Value received is compatible, the type-specific column ends up with a value in it. A numeric value always populates the Float column, but only populates the Integer column if the value contains no fractional component. The Date, Timestamp, and other columns only populate if valid data are in the value column. This kind of multivalued fact row doesn't present any problems with units of measure or data state because each value column is actually a variation of the same fact, so no dimensional conflicts can arise.

An alternative to making the standard Fact table wider is to implement the new type-specific columns as a multicolumned outrigger table (right side of Figure 15.4). For biomedical data, this approach would serve as an improvement because the vast majority of facts will be text, in which case all of the extra

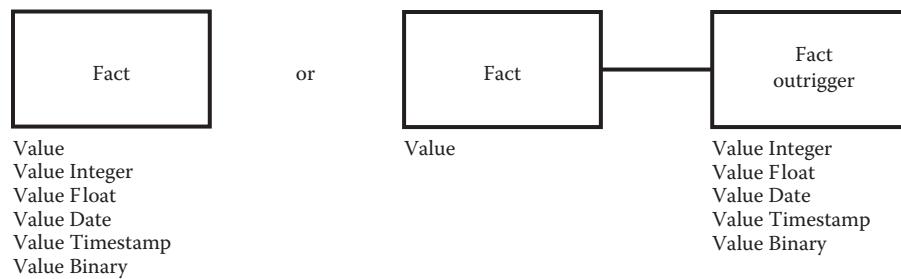


Figure 15.4 Multicolumn generic Fact table alternatives.

columns will be NULL. Using an outrigger, the wider variation needs to be stored only if at least one of the columns is non-NULL. In either of these alternatives, I still consider the Fact table to be single valued. Both of these variations are differences in the physical implementation, not the logical model. Altering your approach to physical implementation concerns me far less than adding new logical fact tables and columns to my general design model.

Since users typically access the warehouse through a semantic layer, it shouldn't matter to them whether the object they pull for Value Float is a direct pull of that column or a CAST to Float from the value column. If you choose to go down this path, ensure that you consider what your users will see in their queries when they pull a fact that failed a type test. For example, if they query the Value Date column, it will be NULL if the Value received was an invalid date (e.g., "2005-02-31"). If they query what they think should be a Value Integer lab result, but the sourced value failed the numeric integer-type check (e.g., "<5"), they won't see that value. Biomedical data violate many of the rules that we information technology support staff would like to be able to enforce, and that we might have been able to enforce at some point in the past. We need to be cautious about making choices in this area that we think will improve performance but end up doing so by sacrificing usability, or the correctness, of the data, queries, and warehouse.

Although I argue against the indiscriminant use of multivalued or wide fact tables, there will certainly be situations where they might be needed, and in which the necessary constraints will be met. Chapter 16 includes an example of a wide fact table that you'll use to store descriptive statistics for quantitative values in the primary Fact table. That table will include different columns for mean, median, minimum, maximum, range, and standard deviation that will always be in the same unit of measure and data state. To implement that table, or any other legitimate multivalued fact table that meets the dimensional constraints, you have to process metadata differently than you've been doing up to this point. Up until now, the metadata you've used to define facts have represented both the row of your fact table and the column in which the value is placed (Figure 15.5a). As you shift toward wider fact tables, the synchronicity of row and column is lost.

A wide multivalued fact table requires different metadata for its rows versus its columns (Figure 15.5b). The metadata entries for the columns are just as they were in prior versions except that the values for Target Table and Target Column will now vary from their standard Alpha–Beta values. You'll add rows of metadata to the Definition table for each sourced version of your multivalued row in the Fact table. [Figure 15.5](#) illustrates a wide fact table that is still very generic. Any five values can be mapped into those five value columns, so there will be a separate metadata row definition for each permutation of different sourced columns that end up in a row. Even if the new fact table included very functionally specific value columns (e.g., systolic and diastolic blood pressure), there still need to be a new metadata row for each combination of sourced

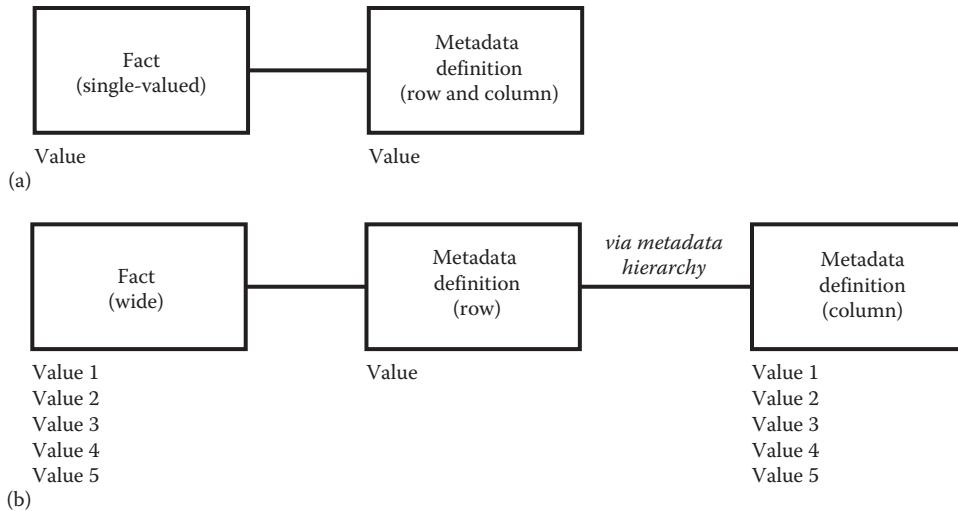


Figure 15.5 Metadata for (a) single-valued fact tables versus (b) multivalued fact tables.

columns that might supply those two values for loading. The new row metadata exist only in the Metadata Definition table. Data sources provide column values, not row instances, so that the Metadata dimension will not include Reference data for these new metadata rows. The Metadata Hierarchy table is used to group the source column metadata under the shared row metadata as a flat hierarchy.

With the new row metadata added to the dimension, the next step is to change the metadata transformation query in the ETL Fact pipe. The JOIN to the Metadata dimension in that transformation is based on the source mappings, so it represents the column metadata. You need to add an additional LEFT JOIN to that query, through the Metadata hierarchy, to obtain the appropriate row metadata identifier. Use a LEFT JOIN because most of the data map to a single-column fact table. You will COALESCE the column metadata identifier to use as the row identifier for single-valued facts.

```

SELECT ...
    R.MASTER_ID                               AS COLUMN_METADATA_ID,
    COALESCE(H.ANCESTOR_MASTER_ID, R.MASTER_ID) AS ROW_METADATA_ID
    ...
    ...
    LEFT JOIN METADATA_H H
    ON H.DESCENDENT_MASTER_ID = R.MASTER_ID AND PERSPECTIVE = '~Fact~'
  
```

The new Row Metadata ID is the value that the ETL Fact pipe will use to create the Metadata dimension foreign key in any inserted fact rows. For single-valued fact tables (e.g., those where the Row and Column Metadata IDs are the same), no other coding changes are needed in the ETL. For multivalued fact tables, the data require one additional pivot to be performed to bring all the values to the same row before that data can be joined to the reference group and bridge data for the creation and insertion of new facts.

Having multiple fact tables, and having some of those tables multivalued, doesn't violate any aspect of this design architecture. I strongly oppose those options during the Alpha and Beta versions because, in my experience, the vast majority of decisions to follow those paths have been mistakes. As you gain experience with this design model, particularly on the user query side, you'll develop a sense for when and where moving away from the one-fact value column implementation might add value. The fact data-type variation I described earlier and the metric model in the next chapter are the only two exceptions I've made in my 9 years of working with biomedical data.

Redaction Control Settings

There are two ways that data values can be excluded from a deidentified view: (a) at the row level, when the Redaction Control Flag in the Definition table is on, or (b) at the column level, when the Redaction Control Flag in at least one Metadata definition row targeting the subdimension in which the data reside is on. The Redaction Control Flag, in all the dimensions except Metadata, controls row level access, while the Redaction Control Flag on the Metadata dimension controls access at the column level. To implement this control, you must add the Redaction Control Flag to the list of source fields that can be pivoted and loaded through the ETL Definition pipe.

A fundamental issue that must be resolved by each organization implementing a data warehouse is the balance between automated and procedural privacy controls. Automated controls include capabilities built into the warehouse to prevent noncompliant privacy-related behaviors from being exercised: If users *can't* look at certain data, privacy *can't* be violated. Procedural controls include the policies and training put in place to ensure that individuals accessing the warehouse follow privacy-related rules when performing their queries consistent with all privacy rules and regulations: If users *don't* look at a certain data, privacy *won't* be violated.

Achieving an effective balance between these two types of controls can be difficult. If only deidentified access to the warehouse is provided, privacy violations become extremely unlikely, being enabled only by loopholes in what can be automated. If, instead, completely identified full access is provided, the organization becomes totally dependent on procedural controls. The motivations and intentions of the warehouse users become the determinants as to whether privacy is violated. We're looking for the most appropriate balance between these two extremes. Insight on this issue can be gained by looking around the institution at other large-scale information systems. Those systems already implement a certain level of balance between automated and procedural controls.

Your electronic health record system probably provides automated controls to ensure that staff isn't looking at health records for patients not admitted or assigned to their area of care. That system probably doesn't prevent them from

looking at the most general facts within those records, even though some of those facts would not be relevant to the care currently being provided or considered. It is the procedural controls that prevent caregivers from simply wandering around patients' information once they have access to those patients in the systems.

The inclusion of the Redaction Control Flag in each dimension of the warehouse design provides a heuristic for defining privacy controls in the data warehouse: Do what you can to lock down data that should not be viewed under privacy regulations, particularly at the level of access to individual patients, and rely fairly heavily on procedural controls to prevent access that either can't be clearly anticipated or that is excessively burdensome to try to prevent in some form of automated way. These controls are likely to be more effective than the various controls already in other health information systems, like the EHR, so they should be considered appropriate by the organization's privacy controlling body. The warehouse team will need to work with that staff to ensure they are comfortable with, and ultimately approve of, the controls put in place.

To implement the privacy controls in the warehouse, a policy decision is needed: Whether to simply exclude data from queries that present privacy issues or include the data and mask those portions that might be sensitive. The distinction is important to the extent that knowing a patient has some sensitive data might itself be sensitive. Suppose a query is listing procedures that a patient has undergone in the last 3 months. Some of those procedures might be protected data (e.g., HIV-related or psychology procedures) with Redaction Control Flags turned on. Should you include the names of those procedures in the result set? You could redact information about those procedures, but that would mean the result set would include an indication that the patient had a sensitive procedure, something that is often considered protected under privacy regulations. Alternatively, you could prevent those protected procedures from being included in the result set. That is easy to do, but might cause the query to no longer make sense because the context of the other procedures for that patient is now obscured since the existence of those procedures is not indicated in the result set from the query. A knowledgeable user might even infer the existence of protected data from their absence, making the omission of the data no better than the redacting of the sensitive procedure names. For this reason, I recommend redaction as the level of desirable automated control, and I rely on procedural controls to limit the impact of any inferences a user might be able to make from any redacted data presented in a result set. Whether you take my recommendation or implement more restrictive controls is an issue that needs to be decided by your emerging data governance group.

A redaction-based control requires a series of changes to most queries that access data in any way that might connect the results of the query to individual patients. Start with a basic query to which a connection to a patient could be added later.

```
SELECT DESCRIPTION FROM PROCEDURE_D;
```

The logic to ultimately join these results to patients is omitted here, as it would be in the definition of this kind of column-based access in a semantic query or reporting layer, in order to illustrate just the logic of redaction. The values returned by this query need to reflect whether the diagnosis entry in the dimension has been flagged for redaction.

```
SELECT CASE WHEN REDACTION_CONTROL_FLAG = 'Y'
            THEN '~Redacted~'
            ELSE DESCRIPTION
        END AS DESCRIPTION
  FROM PROCEDURE_D;
```

If the Redaction Control Flag is turned on in a Definition row, the values in that row will not appear in query results. For Definition entries where the Redaction Control Flag is off, individual columns might still be identified as privacy protected through the metadata that would have been used to load those columns. The query should redact the column value if there was any metadata targeting the column in the same subdimension with its Redaction Control Flag turned on.

```
SELECT CASE
      WHEN PD.REDACTION_CONTROL_FLAG = 'Y'
      OR REDACTION_PROCEDURE_DESCRIPTION_FLAG = 'Y'
      THEN '~Redacted~'
      ELSE DESCRIPTION
    END AS DESCRIPTION
  FROM PROCEDURE_D PD
  LEFT JOIN
    (SELECT DISTINCT REDACTION_CONTROL_FLAG AS REDACTION_PROCEDURE_DESCRIPTION_FLAG
     FROM METADATA_D M ON (M.TARGET_DIMENSION = 'Procedure'
                           AND M.TARGET_SUBDIMENSION = PD.SUBDIMENSION
                           AND M.TARGET_COLUMN = 'DESCRIPTION'
                           AND M.REDACTION_CONTROL_FLAG = 'Y')) ;
```

The additional `LEFT JOIN` in the query produces a single-valued result that is either “Y” or NULL, and the name of the result column has been customized to the specific dimension and column being controlled. In this manner, a query that selected multiple Definition columns would have a series of these `LEFT JOINS` added to provide the appropriate metadata-based Redaction Control Flags for each column being selected. The selection of the column is always standardized to evaluate the Redaction Control Flag settings for both the dimension row and metadata column definitions, returning the “~Redacted~” value if either is turned on.

The implication of this choice can get complicated as more and more data are added to any particular query. Suppose the query includes both a procedure name and the name of the facility in which the procedure was done. Some procedures might be redacted because they are protected, as

mentioned earlier, and some facilities might be redacted because they are protected (e.g., certain research settings or clinics). This can result in a very varied or intermittent display of query results in certain situations, although in my experience, this is very rare. The vast majority of queries don't involve enough protected data that the choice made here becomes significant to the overall results. Users who query a lot of protected data are typically working in a context where they will have, or will obtain, IRB approval to see the data in question. I find that simple redaction works well to meet the privacy needs of most queries, and when users complain about too much redacted data, I send them to the IRB. If users need to see nonredacted data for a group of patients, they should need to obtain IRB approval.

These privacy redaction controls can also be implemented in the warehouse database as table views. For queries selecting many dimension columns, the set of extra LEFT JOINS to establish column-based control can be very cumbersome. Depending upon your technical environment, they can also impact the performance of your queries. Building the extra joins into a table view can simplify the query code that users have to provide and can improve performance if those views are materialized. The table view contains all of the columns of the original table, plus the column-specific Redaction Control Flags for those columns. Using the view eliminates the need to add the LEFT JOIN constructs to the query for each column being accessed. However, it still requires the actual redaction logic to be included in a CASE statement for each column being selected. If you want to eliminate even that extra complexity, you can create an additional table view that includes those CASE statements in the selection of the columns of the original table, and you don't need to also include the column-specific control flags. This option greatly simplifies your query code but at a cost: You can no longer filter your queries on any values that might have been redacted by the view. If you want your filters to see the unredacted values, you need to control the selection CASE statements yourself. I find both extraviews very useful, using them as needed based on the queries that I am writing. Applying a meaningful convention for naming the views (e.g., PROCEDURE_D, FLAGGED_PROCEDURE_D, and REDACTED_PROCEDURE_D) makes switching among the original table and the two views in different circumstances very easy and self-documenting.

Data Monitoring

As your data warehouse implementation matures, you'll begin to encounter areas of concern that require you to add capabilities to your warehouse system data and code, as well as integrate the decision-making process from your governance group into your support strategy and procedures. To do this, you should add

a set of data-monitoring tools to the warehouse that will allow you to generate hypotheses about the quality and management of data, which you'll send to your data governance group and data owners for confirmation and resolutions. Some of those resolutions will require you to provide support in updating data and metadata parameters to accommodate decisions made or approved by the data governance group.

Pipeline Counts

One form of monitoring that you'll do continuously involves knowing how much data are flowing through your ETL pipes over time. You need to be able to spot shifts in data trends that might signal changes taking place directly within the data sources or in the processes supported by the systems that provide those data sources. Most of the counts you'll be interested in are already being generated into the Datafeed dimension by the Beta version ETL. Depending upon the grain you selected for implementing data controls in that dimension, you'll have access to control counts at all of the points in the ETL workflows where you choose to implement the control point-counting pattern. As you begin monitoring that data on a regular basis, it will be very common for you to go back and add control count points in the ETL pipes anywhere you want to gain increased visibility.

System Rows

Every dimension of the warehouse has a System subdimension with six Definition rows that provide for different functions in the ETL loading of Facts. Two of those six rows represent some form of error condition that you'll need to monitor.

1. *Unknown*. Facts that are dimensionalized to the Unknown row in one or more of the dimensions had an error in a natural key that was received from the source system. The Unknown row is assigned any time that any component of a multipart key is either not received or is NULL when received from a source. A list of the actual values that triggered the Unknown assignments can be queried from the audit values in the Fact table.
2. *Ambiguous*. Facts that are dimensionalized to the Ambiguous row in one or more dimensions encountered problems during the ETL when trying to resolve a source entry that was received as Undetermined. As a result of the final ETL logic defined in Chapter 14, those Undetermined entries should have been converted to their functionally appropriate values. If they couldn't be automatically assigned by that logic, the Ambiguous entry was reassigned to the fact.

Nothing in the generic ETL processing will correct these Unknown and Ambiguous entries. They need to be counted, reported, and tracked through the data governance process for corrective action.

Unexpected and Undesired Values

Another form of monitoring you'll do fairly continuously involves counting and reporting how often unusual data activity and relationships are created against the warehouse dimensions. To implement this control, you must add the Unexpected Flag and Undesired Flag to the list of source fields that can be pivoted and loaded through the ETL Definition pipe. Both of these columns should be sourced as slowly-changing values so time sensitivity can be included in your monitoring and reporting. You'll use these two columns, along with each dimension's Status Indicator, to monitor transactions and loads being carried out in the ETL.

Your monitoring queries are looking at facts that might be of interest to data owners or the data governance group. By joining the fact table to each dimension, you can report any facts that exhibit one of the three conditions of interest: (a) the Unexpected Flag is on, (b) the Undesired Flag is on, or (c) the Status Indicator is Deleted. The general monitoring query is typically designed to count the conditions against the master data.

```
SELECT D.MASTER_ID,
       D.FULL_DESCRIPTION,
       D.UNEXPECTED_FLAG,
       D.UNDESIRED_FLAG,
       D.STATUS_INDICATOR,
       COUNT(*)
  FROM FACT F
 INNER JOIN PROCEDURE_B B ON (F.PROCEDURE_GROUP_ID = B.GROUP_ID)
 INNER JOIN PROCEDURE_D D ON (B.MASTER_ID = D.MASTER_ID
                               AND F.SOURCE_TIMESTAMP BETWEEN
                               D.EFFECTIVE_TIMESTAMP AND D.EXPIRATION_TIMESTAMP)
 WHERE STATUS_INDICATOR = 'D'
   OR UNEXPECTED_FLAG = 'Y'
   OR UNDESIRED_FLAG = 'Y'
 GROUP BY D.MASTER_ID,
          D.FULL_DESCRIPTION,
          D.UNEXPECTED_FLAG,
          D.UNDESIRED_FLAG,
          D.STATUS_INDICATOR;
```

The query must be coded to use time-sensitive logic to join each fact to the dimension rows that were current at the time of the fact. You don't want to erroneously report that a fact is assigned to an unexpected, undesired, or deleted dimension row unless that characteristic was true at the time of the fact. These flags tend to get turned on in a certain typical sequence: a row becomes unexpected, and then later becomes undesired, and eventually becomes deleted.

These actions don't necessarily affect the validity of facts already stored as a result of these changes. It is new facts arriving after these flags have been updated that are usually of greatest interest in these monitoring queries.

The Unexpected and Undesired Flags are data governance tools. Their values don't affect what can or will be loaded into the data warehouse. The choice to turn on one of these flags for any particular definition row is always made by either the data governance group directly or by a data owner delegated by the data governance group to perform that function. If your source data analyst identifies a column in a source dataset that might imply a correct value for one of these flags, you should check with the data governance group on its acceptability for data control before using that source column to set the flag. It can seem more efficient to have a source system maintaining these values, but if the source system's definition of unexpected or undesired is ultimately different than the way your data governance group defines those concepts, the value of your monitoring will be negatively impacted.

While some of the conditions reported using these control queries might result in corrective action being taken, in either the fact sources themselves or through some form of manual intervention, these actions are extremely rare. Most unexpected or undesired connections remain in the warehouse indefinitely, so your monitoring queries will likely need to be filtered to include only cases of interest to your data governance group at any given time. A common filter is to only report the exceptions that have taken place in the last 6 months or year. Exceptions older than that are of less interest to your data governance group, although even they sometimes will ask for longer trend results. Another common filter is to look at individual source systems or transaction types of interest, particularly if you have many sources or transactions originating outside your enterprise from which you might have expected lower data quality levels. You should work with your data governance team to establish the levels, frequency, and details of controls that they find most useful in performing their function.

While the Unexpected and Undesired Flags represent generic data functionality for the purpose of data control, you can also add your own monitoring conditions to your queries as needed, typically on a dimension-by-dimension basis. For example, when checking values against the Subject dimension, you can also add a test to see if the Patient was deceased at the effective time of the fact (or at some elapsed time after the fact to allow for some transactional lag for recently deceased patients). You can do the same thing for your encounters in the Interaction dimension. While a recently discharged encounter might continue to receive facts for a time period measured in months, it is very unusual for those facts to continue to arrive over a period of years. Those kinds of exceptions are what the Unexpected and Undesired Flags are intended to help you control, and if you can find an algorithmic way to detect certain occurrences without the need to actually set those flags under those conditions, your effort to improve data quality will be more effective.

While these basic controls are implemented at the Definition level, you should also implement more involved or sophisticated checks as your data governance function matures to take advantage of them. An example could involve querying against any dimension that includes a gender-specificity property, such as diagnosis, procedure, or pathology. If you include the Subject dimension while checking the unexpected conditions for these dimensions, you can also flag instances where a fact connects a Patient of the wrong gender to an entry in the dimension. The controls that can be introduced in this manner are only limited by the availability of source data to establish the needed relationships and the imagination of your analysts and governance staff to define the desired controls.

Orphan Tracking

Your warehouse support team, working with any assigned data owners, will need to keep track of dimensional orphans created as a result of routine fact loading. A few orphans here and there are not a problem for a data warehouse. It's not uncommon for source systems to mishandle a few keys now and then, resulting in orphans being loaded into various dimensions at times, but it is important to ensure a control mechanism is in place that can be used by the data governance functions to monitor and control these occurrences.

Many orphans are routinely *expected* and are not surprising. In most healthcare settings, procedures are in place to ensure people can do their jobs as soon as they are needed. This sometimes results in new entities being created in transactional systems before the corresponding entities are updated in their source master system. For example, when a new drug is being deployed in the provision of care, one of the first places the drug is defined is in the CPOE system so providers can order it. The maintenance to properly define the drug in the pharmacy system might be done after it is added to the CPOE system. The pharmacists know how to dispense the drug if it shows up in an order, but the paperwork for setting up the drug in the system can sometimes wait a few days. As a result, you get medication orders into the warehouse for medications that aren't yet defined in your Material dimension. You should create an orphan, so when the Pharmacy system sends you the dimension load data for that drug, your ETL will autoadopt the orphan in the dimension. The result is a valid set of ordering and administration facts connected to a valid and up-to-date master definition in the dimension.

The problem with orphan definition is that your master data lack what might be key properties in the definition because they are unknown by the Fact transactions that instigate the orphan creation. In this new drug example, you don't know the therapeutic class of the orphan drug, so any queries made by therapeutic class won't correctly aggregate facts related to this drug.

Another common occurrence in many healthcare settings involves provider credentialing. New providers hired by the organization are made productive quickly by defining them and their roles to the EHR and other ancillary systems.

This means your warehouse is very likely to receive new transactional facts for providers who haven't yet been fully vetted in the institution's credentialing system, so they won't be known to your Caregiver dimension. You should create orphans for these providers to support the incoming facts until the credentialing source provides the data needed for autoadoption. As with any orphan, you lack key details that would typically be available in the master data for these providers: You don't know their names, nor do you know their clinical specialties. You can't differentiate the new surgeon hired last week from the 30 new residents that started at the medical school last month until the credentialing system sends you the appropriate master data for the autoadoption.

If the time gaps between orphan creation and autoadoption events are small, or the situations themselves are rare, they might not be of major concern. One major concern is the ability to identify actual errors among the orphans: keys that are wrong or broken as opposed to valid keys caught in the different cycles of data maintenance. There might be some orphans that will not autoadopt because they are actually wrong, and sometimes it can be difficult to differentiate them from the rest until enough time has passed that all autoadoptions are likely to have been completed. What remains are the orphans that represent potential errors to be addressed. To help control for these errors, you must be able to measure your orphan population in order to assess risk and take action if warranted. Some of the questions your control queries should be able to answer include

- *How many orphans do we have by subdimension?* There will be a pattern that develops over time for each type of entry, but you should pay particular attention to any sudden spikes in orphan numbers in any particular dimension or subdimension.
- *How old are these orphans?* In situations where many types of orphans routinely autoadopt, you don't want to focus too much attention on the freshest ones. You should pay more attention to orphans that are getting stale or that seem to be aging beyond the norm.
- *How many facts use these orphans?* An orphan that was created against one obscure fact should receive different scrutiny than one that is loading hundreds or thousands of facts per day.
- *What kind of facts are the orphans referenced by?* Some facts are considered more critical than others, with operational details often ranking fairly low. For example, an orphan medication on an Order Cancelation isn't as serious as one on an Order Administration.
- *What roles do the orphans play in those facts?* How a dimension is referenced on a fact matters in terms of any risk associated with an orphan. For example, an orphan caregiver in a Consulting role is far less serious than in an Attending role.

Orphans that don't autoadopt over a reasonable period of time become stale and end up as issues to be investigated by data governance. The data warehouse

support team should develop queries that aid the investigation activities that are assigned to data owners by the data governance group. Working together, these stakeholder groups will eventually identify the causes of many of the orphans and use different surrogate alignment techniques to correct or eliminate orphan conditions.

There isn't a single source of nonadopting orphans, but there are patterns that emerge over time. One of the most common patterns is front-end or back-end truncation: the loss of one or more characters from the front or back of an existing natural or system key. I've seen this happen a lot with numeric identifiers that arrive through transmitted ASCII files that are not handled correctly at some point in their transmission, storage, or processing. To assist the investigation of these nonadopting orphans, perhaps the warehouse support team can build queries to help with diagnose of these truncation situations.

The simplest type of truncation occurs when an identifier with an unexpected number of digits is loaded into the dimension. For example, imagine a dimension context where a 10-digit number is expected as the source key. All of the entries in the Reference table for that context should be 10-digit numbers. However, a series of 9-digit numbers are present in the dimension, all identifying rows that are orphans in the dimension. If you suspect that these orphans resulted from a leading zero being dropped from the key, you can confirm this possibility with a single simple query.

```
SELECT OR.CONTEXT_KEY_01      AS ORPHAN_IDENTIFIER,
       OR.MASTER_ID        AS ORPHAN_MASTER_ID,
       CR.CONTEXT_KEY_01    AS CANDIDATE_IDENTIFIER,
       CR.MASTER_ID        AS CANDIDATE_MASTER_ID
  FROM DIAGNOSIS_D D
 INNER JOIN DIAGNOSIS_R OR ON (D.MASTER_ID = R.MASTER_ID)
 INNER JOIN DIAGNOSIS_R CR ON (CR.CONTEXT_KEY_01 = CONCAT('0',OR.CONTEXT_KEY_01))
 WHERE D.ORPHAN_FLAG='Y';
```

Each row returned by this query identifies an orphan row in the dimension, the identifying key of which matches another key in the dimension that has an extra leading zero. If the data owners or data governance group confirm that these two entries in the dimension are the same, then the Master IDs can be merged as a surrogate alignment transaction. However, before this decision can be made, the risk of making incorrect choices must be addressed. What if the orphan identifier matched other keys in other ways? To be sure of each surrogate alignment action, it's important to check a variety of possibilities that might provide alternative explanations for any orphans that are becoming stale. Instead of only checking for an exact match with a leading zero added to the key, you should check to see if the key of the orphan is a subset of any other keys.

```
SELECT OR.CONTEXT_KEY_01      AS ORPHAN_IDENTIFIER,
       OR.MASTER_ID        AS ORPHAN_MASTER_ID,
       CR.CONTEXT_KEY_01    AS CANDIDATE_IDENTIFIER,
       CR.MASTER_ID        AS CANDIDATE_MASTER_ID
```

```

FROM DIAGNOSIS_D D
INNER JOIN DIAGNOSIS_R OR ON (D.MASTER_ID = R.MASTER_ID)
INNER JOIN DIAGNOSIS_R CR ON (CR.CONTEXT_KEY_01 LIKE CONCAT('%',OR.CONTEXT_KEY_01,'%'))
WHERE D.ORPHAN_FLAG='Y';

```

This query might return multiple candidates against a single orphan, resulting in a more thorough investigation before committing to any data adjustments. If only one existing key can be identified as a candidate match, there's usually a good chance that the orphan entry was intended to be that candidate. If multiple candidates are found, greater care must be exercised. If more investigation is needed, I continue my investigation of orphans by looking at the valid dimension entries that are *typically* associated with the type of fact that created the orphan. By anchoring the query on the original fact against the orphan, I profile the dimension entries used on those facts in the same role.

```

SELECT OR.CONTEXT_ID AS CONTEXT_ID,
       OR.REFERENCE_COMPOSITE AS ORPHAN_IDENTIFIER,
       OR.MASTER_ID AS ORPHAN_MASTER_ID,
       CR.REFERENCE_COMPOSITE AS CORRELATED_IDENTIFIER,
       CR.MASTER_ID AS CORRELATED_MASTER_ID,
       CD.DESCRIPTION AS CORRELATED_DESCRIPTION,
       COUNT(*) AS COUNT_OF_ORPHANS
  FROM FACT OF
INNER JOIN DIAGNOSIS_B OB ON (OF.DIAGNOSIS_GROUP_ID = OB.GROUP_ID)
INNER JOIN DIAGNOSIS_D OD ON (OB.MASTER_ID = OD.MASTER_ID AND OD.ORPHAN_FLAG = 'Y')
INNER JOIN DIAGNOSIS_R OR ON (OD.MASTER_ID = OR.MASTER_ID)
INNER JOIN FACT CF ON (OF.METADATA_MASTER_ID = CF.METADATA_MASTER_ID)
INNER JOIN DIAGNOSIS_B CB ON (CF.DIAGNOSIS_GROUP_ID = CB.GROUP_ID
                             AND CB.ROLE = OB.ROLE AND CB.RANK = OB.RANK)
INNER JOIN DIAGNOSIS_D CD ON (CB.MASTER_ID = CD.MASTER_ID
                             AND CD.ORPHAN_FLAG = 'N' AND CD.STATUS_INDICATOR = 'A')
INNER JOIN DIAGNOSIS_R CR ON (CD.MASTER_ID = CR.MASTER_ID
                             AND CR.CONTEXT_ID = OR.CONTEXT_ID)
 GROUP BY
       OR.CONTEXT_ID,
       OR.REFERENCE_COMPOSITE,
       OR.MASTER_ID,
       CR.REFERENCE_COMPOSITE,
       CR.MASTER_ID,
       CD.DESCRIPTION
 ORDER BY
       OR.CONTEXT_ID,
       OR.REFERENCE_COMPOSITE,
       COUNT(*) DESC;

```

This query returns a set of rows for each orphan in the dimension. Each returned row includes an identifier of a nonorphan entry in the same context as the orphan identifier, a description of the nonorphan row, and a count of how many times the nonorphan entry was associated with the same kind of fact as the original orphan-generating fact. The set of entries returned for each orphan can serve as the starting point for investigating possible corrective actions. Only rarely does this query provide enough information to clearly identify the correct dimension entry for the orphan. In most cases, the number of rows returned is too great to draw definitive conclusions. When that happens, I add joins and filters to the query to try to narrow the range of results to a more meaningful list.

Suppose that the orphan being investigated is a drug associated with a medication administration fact. The aforementioned query will return a count for every valid nonorphan medication that has ever appeared on a medication administration fact. The largest counts will be associated with the most commonly used medications, providing no real clues regarding the correct identity of the orphan being investigated. Additional joins and filters can considerably narrow the range of results. You could add joins to the UOM dimension, so the facts being compared would have to be in the same unit of measure to be included in the counts. Such a change would dramatically reduce the rows returned for consideration. You could add joins to the Caregiver dimension, so the facts being compared would have to be associated with the same attending physician on the assumption that the provider who entered the orphan fact probably used the correct identifier on other transactions. If the provider proves too narrow, you could alter your join to require the attending physicians on each fact to be practicing in the same clinical specialty.

Each new join or filter reduces the range of data returned by your matching query, bringing the investigation closer to finding the desired nonorphan entry. If the orphan fact is an older entry that was created during a historical load, it might help to join the query to the Calendar dimension to require that the facts being compared occurred within 6 months of each other, so only medications used during the proximate time period of the orphan entry are included. The different combinations of joins and filters that might improve the query results are too numerous to mention here. The more you know about your data, the more meaningful you can make your queries. The ultimate objective is to work with your data governance team to identify the nonorphan dimension entry that should have been the intended dimensionalization target of the fact that resulted in the creation of the orphan. Once discovered, and confirmed by the data governance team, the orphan can be corrected through a surrogate alignment transaction.

Surrogate Merges

A surrogate merge involves taking two Definition entries in a dimension and consolidating them to identify one single Definition entry. To a user, the result of the merge action will look as though any Reference entries for those Definitions were originally entered into the warehouse as Alias transactions. The common reason for needing to be able to carry out this kind of support transaction is that aliases arriving at the warehouse are often not identified as aliases by their sources. Sometimes, this represents an oversight by the data analyst who provided the metadata for loading that data, but more often it's because the relationship between the two Definition entries was unknown, or even unknowable, at the time the source datasets were analyzed.

Your orphan investigation process will be a common origin for your surrogate merge transactions. Those investigations often result in determining, and confirming through your data governance group, that an orphan Reference and Definition in a dimension should have resolved to another nonorphan Definition entry. In those cases, a surrogate merge is carried out to (a) update the Reference for the orphan entry to instead contain the Master ID of the target nonorphan Definition entry, (b) delete the orphan definition row (physically or logically by updating its Status Indicator), and (c) update any facts pointing to the original orphan Master ID to instead point to the nonorphan Master ID. This third action is complicated by the fact that the Master ID of the orphan is potentially buried in many complex Group structures in the dimension. You can't just update the Bridge tables (and Group Composites in the Group tables) with the revised Master ID because the resulting Groups might already exist in the dimension. If a revised group doesn't already exist, the Bridge and Group tables can be updated with the new target Master ID, and the Facts don't need to be updated. If a target Group does exist, the actual Fact rows should be updated to point to that already existing group, leaving the old Group unused in the dimension. The old Groups that weren't updated can be deleted from the Bridge and Group tables, although they'll be deleted automatically by the next dimension cleanup. They won't be used again by a Fact because there are no Reference entries left in the dimension that would resolve to the old orphan Master ID.

The surrogate merge to resolve a stale orphan entry is a simpler, special case compared to the more general approach to merging surrogates. The orphan case is simple because there was nothing contained in the orphan Definition row that needed to be retained after the merge, and the orphan row couldn't have had multiple Reference entries pointing to it from multiple contexts. In less simple cases, you will be merging two surrogates that have extensive data in the Definition table and multiple Reference entries. For the general case that can handle both simple and complex scenarios, your merge transaction must designate one of the two Master IDs as the target and the other Master ID as the source. Any Reference entry containing the source Master ID needs to be updated to point to the target Master ID. Groups, Bridges, and Fact entries are updated in the same manner as for orphan-based merges. The difference in the general case is that certain column values associated with the source Definition might need to be consolidated into the target Definition. This can become a very complicated transaction if both source and target Definitions exhibit many slowly changed variations. The logic required to merge them can be developed algorithmically, but the situation is rare enough that I recommend allowing the target Definition to be left in its existing state and allowing the source Definitions to be deleted.

I've seen situations where source data columns were only transferred to the target columns if the target columns were NULL going into the Merge. If a true consolidation of the data is needed, I recommend writing a source data extraction query against the source Definitions and reloading that data through

the standard ETL process. This standard process will implement an appropriate slowly-changing cascade of data in whatever manner you define the metadata, without you having to develop special logic just to carry out this rare transaction.

Security Controls

Ultimately, the data warehouse is an information system that is subject to the controls and limitations of the information technology platforms on which it is implemented. Final control of the warehouse, beyond that built directly into the warehouse, is dependent upon the availability and implementation of controls within the organization's information technology arena. Security is an example of a function that can't be managed within the warehouse design: it is dependent upon the security tools and policies already in place for the environment in which it is implemented. Security of the data warehouse is typically defined at three levels:

- *Network*. Users of the warehouse will be presumed to have been authenticated as legitimate users of the organization's information technology capabilities.
- *Database*. Users of the warehouse will be presumed to have been authorized to access the parts of the warehouse to which they request access. Database access might be restricted to only certain portions of the database, and queries performed would be limited accordingly.
- *Application*. Users of the warehouse will be presumed to be authorized to access the application systems needed for warehouse use, particularly the organization's business intelligence toolset. Access might be restricted to only certain capabilities or features of those applications, and queries performed would be limited accordingly.

Defining all of the requirements for each of these controls is beyond our scope here, but implementation is dependent upon these controls being in place.

Certain aspects of these controls might become important to some of the delivery capabilities described in the next chapter. I am not a security expert. I know certain access restrictions can be put in place against tables, rows, or columns at the database or application levels. I do not completely understand how these things are done, but I know I am dependent upon them being done well. I also know these capabilities are a moving target as technologies improve, so I expect the warehouse team will work closely with the security people in the department who can make this aspect of the warehouse much more helpful and successful.

One design question that typically becomes a security concern involves whether a user of the warehouse will always have a single user identification. Typically, I expect that users will have a single network user ID, and that any differentiation of roles and responsibilities take place at the application level. Your business

intelligence toolset probably has a certain amount of user security functionality. The integration of these two layers is important, and I urge you to start addressing this area very early in your initiative. My concern involves knowing whether a user logging into the warehouse should be given only deidentified access to the data, partial identified access, or fully identified access. That question needs a rigorous answer as early in your design cycle as possible. Even within the realm of identified access, the actual data that can be identified could be different for different IRB-approved projects, even for the same user ID. Fully deidentified or fully identified data options are fairly easy to implement either at the network security or semantic layer levels. Partially identified data are much more difficult to implement because it involves IRB approvals that could vary the types and complexity of data constraints that are being approved on a project. If you implement IRB approvals by creating a different view of the warehouse for each IRB-approved project, you need to resolve how you'll interconnect the network security, the database view, and your semantic layer. In my experience, if you can provide a view that limits access to only certain cohorts of patients (typically implemented as a Cohort hierarchy in the Subject dimension), you satisfy many IRB requirements. If you can also restrict date ranges or types of facts, you'll cover the vast majority of the restrictions I've seen. Other views can be defined as needed, but if an IRB-approved project view gets too complicated to handle in your database management system, you would be forced to create a periodically updated physical data mart into which to export that project's data. This last resort should be avoided for as long as possible because the effort required to maintain and support a separate data mart is much greater than required to support a simpler view.

Implementing Dataset Controls

To operationalize data loading, you need a way to keep track of the data that have already been sourced and loaded into the warehouse in order to minimize processing of redundant data. If appropriate controls are not provided, there's a risk that the sourcing jobs will unnecessarily resource data that has already been loaded into the warehouse. While the generic ETL has been specifically designed and implemented to ignore or discard inbound data that already exist in the warehouse, there is a significant risk of wasting a lot of resources if all of the data in some datasets were sourced every day.

For example, consider the SDI job that extracts from your patient master file that you designed and built in the Beta version. It might pull tens of thousands, or even millions, of patients. Those patients are very likely already loaded your Beta version. If you were to rerun that SDI job to pull those patients again and load them to the warehouse, for the most part, nothing would happen. With the exception of patients that have been added or changed since you last extracted them, the data arriving in the ETL streams are the same data as already processed. Keys will resolve properly in the Reference pipe, pass through the

Definition pipe, and be discarded by the update logic because the values and source effective dates are the same as already in the warehouse. This processing isn't erroneous, but it is certainly unnecessary and wasteful.

Of particular interest in this example is any data that have been added or changed since the last source extract. These are the data you want to source and process. To accomplish this without redundant resourcing all of the other data in the source that has already been loaded, the Gamma version implements a control table that explicitly keeps track of the dates for which processing has already been completed for each source physical dataset. The columns of the control table need to include

- *Physical Dataset*. Serving as the key to the row, there will be a row in this table for each Physical Dataset that exists as a Context Key 01 in the metadata dimension.
- *Status Indicator*. Marking a control row as Active or Inactive will determine whether or not it is sourced and processed or not.
- *Control Timestamp*. This timestamp is the latest timestamp that has already been completely processed, and determines the earliest timestamp that will be sourced when an SDI job is scheduled.
- *Maximum Timestamp*. This timestamp determines the latest timestamp that should be sourced or processed. If not NULL, no data changes after this date should be sourced. It is used for performance reasons during history loading to isolate subsets of data.

With the control table populated, an additional JOIN is added to the main extraction query in each SDI job in order to enforce that no data is extracted redundantly.

```
INNER JOIN PHYSICAL_DATASET_CONTROL PDC
    ON PDC.PHYSICAL_DATASET = 'Patients'
    AND PDC.STATUS.INDICATOR = 'A'
    AND PDC.CONTROL_TIMESTAMP < SOURCE_EFFECTIVE_DATE
    AND PDC.MAXIMUM_TIMESTAMP >= SOURCE_EFFECTIVE_DATE;
```

This control will cause no data to be extracted if the control entry has had its status changed to Inactive, or it will only extract data that falls after the Control Timestamp and before the Maximum Timestamp. For everyday operation, the Maximum Timestamps are set to the warehouse maximum expiration date so there is no effective upper limit on extraction dates. However, a Maximum Timestamp is an effective tool for limiting the data to be extracted from new sources from which a lot of history is being extracted. I've used it to allow me to extract, for example, 1 year at a time from large data sources.

To implement this structure, two steps are needed: (a) a process to load and initialize rows in the control table and (b) a process to update the control row after ETL processing. To load the table, join the control table to the Metadata

reference table, inserting a control row for any physical dataset not already found in the control table, with columns initialized for active processing of all dates.

```
INSERT PHYSICAL_DATASET,
       STATUS_INDICATOR,
       CONTROL_TIMESTAMP,
       MAXIMUM_TIMESTAMP
   INTO PHYSICAL_DATASET_CONTROL
SELECT DISTINCT
       META.CONTROL_KEY_01      AS PHYSICAL_DATASET,
       'A'                      AS STATUS_INDICATOR,
       0001-01-01                AS CONTROL_TIMESTAMP,
       9999-12-31                AS MAXIMUM_TIMESTAMP
  FROM METADATA_R META
 LEFT OUTER JOIN PHYSICAL_DATASET_CONTROL PDC
    ON META.CONTROL_KEY_01 = PDC.PHYSICAL_DATASET
 WHERE PDC.PHYSICAL_DATASET IS NULL;
```

Updating the table takes place at the end of ETL processing, updating the Control Timestamp for each Physical Dataset with the maximum source timestamp from the loaded data. This prevents data earlier than that timestamp from being processed again on the next scheduled source extraction.

```
UPDATE PHYSICAL_DATASET_CONTROL PDC
  SET CONTROL_TIMESTAMP = S.MAX_SOURCE_TIMESTAMP
  FROM (
    SELECT PHYSICAL_DATASET,
           MAX(SOURCE_TIMESTAMP) AS MAX_SOURCE_TIMESTAMP
      FROM #SOURCE0
     GROUP BY PHYSICAL_DATASET
  ) S
 WHERE PDC.PHYSICAL_DATASET = S.PHYSICAL_DATASET;
```

These source controls can be used in a variety of ways, and their effectiveness relies on many factors. The main issue to be considered in determining these dataset controls is the efficacy of the choice of the source column being used as the source timestamp for serving as the control parameter for extraction. As part of the analysis of new data sources, the analyst designing the metadata and SDI query needs to select a source for this timestamp. Ideally, the chosen column will be a direct functionally appropriate timeframe within which the source event being modeled actually occurred. If the data source is a contemporaneous system source, new data won't arrive in the source for effective dates in the past, making the functional timestamp an effective tool for controlling data extraction.

Alternatively, for situations where data can arrive late for an effective date already extracted, a different control timestamp is needed. In these cases—and this also includes any case where a functionally appropriate source timestamp

Table 15.1 Example Account Definitions with Functional and Systemic Timestamps

Account	Admission Timestamp	Modified Timestamp
12345A12	2014-07-03 09:14	2014-07-03 12:38
35245B53	2014-07-03 11:16	2014-07-03 13:45

is not defined or available in the source—a more system-oriented timestamp can be used. One example is the traditional timestamp in many database tables indicating when the row was last updated. In fact, this is probably the most common column used to populate the Source Timestamp in any SDI query.

Suppose a couple of admission transactions need to be sourced into the warehouse. Each involves rows in a table that have been updated subsequent to the functional timestamp of the admissions of interest (Table 15.1). We don't know if the modified timestamp in these rows is later than the event time because of delays in the processing of those event transactions themselves or because other updates were made to those accounts after the admissions had been recorded. We actually don't need to know that. We only need to know each row provides us with an admission event that must be recorded. Each transaction will be dimensionalized to the Calendar and Clock dimensions to the appropriate minute on the morning of July 3. Those timestamps will not, and should not, be used in the SDI query as the Source Timestamp for those transactions. What we actually know about the admission timestamps is that they were current as of the Modified Timestamp. It is the Modified Timestamp that should be extracted as the Source Timestamp, and that is the Control Date we should use for controlling the extractions.

Based on that date, if some other property of the Account changes after we extract the data, the Modified Timestamp will be changed, and you will reextract the account on the next extraction. This means that you could reprocess the July 3rd admission even though the admission event hadn't changed because our generic ETL logic won't actually apply any updates to the warehouse if the data haven't changed. If the admission time had changed, our extraction would pull it again, and we'd add a new fact with a new source timestamp. If the admission fact has been defined as one-per-encounter in the metadata, the new fact will automatically supersede the older fact that had been loaded previously.

Warehouse Support Team

The most important data control for your warehouse is your warehouse support team. Just as with the evolution of ETL functions described in Chapter 14, this chapter has described controls that have evolved over time by recognizing

patterns of activity carried out by members of warehouse support teams across my clients. At one time, all of these controls were simply one-off coded routines or update statements that I have run against a particular warehouse for a particular purpose or to correct a recognized problem. It is by reusing these routines in many circumstances that the opportunity to define another generic control in this architecture emerges. There are still many control requirements that need some form of direct intervention by your support team because their circumstances are too rare or too complicated to have yet been designed as a general solution. With all of these controls in place, and a support team prepared to keep data entropy at bay, it's time to build out the data in the warehouse.

Chapter 16

Building out the Data

With the ETL workflows and controls finalized, it's time to load all the desired and scoped data into the warehouse. Somewhere around this time will be the last opportunity that the development team can reinitialize the warehouse. The volume of the data is about to explode, and there will be no going back. During the months your team has been developing and finalizing the generic ETL subsystem, you've also been developing data sourcing workflows, typically numbering in the hundreds. All of that comes together now in the bulk loading of the Gamma version of the warehouse. The choice of what data to load, and in what order, is no longer a technical decision, but one that focuses on making the warehouse meaningful to the early adopting users who will be given access and trained first. The warehouse is now functionally complete.

Minimize Data Seams

Early users of the warehouse will tend to judge what they see by the types and volume of data available and the functionality available for querying that data. Having a large quantity of data available usually isn't a problem because many of the datasets from which you'll draw your historical data contain information going back many years. Where users start to have concerns is in cases where they see seams in the data. Seams are temporal or conceptual points where you've had to stitch together data from multiple physical sources in order to create a continuous stream or timeline, and differences appear in the data that can be traced to differences in the related sources. At their worst, seams in the data can result in orthogonal data that are no longer possible to retrieve using a single query. Different queries are needed before and after the seam. If these queries can be defined as subqueries of a larger common query, it's possible to recombine the orthogonal data to create the

originally intended stream, but the difficulty and complexity of doing so raises serious concerns among your early users.

The most important seam to avoid, and one already discussed in Chapter 3, is the temporal seam between your historical and prospective data sources. If both sources will be derived from the same system, it's usually possible to avoid most seam issues because the same tables or datasets end up serving as input to the different source data intake (SDI) queries that pull the data. The differences in scheduling and access don't end up affecting the way the data are actually sourced, but when the historical and prospective sources are different, the impact can be more serious. The classic case is when a relational table within an application database will serve as your historical source, and an HL7 message through your integration engine will serve as your prospective source. HL7 messages often contain a variety of data that are processed by the receiving applications into many database tables and columns, and some of that data will invariably map to aspects you will not be sourcing as part of your historical extracts. Some of that data might be sourced by other jobs for other purposes, so all the data might end up in your warehouse, just not in a seamless way that users will see as a continuous stream. When this happens, the data are said to be *orthogonal*.

As an example, let's imagine a case of sourcing inpatient admissions into your warehouse. Your historical data source might be the Encounter table in your EHR, while the prospective source might be the HL7 A01 messages you receive from the organization's integration engine. The A01 message includes information identifying the attending provider for the admitted encounter. Upon inspection, you note that the Encounter table for your history doesn't include a provider identifier. Instead, the EHR has a provider assignment table from which you are sourcing all historical provider assignments to encounters, and the attending provider assignment is a subset of that extraction. The problem isn't that the necessary information doesn't make its way into the warehouse. It does. The problem is that a seam appears in the data that your users now have to be aware of and query around.

For example, if a user wants to count admissions over time by the clinical specialty of the attending provider, the query gets more complicated because of the orthogonal data. After the seam, a simple count query can be built because the admission fact is dimensionalized directly to the attending provider. Before the seam, an additional subquery will be needed to establish the attending provider for each encounter through the provider assignment fact. All of the data are present, so the results will be correct; but users shouldn't have to deal with that kind of query complexity in order to answer such a simple question.

There are query design patterns that can be used to implement these kinds of orthogonal queries. The most common strategy is to treat the before-seam and after-seam queries as distinct subqueries under a consolidating query that presents the combined seamless result. This strategy can be taught to nonnovice users, and it works well because it is always done in the same manner. In my

experience, the strategy works well even when there are multiple seams in data for the same user query. An alternative would be to build the resolution of the orthogonal data into a single query, providing the join to the provider assignment fact as an optional left join and then coalescing the provider identifiers from both parts of the query, knowing in advance that only one will be present. This alternative strategy is much more complicated and is much more difficult to teach to even advanced users, although it usually offers some query performance advantages over separate subqueries. Those potential performance advantages have greatly lessened in recent years as database optimizers have gotten better. I find that when I try to outsmart an optimizer, I usually regret the effort. Some of this complexity can be hidden from users by using specialized objects in your semantic query layer; but this requires a much more aggressive effort for the support team to keep the objects in the semantic layer up to date, and it requires extra training to keep users informed of those specialized objects.

The preferred strategy is to avoid sourcing data with these kinds of temporal data seams. Usually, the seam can be corrected by adding some new sourced data to one of the two related source queries. In the case of the historical–prospective data seam, adding data to the historical extraction usually sidesteps the problem. The historical SDI query gets a bit more complex; but the life cycle of that query is fairly short relative to the ongoing set of the prospective SDI loads. This fix won’t always be available because there isn’t always the additional data needed to smooth the seam, but I find it’s usually an option.

While the historical–prospective seam is the one you’ll usually pay the most attention to when planning your sources, it’s actually just a special case of the more general problem of temporal seams. You also have to watch for other smaller-scale historical–historical seams. These seams tend to occur within data sources that have undergone procedural or system changes during the time period for which historical data are being sourced. An example could include an allergy data source that includes an indicator of the severity of the patient reaction to an allergen exposure. The severity property is a routine data element today in allergy-related applications, but it hasn’t always been. There will be some point in time before which that property wasn’t supported by the application from which you source your history data. The database column for that property is most likely a `NULL` value prior to the seam. If the current value of that column can’t be `NULL`, users are unlikely to misinterpret the `NULL` values prior to the seam; but if the current values can include a `NULL` value, it becomes more important to smooth over that seam in your historical load. A common way to do that smoothing is to insert an additional entry in the Severity subdimension of the Quality dimension that can represent a “~Not yet collected~” value and then source the `NULL` values in that property prior to the seam as that value. The investment you make in eliminating temporal seams in your source data, while more work for you, provides your users with a richer data experience that doesn’t require them to have database-aware expertise in order to get answers to their questions.

At the opposite extreme from the historical–prospective seam is the seam that I refer to as the “jagged edge.” The jagged edge occurs at the beginning of your historical source timeline as a result of different source applications having data available starting at different points in time. Your EHR might go back only 7 years, while your lab systems go back 10 years. There’s really no way to avoid the jagged edge. Its effect can be smoothed somewhat, but there will be queries that use data from multiple sources that will only be able to go back to the starting point of the most recently initiated source. Older data are still available going back further; but if a query is dependent upon another source that didn’t exist at the time, you won’t be able to easily meet that need. Generally, users don’t have a problem with the jagged edge. They understand the limitations of loading a warehouse using applications of different ages and maturities. Ensure your user training materials clearly indicate the starting horizon for each major source.

In addition to temporal seams occurring in the data, you’ll also have situations where the way you sourced your data creates a conceptual seam in the way the data are stored from two different sources. Each source was modeled and sourced in a way that made sense to the analysts doing the work, but when the two related sources are compared, they’ve adopted orthogonal strategies. Often these seams arise between two sources that are being analyzed during the same warehouse release, in which case a large number of resolution options are available. At other times, the seam will arise between two sources being sourced across a wide period of time, sometimes years. The most common situation is when a new source is added to the warehouse that includes logical data already being loaded from other sources. The physical organization of the new source might lead the analyst to source and map the data in a way that could conflict with how the data have been sourced in the past. Part of the answer is for the analyst to have a strong familiarity with all the data already in the warehouse in order to avoid creating this situation accidentally, but in situations where the new way of sourcing and mapping the data would really be better than the way it was done in the past, different strategies are required.

An example of a conceptual seam is often found in vital-sign data. Your oldest data sources probably include a table of vital-sign observations that has multiple columns, each representing a different observation. There are oral temperature, systolic blood pressure, and diastolic blood pressure columns, each containing the values that need to be sourced into the warehouse Fact table. This wide-table orientation is very common in older application systems where priority was given to normalizing the database tables. In more recent application systems, a higher priority is given to the flexibility and extensibility of the data in the system, resulting in narrower tables where each row includes only a single observation, and there’s an additional column that indicates what that observation is. Instead of oral temperature, systolic blood pressure, and diastolic blood pressure being separate columns in the table, they become distinct values in the new column across multiple rows. Today, this narrow table strategy is considered a superior

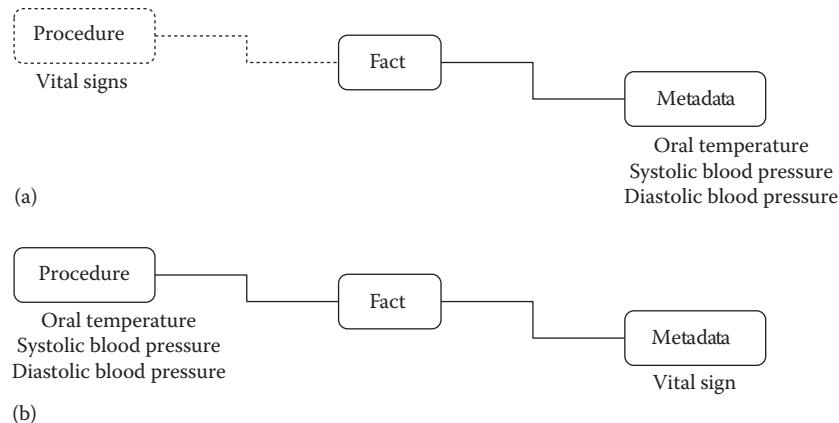


Figure 16.1 Alternative views of multifact sources using (a) metadata wide emphasis or (b) dimension narrow emphasis for differentiation.

database architecture because it allows maximum flexibility for adding additional observation types over time without requiring database changes.

For this example, imagine you're loading an older data source that consists of a vital-sign table that followed the wide-column architecture. That data might exist in the warehouse today as depicted in the top of Figure 16.1. Each observation column in the source table might have been defined as a different fact, resulting in each sourced column having its own metadata entry. The dimensionalization to procedure (depicted as dotted lines in the figure) would be present if the analyst made a habit of always dimensionalizing observations to some form of procedure, but technically, it wouldn't have been required to load the data. The event property in the Metadata dimension would indicate that the observation was part of a vital-sign observational event, making the procedure entry discretionary.

A new, more recent data source for vital-sign data is very likely to come from an application system that uses the narrower data architecture that tabularizes the specific observation types. The analyst who sources these data should create a single metadata entry for the generalized fact as depicted in the bottom of Figure 16.1. In this case, the observational property should also be sourced in order to provide for the dimension load of procedures from the values in the observation-type column of the vital-sign source. Typically, there's a lookup table somewhere in the source application that can be used to populate the definitions of those observation types.

This is a situation where the orthogonal nature of the data seam can't be avoided because the newer way of defining the facts is superior to the older way. There will be users who query the earlier data and expect that data to be available in the future. There will be users familiar with the new data who will expect to be able to query the data according to their encoded-observation-type experience in the source. To resolve this type of conceptual data seam, I recommend storing the data both ways. Doing so only requires that additional metadata be defined for each of the involved data sources. For the newer source,

the observational type property is sourced as the fact break in the new metadata, so each type of observation gets its own separate metadata entry just like in the old source. For the older source, the new metadata creates a new logical dataset where all the fact source columns share the same metadata titles as those used in the new source. The missing link in the treatment of the old source is that the column name in the old source needs to be used as an identifier in the Procedure dimension. To start, this should be hard coded into the SDI job for the old source. In the Gamma version, there will be new functions for more explicitly sourcing these properties. You'll also need to source the vital signs stored the old way from the warehouse in order to add them as new historical facts under the new metadata. This ability to source existing facts to create new facts is a standard function that you'll develop in the Gamma version.

By storing certain facts in more than one way, you maximize the flexibility available to your users for performing queries using their existing perceptions of what the data are and how it is stored. If your reaction to all of this is that you have to define one correct way of storing the data, you haven't completely internalized the dimension-fact architecture. A key power of this model is that all facts are independent of all other facts. There's no problem with having redundant facts. As you move toward consolidating data from a wider array of sources, you'll find that there are often redundant facts across these sources, and you should load all of them. Sometimes, the redundant facts will be used to validate each other, while at other times, they'll just be ignored. Eventually, you might even choose to remove some of them from the warehouse. The time to do that is when you observe that no user ever accesses them, not because you have some sense of needing to normalize the data. Let the Fact tables be messy and redundant just like the sources from which they are drawn.

Shifting toward Metrics

In recent years, many institutions have been motivated to build data warehouses in response to the need to produce metrics for organizational and regulatory reporting. This trend has caused organizations to focus more attention on their information and has increased the use of data warehousing generally, but the urgency with which many new metrics and reports need to be produced has often resulted in implementations that didn't effectively integrate or manage the information being used. New metric-oriented information technology projects were often about getting a new series of reports created, even at the expense of creating and managing a host of new silo datasets created explicitly for the needed reporting. Many of those datasets were described as data warehouses but failed to achieve any of the generality that would justify calling them warehouses. Many of these datasets were copies of transactional data from one of the clinical or operational systems that were made available for querying or reporting. These datasets were used to create

metrics and reports for the organization, and a complete set of metrics might need access to many of the individual silo datasets.

Data warehousing created an environment where all of these data were consolidated into the warehouse rather than creating new application-specific datasets, and desired reports were generated from the data warehouse. Organizations were still often report centric, but the data became common and centralized. Over time, the availability and use of the data warehouse drove a shift from report-dominated data usage toward more ad hoc querying and analysis of the data. Organizations gradually changed from being focused on reporting to being informatics based.

The irony of the rise of metrics in healthcare is that it has driven a renewed emphasis on *reporting* over *informatics* because the requirements of the metrics programs are often defined in terms of timely reporting. The challenge for data warehouse programs is to support these new reporting requirements while maintaining a focus on analytics and informatics as the cornerstone of the warehouse. You'll do this by keeping a clear distinction between the measures in your warehouse versus the metrics of interest to stakeholders. You'll design metrics reporting into the warehouse, but not by sacrificing the measures you've collected for your informatics programs.

Metrics Process Design

There's a distinction between your data warehouse design and the design of the organizational processes needed to conduct an effective metrics program. Figure 16.2 illustrates a process map of a general metrics program that will be supported by your data warehouse.

An effective metric program must scan the internal and external environment for measures that are needed to define and calculate required metrics and collect and categorize those measures. That function is served

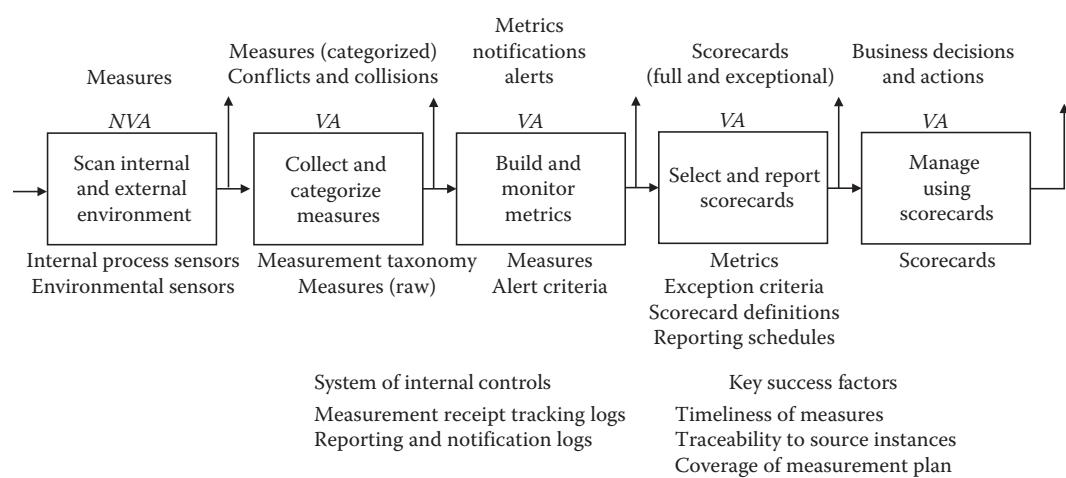


Figure 16.2 Organizational metrics management processes.

in your implementation by the source analysis and intake design processes conducted around the data warehouse and by your loading of that data. In maximizing your loaded data discussed earlier, making sure that all measures required for reportable metrics are included in your warehouse loads is a central concern. You ideally want to be able to calculate metrics directly from the measures stored, but this will often not be possible. In many cases, source measures for many metrics will be unavailable to the warehouse. In those cases, the values of the metrics themselves will sometimes be loaded into the warehouse directly from an external sensor or tool that is able to have access to data or algorithms that are not available to the warehouse. A certain amount of source traceability is lost, but the rest of the metrics reporting capability of the warehouse is enabled.

As long as the distinction between measures and metrics is maintained, these first two steps of the metrics management process directly coincide with the analysis and sourcing activities already associated with your new data warehouse. The data governance function must prioritize the definition and loading of measures that support desired metrics. The remaining steps of the metrics management process are distinctive to the definition and use of metrics and the displays in which they appear. Ultimately, your warehouse will be able to produce the metric dashboards and scorecards that your organization needs using metrics that have been derived from measures you've loaded into the warehouse or metrics you've loaded directly because the underlying measures were not available.

As the data warehouse approaches production status, the facts being loaded will become of paramount interest to new and potential users of the warehouse. Typically, users who write queries against your Fact tables are attempting to turn the measures in the warehouse into meaningful and useful metrics for research, management, and reporting. The storing of some of those metrics as derivative facts is discussed in Chapter 19. Much of the needed reporting will be outside of the institution as the organization works to satisfy the ever-increasing demands of program level and regulatory reporting that has emerged across the healthcare sector in recent years. The distinction between measures and metrics becomes important and will alter the Fact tables and dimensionality of your warehouse schema.

New Dimension and Fact Tables

Metrics are measures put to use. This is more than just a rhetorical device. Loading metrics into a data warehouse is a fundamentally different activity than loading measures into a data warehouse. Everything you've done up until now has involved measures in your single Fact table. Since any measure can be put to use in a variety of ways in the warehouse, an additional dimension is needed to accommodate the definition of the use to which any particular

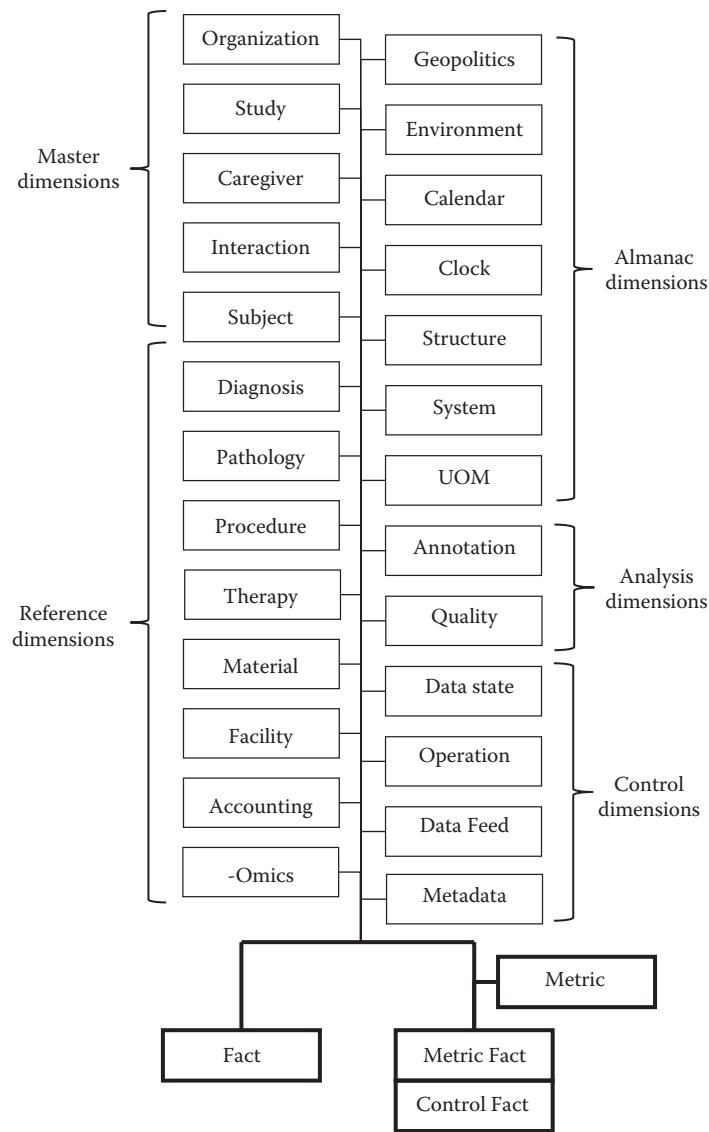


Figure 16.3 New dimension and Fact tables to support metrics.

measure, or set of measures, is being put. The new dimension is the Metric dimension (Figure 16.3), and it will have the following three subdimensions:

- *Metric*. Definitions of the metrics included in the warehouse
- *Control*. Definitions of the control values associated with those metrics
- *Display*. Definitions of the collections of metrics that are associated with each other for reporting purposes as dashboards or scorecards

The new Metric dimension is not added to the existing Fact table. By definition, the metrics defined in this new dimension describe how measures in the Fact table are used. They cannot be part of the dimensionalization of those original measures. We do need two new Fact tables in order to

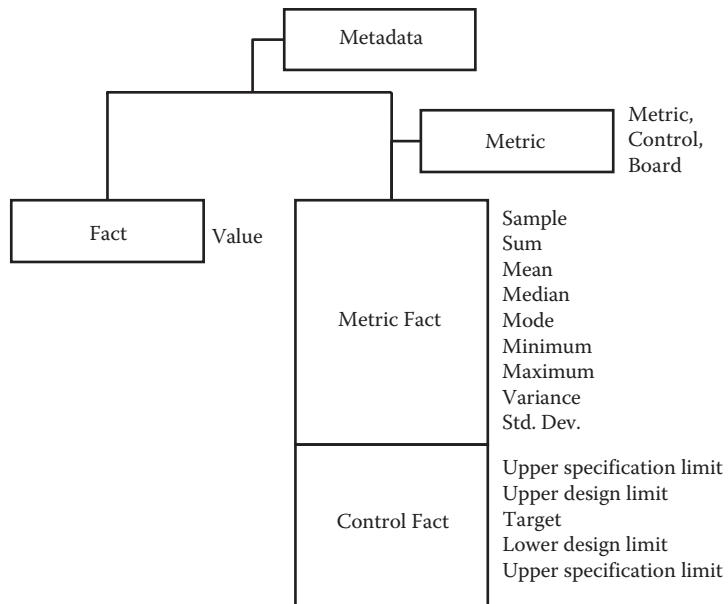


Figure 16.4 Properties of two new facts.

implement this metric model (Figure 16.4), and they will dimensionalize to this new dimension as well as all of the original dimensions in the model. These two new Fact tables are as follows:

- *Metric fact*. The values of a metric as a descriptive statistical profile
- *Control fact*. Supports the setting of targets and control values for metrics

These two new Fact tables are structurally different from the original Fact table used in the Alpha and Beta versions of the warehouse. While the original Fact table had just a single value column, these Fact tables have multiple value columns, each serving a specific purpose in the statistical definition or control of the metric in question. The three Fact tables are not interchangeable. As a sidenote, the existing Fact table might be more correctly referred to as Measure Fact table once this metric model is introduced into the design, but I don't see that happen typically. An unqualified reference to a Fact table is typically a reference to the original Fact table of measures unless the discussion context makes one or the other two the obvious subject of discussion.

Metric Fact Table

The Metric Fact table includes the columns necessary to record the aggregated parameters that define an instance of a sourced or calculated metric. The columns of the Metric Fact table include the following:

- *Sample size*. The number of included measure values aggregated
- *Sum*. The summed total of the included measure values

- *Mean*. The mean (average) value of the included measure values
- *Median*. The median value of the included measure values
- *Mode*. The mode value of the included measure values, Null if multimodal
- *Minimum*. The minimum value from the included measure values
- *Maximum*. The maximum value from the included measure values
- *Variance*. The total variance of the included measure values
- *Standard deviation*. The standard deviation of the included measure values, derived as the square root of the variance, typically in a nonmaterialized view

All columns are in the same unit of measure except for variance, which is in the unit of measure squared. For base measures included in the Metric Fact table, the sample size will be 1; the sum, mean, median, mode, minimum, and maximum will be the same value; and the variance and standard deviation will be NULL because they are not defined for a sample size of one. In the case of metrics that have been defined as binomial, the sum, sample size, and mean serve as the expected numerator, denominator, and ratio for analysis.

Control Fact Table

The Control Fact table includes the columns necessary to define expected design and specification limits around the metric in question. The columns of the Control Fact table include the following:

- *Upper specification limit (USL)*. The value above which the metric would be considered defective, typically indicated by red in metric displays, ignored for more-the-better (MTB) metrics
- *Upper design limit (UDL)*. The value above which the metric would be considered an exception, typically indicated by yellow in metric displays, ignored for MTB metrics
- *Control value*. The value against which the metric is to be controlled
- *Lower design limit (LDL)*. The value below which the metric would be considered an exception, typically indicated by yellow in metric displays, ignored for less-the-better (LTB) metrics
- *Lower specification limit (LSL)*. The value below which the metric would be considered defective, typically indicated by red in metric displays, ignored for LTB metrics

These control values support building basic data controls to be built into scorecards and dashboards. If the data fit the necessary assumptions of statistical process control (SPC), a rudimentary SPC-oriented program can be enabled with the displays ([Figure 16.5](#)).

I refer to this processing as a form of *rudimentary* SPC because the viability of the approach depends on the way metrics are calculated and sampled.

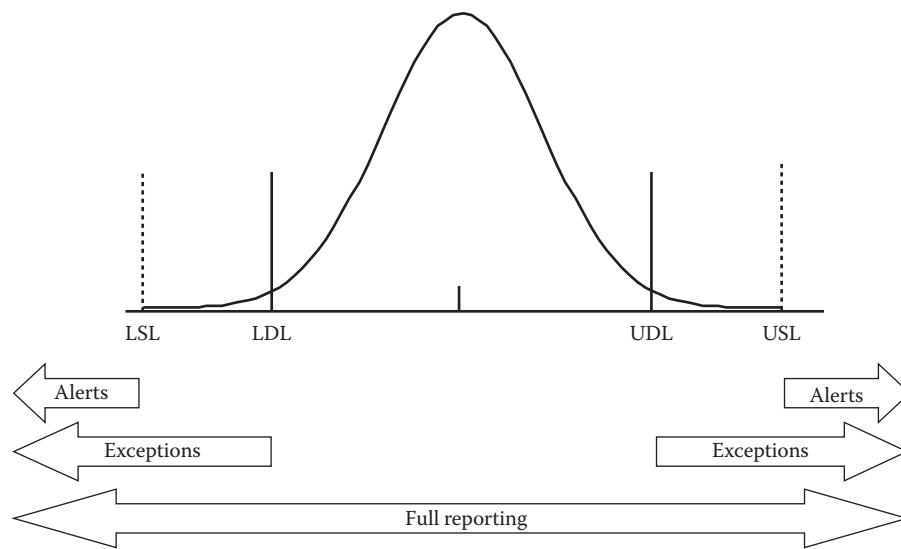


Figure 16.5 Data control levels enable a form of statistical control.

There is nothing about this functional design of the metric capabilities of the warehouse that ensures that the assumptions of data validity and distribution that needed to use SPC to manage processes will be met. True statistical control of data depends on the rational sampling of data to take advantage of the central limit theorem in statistics to ensure normal-behaving data, and the sample size of any data point determines the various parameters that would be used to calculate control limits in any particular SPC chart. If those assumptions are met, and the control values set accordingly, the displays enabled can be used to provide a certain level of management control directly from data warehouse displays. More likely, these displays can serve as a first-order rudimentary control, and the associated measures and metrics can be exported into a more robust or mature toolset for conducting more detailed SPC analysis. This external analysis can be used as a new data source for loading the analytical results back into the data warehouse for further data integration and analysis.

Metric Subdimension

The new Metric subdimension allows each individual metric to be defined to the warehouse. Since the warehouse doesn't directly derive or calculate base metric values (i.e., it is not a metric engine), the metric definitions that populate the new subdimension are largely descriptive. They are also highly dependent upon the dimensionality associated with each value in the warehouse. Failure to integrate metric definitions with the larger metamodel for the warehouse will impede a lot of the data integration being sought.

For example, the Metric subdimension might be used to define Length of Stay as a metric. Depending upon the dimensionality that is used to load values for this metric, the warehouse might contain any of the following reportable values:

- Average inpatient Length of Stay by year
- Median Length of Stay by year for female, obese, diabetics
- Average inpatient Length of Stay by month by clinical specialty
- Monthly Length-of-Stay standard deviation by nursing unit
- Median inpatient Length of Stay by year by payer
- Outpatient daily maximum Length of Stay by clinic
- Median Length of Stay by year by provider

These are all the same metric. They vary based on the dimensionality and level of aggregation assigned to each of the various warehouse dimensions within which values can be loaded. It is important that the warehouse dimensions be allowed to serve their ontological role in the continuing design of data coming into the warehouse.

What all of these examples have in common is their underlying focus on the Length-of-Stay metric. Metric values tend to have some temporal component (i.e., we typically wouldn't discuss the average Length of Stay since the beginning of time) and some area of focus. Focus might be on encounter types, disease categories, patient demographics, or healthcare facilities. Likewise, any one of these metric values might be loaded in a variety of units of measure. Some might report in hours, others in days. Long-term care might be reported in months. The point here is that those differences are in the dimensionality of the metric values, not the ontological definition of the metrics themselves.

This doesn't mean that there will be only a single Length-of-Stay row in the dimension. Ideally, we'd like there to be only one, because we'd like to think that our entire institution can agree on, and manage to, a single definition of each metric, particularly core metrics used throughout the healthcare sector. Chapter 20 will discuss data governance strategies for making this agreement happen over time, but our design will have to be based on the notion that people will have lots of subtly different definitions for what might otherwise be considered the same metric. Basically, it comes down to the notion that there should be a row in the Metric dimension for each different way we calculate the metric in question. We might have

- Length of Stay—simple (discharge minus admission timestamps)
- Length of Stay—extended (departure minus arrival timestamps)
- Length of Stay—ad hoc (latest minus earliest clinical event timestamps)
- Length of Stay—heart center (whatever they report!)

There might be any number of metric definitions for any single concept, and many will be specific enough that someone's name or functional area can end up in the name of the metric itself.

To the extent that Metric is a standard dimension in the data warehouse architecture, you can take advantage of all of the capabilities of a standard dimension to help in managing these definitions. At the very least, I typically create a single generic version of each metric that I can use as a hierachic parent to all of those variations. This allows one to easily query all of the different definitional values for comparison without having to know about all of the variants and without having to design separate queries for each (assuming that the variants share enough dimensionality to make the comparisons meaningful). There are a few distinct properties that are expected as part of each metric definition in the subdimension, which are as follows:

- *Binomial flag*. Whether or not the metric is binomial in its base values, meaning that the metric with a sample size of one will always be either one (1) or zero (0)
- *Goodness indicator*. Whether goodness for the metric is defined as More-the-Better (MTB), Less-the-Better (LTB), or targeted (TGT) (Figure 16.6)

Examples of some metrics and how they might be characterized by these two dimension-specific properties include the following:

- Survival (binomial, MTB)
- Readmission (binomial, LTB)
- CPOE orders (binomial, TGT)
- Early Obstetrics (OB) deliveries (binomial, LTB)
- Arrival to departure time (nonbinomial, LTB)
- Clinical result (nonbinomial, TGT)
- Blood glucose (nonbinomial, TGT)

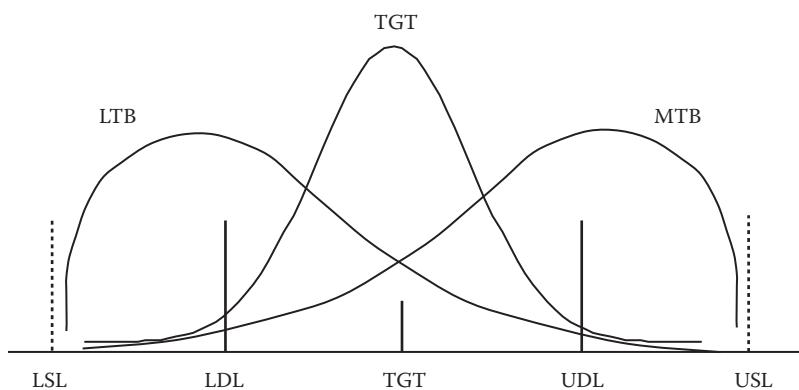


Figure 16.6 Determining direction of goodness.

Beyond these properties that help control metric processing, any number of other descriptive entries can be added to this subdimension depending upon the organization's requirements and what other data dictionaries might already be in use. Typical additional descriptions might include the definition of the metric, the general algorithm for calculating the metric, and the specific inclusions, exclusions, or rejections that are implemented by an algorithm. These columns are descriptive, not operational, so the warehouse team might put any number of these columns into the subdimension as documentation.

Control Subdimension

The Control subdimension allows the control parameters to be defined for any particular metric. Controls can be defined for any metric using any of the descriptive statistical values associated with the Metric Fact table. In fact, rows in the Control subdimension are natural hierarchy children of their respective metrics. As such, they can actually be internally generated any time a new metric row is added by cross joining the key of the metric with the list of columns available in the Metric Fact table. In this way, users might define controls against any aspect of any metric. For example, as in the aforementioned Length-of-Stay examples, one user might want to control against a median value for a metric, while another might want to control against the maximum or variance values. The control column chosen will be specific to the intended use of any display into which the controlled metric is introduced.

Not all possible controls make logical sense for every metric. Binomial metrics (those characterized as either a 0 or 1 numerator value) are typically only controlled as counts and rates. Examples of some binomial metric controls include the following:

- Survival count
- Survival rate
- Readmission count
- Readmission rate
- CPOE orders count
- CPOE orders rate
- Early OB delivery count
- Early OB delivery rate

Nonbinomial metrics can typically take advantage of the full range of descriptive statistics available in the Metric Fact table. Some examples include the following:

- Arrival to departure time median
- Arrival to departure time maximum
- Arrival to departure time variance
- Lab result count
- Blood glucose level median
- Blood glucose level mode
- Blood glucose level variance

Display Subdimension

The Display subdimension is where controlled metrics are combined into hierarchies that can serve as dashboards and scorecards. The main mechanism through which this is accomplished is the Hierarchy table in the dimension.

Display entries are created in an arbitrarily complicated hierachic stack, the bottom of which always connects the leaf display with the controlled metrics that make up that display. Use the Sequence column of the Hierarchy table to control the order of the metrics in a display, as needed. In this way, a dashboard or scorecard can be built, made up of arbitrarily nested subsets that eventually present controlled metrics in the values.

Populating Metric Values

Generating metric values in the Metric Fact table will not be conceptually any different than the definition and loading of the measure facts you've been doing throughout your development. The main difference is that you'd like to be able to source the measures to be used by your new metrics directly from the warehouse itself. In theory, you'd like it to always be true that the measures used to populate a metric also exist in the warehouse for inspection by anyone using that metric. In practice, it will often be true that something that needed to populate a metric doesn't exist in the warehouse. In extreme cases, the value for a metric will arrive in a source data feed like any other value being loaded into the warehouse. Along a continuum in between, some of the measures needed to calculate a metric will exist in the warehouse, and others will not. Regardless, metric values loaded into the warehouse are new facts that target a different Fact table, but that otherwise take advantage of all of the ETL functionality of the generic warehouse architecture.

Let's go back to our Length-of-Stay metric as an example. Suppose we have an inpatient variant of the metric defined as the difference between the admission and discharge times for the encounter. We might also have a variant for ambulatory encounters defined as the difference between the arrival and departure times for the encounter. To keep it simple, let's suppose that we want the dimensionality of the new metric value to match the dimensionality of the encounter's discharge or departure fact. This means we'd most likely be aggregating the resulting metrics by final facility, final attending physician, final diagnosis, etc. If we want other aggregations, we'll need an orthogonal query to get them, or we'll need a more complicated SDI job to derive different dimensionality for these metric facts. We'll keep the assumptions very simple for now.

You want to load new metric facts—your calculated lengths of stay—into your warehouse as new metric facts. As such, you need to conduct the same analysis and design you've already been carrying out for other sources being brought into the warehouse. The difference is that the source data you're looking at happen to

be in the warehouse itself. You'll need to identify the necessary data, design, and load metadata for the loading of that data, and construct a SDI query to obtain the data and send it into your standard generic ETL processes.

The general strategy will be straightforward. You want to extract a single Length-of-Stay fact for each encounter that has been completed and for which you haven't already extracted a value. You'll consider an encounter completed for these purposes if your query finds the requisite active discharge or departure event fact in the Fact table, and you'll control your sourcing against the modified timestamp of that Encounter.

```

SELECT META.EVENT AS LOGICAL_DATASET
      FACT.ORGANIZATION_GROUP_ID,
      FACT.CAREGIVER_GROUP_ID,
      FACT.SUBJECT_GROUP_ID,
      FACT.FACILITY_GROUP_ID,
      FACT.DIAGNOSIS_GROUP_ID,
      FACT.QUALITY_GROUP_ID,
      FACT.INTERACTION_GROUP_ID,
      DIFF(ID.DISCHARGE_TIMESTAMP, ID.ADMISSION_TIMESTAMP) AS LENGTH_OF_STAY,
      FACT.SOURCE_TIMESTAMP
   FROM INTERACTION_D ID
  INNER JOIN INTERACTION_B IB ON (ID.MASTER_ID = IB.MASTER_ID
                                   AND ID.MODIFIED_TIMESTAMP > ** CONTROL TIMESTAMP **)
  INNER JOIN FACT          ON (IB.GROUP_ID = FACT.INTERACTION_GROUP_ID)
  INNER JOIN METADATA_D META ON (META.MASTER_ID = FACT.METADATA_MASTER_ID
                                   AND META.EVENT = 'Discharge')
  INNER JOIN DATA_STATE_D DSD ON (DSD.MASTER_ID = FACT.DATASTATE_MASTER_ID
                                   AND DSD.DESCRIPTION = 'Active')

UNION

SELECT META.EVENT AS LOGICAL_DATASET
      FACT.ORGANIZATION_GROUP_ID,
      FACT.CAREGIVER_GROUP_ID,
      FACT.SUBJECT_GROUP_ID,
      FACT.FACILITY_GROUP_ID,
      FACT.DIAGNOSIS_GROUP_ID,
      FACT.QUALITY_GROUP_ID,
      FACT.INTERACTION_GROUP_ID,
      DIFF(ID.DEPARTURE_TIMESTAMP, ID.ARRIVAL_TIMESTAMP) AS LENGTH_OF_STAY,
      FACT.SOURCE_TIMESTAMP
   FROM INTERACTION_D ID
  INNER JOIN INTERACTION_B IB ON (ID.MASTER_ID = IB.MASTER_ID
                                   AND ID.MODIFIED_TIMESTAMP > ** CONTROL TIMESTAMP **)
  INNER JOIN FACT          ON (IB.GROUP_ID = FACT.INTERACTION_GROUP_ID)
  INNER JOIN METADATA_D META ON (META.MASTER_ID = FACT.METADATA_MASTER_ID
                                   AND META.EVENT = 'Departure')
  INNER JOIN DATA_STATE_D DSD ON (DSD.MASTER_ID = FACT.DATASTATE_MASTER_ID
                                   AND DSD.DESCRIPTION = 'Active');

```

Note that this sourcing query is taking advantage of the final ETL code build that was completed in the last chapter, particularly the ability to source internal warehouse surrogate keys as source columns. This allows the metric facts to be dimensionalized the same way as the originating admission-discharge-transfer (ADT) facts without the need to access all of the data that would have been needed in the original bridges and groups of each of those dimensions in those original loads. You want your metric dimensionality to

match the original dimensionality, so you source the Group ID surrogates that were available in those facts.

Also note that the sample code also treats the two ADT events as logical dataset identifiers for the metric source data. This will allow the same metadata to be used for both by defining metadata at the physical dataset level, except for the facts themselves which will have their own distinct metadata and implied units of measure. Having different metadata for each metric fact also allows standard fact superseding logic to be used to supersede older version of the metric that might occur if the originating encounters are updated with different ADT dates.

This brings up an important point that needs to be made explicit: there is a big difference between information stored in the Metadata dimension and information stored in the Metric dimension. It's possible to imagine both being used for the same purpose in order to avoid having to implement the new Metric dimension. There was a time when I would have implemented my metrics using just the Metadata dimension to differentiate the metrics. I used to put the metric definitions in the Accounting dimension, thinking of metrics as analogous to memo counts in a general ledger. That approach worked for years, but ultimately failed because the differences between these two constructs became increasingly important as I worked to build more metrics functionality into this general architecture.

The distinction that wasn't supported under that old model was ontological. Metadata tells me about what is in the Fact table. In the example we're looking at here, there are two sets of metadata: one for the inpatient base metrics and one for the ambulatory base metrics. They are different and are subject to different processes and constraints. The metric definition tells us about the metric. Both of these metadata entries result in the loading of the same metric. There's a relationship between the metric definition and the definitions of the metadata for the sourced facts from which it is aggregated, but that relationship is not one to one. In fact, over time, the number of metadata instances that end up mapped into a metric will always grow. This example has two metadata for the two logical sources. Eventually, as more data are added to the warehouse, more ADT events can be loaded using new metadata, some of which would end up being used to create values for this metric. As you merge or acquire more healthcare settings, you end up loading additional EHR data into the warehouse, giving you new encounters and new sources of Length-of-Stay data. That data will all be for the same metric, but it will tend to be identified with different metadata in your main Fact table.

Populating Control Values

In addition to loading all of the desired Metric Fact values, the control facts that will be used for the querying and display of those metrics in defined displays are also needed. Note that no controls are *required* by this implementation. You can

query the Metric Fact table at any time without setting up any controls. You can also define display capability without actually setting control values. The Control Fact table is an optional load, and my experience is that organizations tend to move into this area very slowly. Initial controls are typically built at the overall organization level, and more refined or detailed control values are added over time as the organization becomes more mature in its informatics focus and more experienced in using metrics.

Let's take the Length-of-Stay metric as a continuing example. Suppose someone in your leadership team determined that they wanted to control the Length-of-Stay metric around its median value, and they were interested in controlling against a median value of 4.8 days. The desired target was to be within 25% of that value, and anything beyond 50% off target was to be considered a control exception. This scenario would give you the following control fact requirements:

<i>Control Fact Column</i>	<i>Value</i>
CONTROL_COLUMN	Median
LOWER_SPECIFICATION_LIMIT	2.4
LOWER_DESIGN_LIMIT	3.6
CONTROL_VALUE	4.8
UPPER_DESIGN_LIMIT	6.0
UPPER_SPECIFICATION_LIMIT	7.2

To keep this first example simple, let's assume that your institution has only this single control value in mind, and that it will be applied to all instances of the metric. That means that the source intake you design for these data will not need any dimensional data except the natural key for the metric itself. All other dimensions will default to apply to all entries in the other dimensions.

The control characteristics for this metric end up looking like a fairly typical LTB metric, as shown in Figure 16.7. When included in a dashboard or scorecard display, the metric value will appear medium gray (or green in color) up to and

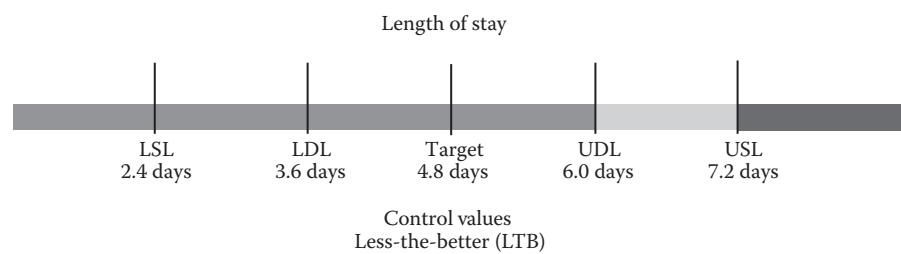


Figure 16.7 Example qualitative coloring for a less-the-better metric.

including the UDL of 6.0 days. Above that point, but not above the USL, the value will appear white or light gray (or yellow in color). Above the 7.2 USL, the value will appear dark gray or black (or red color).

Note that because the Length-of-Stay metric is defined as an LTB value, all values below the target value are treated, for scorecard and dashboard purposes, as green. The same result would have been achieved if both control values had been set to zero. This doesn't mean that those lower values shouldn't be specified however. Making values red–yellow–green is just one function for which control values are used. Admittedly, this is the primary purpose for these values in organizations just starting out with metrics.

Another use for control values is to identify exceptional values in situations where querying all values isn't desired or intended. In the case of Length of Stay, someone might be very interested in querying only those encounters that have lower than expected lengths of stay. The control value provides three thresholds that can be included in queries without having to know the threshold values in advance. You can query those cases with values less than target, or less the LDL, or less than the LSL. The exceptional cases become available in standard queries by using the control values when filtering. So, even for LTB metrics, controls should be designed toward identifying what the most useful thresholds would be for exception-oriented queries.

Populating Displays

The third subdimension of the new Metric dimension involves defining the various displays on which the defined controlled metrics will appear. Like control values, this feature of the warehouse is completely optional, with metric data being available to queries regardless of whether or not display entries are defined and loaded. At its simplest, a display is a hierachic collection of other displays, with Control Metrics at the leaves.

The terminology I use to discuss displays in queries includes both *dashboards* and *scorecards*. The two are structurally identical but serve two different organizational purposes. A dashboard is a presentation of data for the purpose of allowing those exercising a process or procedure to improve their own performance. The data on a dashboard are about that process, preferably as close to real time as possible. A scorecard is a presentation of data for the purpose of making factors regarding a process or procedure visible to some third party beyond the individuals exercising the activities depicted. Dashboards are a self-management tool, while scorecards are an organizational management tool.

Take your car and driving as an example. The dashboard in your car provides you with visible metrics needed in order to better drive the car. You can see your speed, fuel level, oil pressure, and other factors considered valuable to improving driving success in real time. A scorecard that measures your driving wouldn't be interested in your dashboard factors. The scorecard might include the number

of accidents you've caused or the number of speeding tickets you've received. Information technology has helped both dashboards and scorecards to evolve. Your car dashboard can probably now tell you the rate at which your driving style is consuming fuel, hopefully to cause you to drive better. Fuel economy has become a dashboard metric. Your car can now also tell your insurance company how often you have to suddenly hit your breaks hard enough to engage the antilocking feature, a new scorecard metric.

Whether a given metric is an effective value to be placed on a dashboard or scorecard is a discussion of metric program design. That's a big topic beyond the scope of this book, but it's an important area if your organization starts building a lot of dashboards and scorecards out of the data warehouse. The Display subdimension is intended to facilitate the implementation of any dashboard or scorecard. Let's hope they are good ones. Displays are conceptually quite simple. A basic display might be one that is defined as containing only a small set of Control Metrics. Other displays can be defined, the components of which are other displays, often basic displays. Using the hierarchy feature of the Metric dimension, these displays can be made arbitrarily complex, with displays nested within displays to create cascades of sections and subsections in the resulting data displays (Figure 16.8).

The tough part in defining these displays in the Metric dimension is that they are highly abstract, and there is often no source of data from which to do these loads. Unlike loading one of the clinical dimensions, where an analyst can source the data from some existing dataset that contains the required data, the display definitions don't necessarily exist anywhere. Defining them is more like software engineering than data warehouse analytics. The skill sets required to do this well are slightly different. As a result, the earliest displays defined are likely to be very simple, using components that have been defined outside of the warehouse project for their definitions.

As time passes, you'll recognize that there are many situations in which users query the same small set of controlled metrics together almost all the time. These will be your candidates for early displays. You'll also encounter situations where some organizational initiative demands that certain metrics be collected and reported periodically. We've spent years preparing and reporting

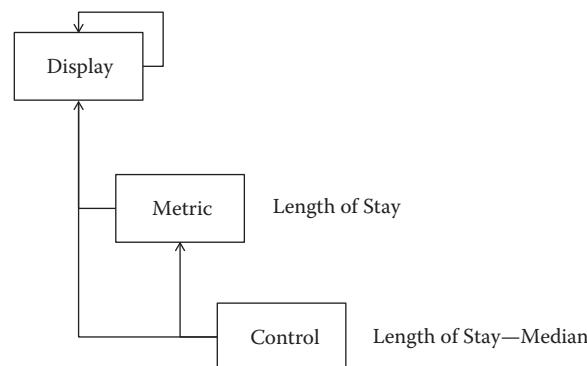


Figure 16.8 Simple display with one targeted metric.

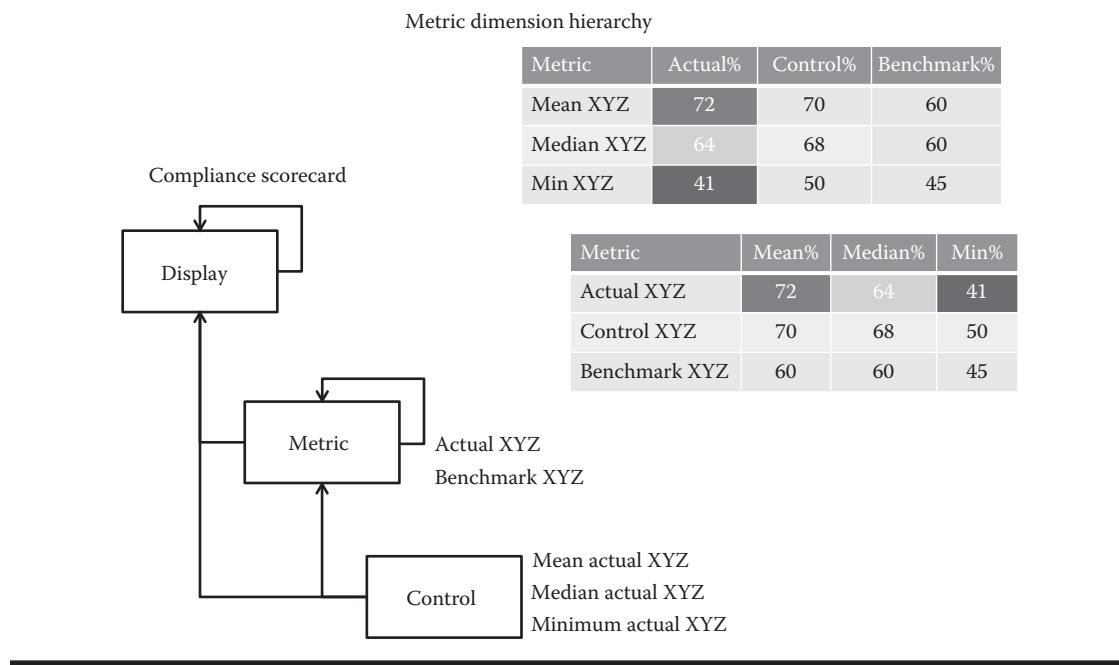


Figure 16.9 Scorecard with three targeted and benchmarked metrics.

meaningful use metrics, and now accountable care metrics that fall in these categories. These are all wonderful examples of displays that can be defined in the Metric dimension. Defining them is optional. Queries can always be built to directly filter on a list of metrics and controls, but being able to simplify those queries by directly filtering on the display desired makes using these metrics much easier (Figure 16.9).

Metric Aggregation

With the standardization of a generic metric and control structure against measures in the warehouse, the aggregation of the actual metrics becomes a fairly straightforward process of querying the measures of interest in order to create the aggregate desired metrics. The query always involves aggregating measures in the Fact table through the hierarchies of the dimensions of interest in order to create a metric value at each level of any associated hierarchy.

Figure 16.10 illustrates the creation of metrics for each nursing unit, building, and campus using measures captured at the nursing unit level. Each entry in the hierarchic structure will end up with a metric entry, the sample size of which is a function of how many measures existed under that entry in the hierarchy.

```

SELECT METRIC,
       FACILITY,
       SUM(MEASURE)          AS SUM,
       COUNT(*)               AS SAMPLE,
       AVG(MEASURE)           AS MEAN,
       MIN(MEASURE)           AS MINIMUM,
       MAX(MEASURE)           AS MAXIMUM
  
```

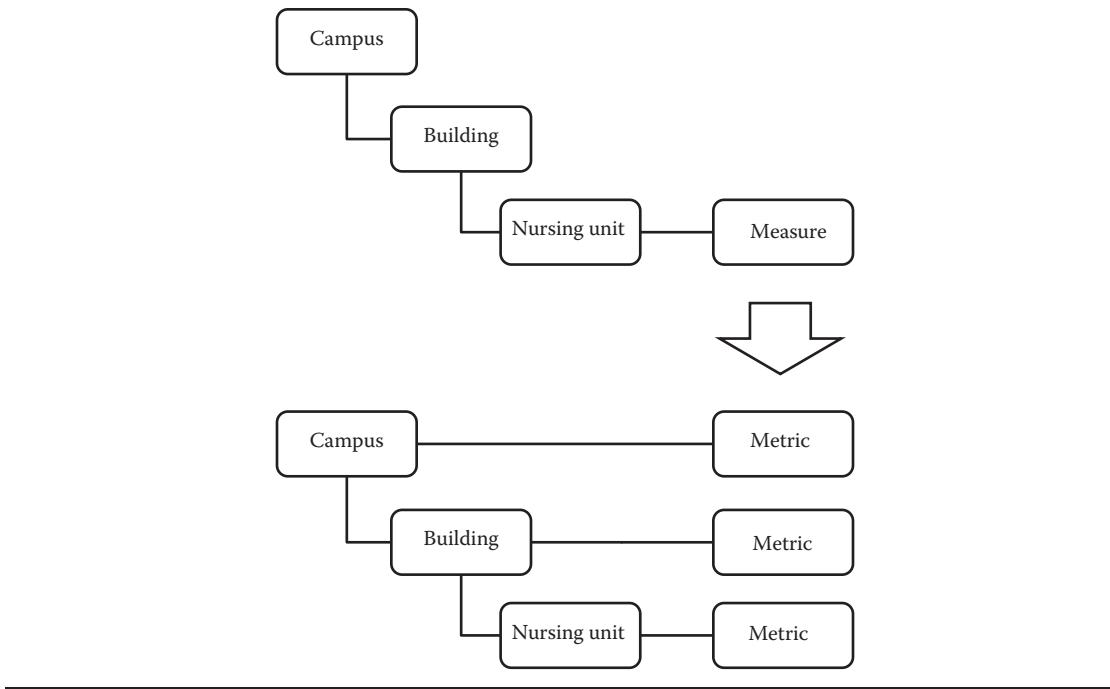


Figure 16.10 Example facility aggregated metrics from single measure.

```

FROM
(
    SELECT META.ENTITY          AS METRIC,
           FH.ANCESTOR_MASTER_ID AS FACILITY,
           TO_NUMBER(FACT.VALUE)  AS MEASURE
    FROM FACT
    INNER JOIN METADATA_D META ON (FACT.METADATA_MASTER_ID = META.MASTER_ID
                                    AND META.ENTITY IN('Stroke', 'VTE')
                                    AND META.ATTRIBUTE = 'Numerator')
    INNER JOIN FACILITY_B FB ON (FACT.FACILITY_GROUP_ID = FB.FACILITY_GROUP_ID)
    INNER JOIN FACILITY_D FD ON (FB.MASTER_ID = FD.MASTER_ID)
    INNER JOIN FACILITY_H FH ON (FD.MASTER_ID = FH.DESCENDENT_MASTER_ID
                                AND FH.PERSPECTIVE IN ('~SELF~', '~NATURAL~'))
)
GROUP BY METRIC, FACILITY;

```

The actual metrics created in this way are subject to some variation depending upon the requirements for the use of the targeted metrics, but typically a simple aggregation will suffice for most purposes. If metrics are required even where no measure values exist, a left join situation is driven through the aggregation query to ensure all metrics are created, even when the sample size is zero.

This empty branch scenario happens a lot in calendar aggregations where measures are not always present for every day on the calendar. [Figure 16.11](#) illustrates a calendar aggregation where measures collected daily are aggregated into metrics at the day, month, quarter, and year levels.

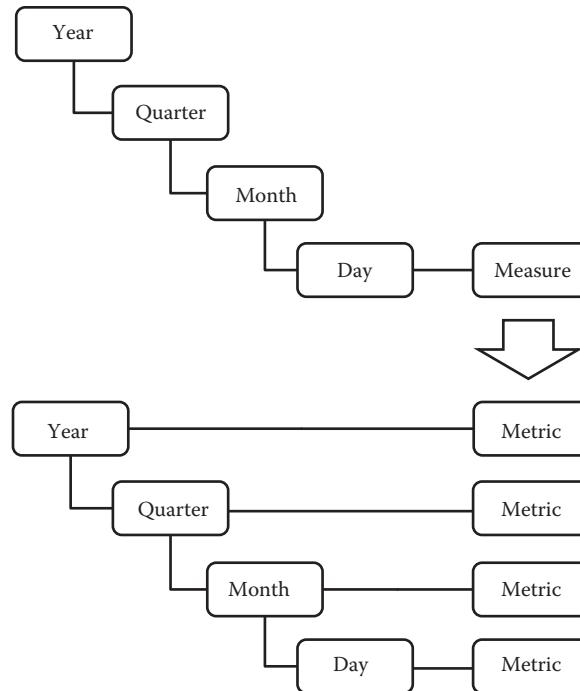


Figure 16.11 Example calendar aggregated metrics from single measure.

```

SELECT METRIC,
       CALENDAR,
       SUM(MEASURE)           AS SUM,
       COUNT(*)               AS SAMPLE,
       AVG(MEASURE)            AS MEAN,
       MIN(MEASURE)            AS MINIMUM,
       MAX(MEASURE)            AS MAXIMUM
  FROM
    (
      SELECT META.ENTITY                  AS METRIC,
             CH.ANCESTOR_MASTER_ID     AS CALENDAR,
             TO_NUMBER(FACT.VALUE)      AS MEASURE
        FROM FACT
       INNER JOIN METADATA_D META ON (FACT.METADATA_MASTER_ID = META.MASTER_ID
                                         AND META.ENTITY IN('Stroke', 'VTE')
                                         AND META.ATTRIBUTE = 'Numerator')
       INNER JOIN CALENDAR_B CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
       INNER JOIN CALENDAR_D CD ON (CB.MASTER_ID = CD.MASTER_ID)
       INNER JOIN CALENDAR_H CH ON (CD.MASTER_ID = CH.DESCENDENT_MASTER_ID
                                    AND CH.PERSPECTIVE IN ('~SELF~','~NATURAL~'))
    )
 GROUP BY METRIC, CALENDAR;
  
```

The appearance of the algorithmic calculation of metrics can get very complicated when the multidimensionality of the warehouse is taken into consideration. Fortunately, it's only the appearance that gets complicated. Calculating an N-dimensional aggregation isn't really any more complex than a 1 or 2 dimensional aggregation because the structures of the dimensions, particularly the hierarchy components of the standard design pattern, make it relatively routine to conduct comparable hierachic aggregations.

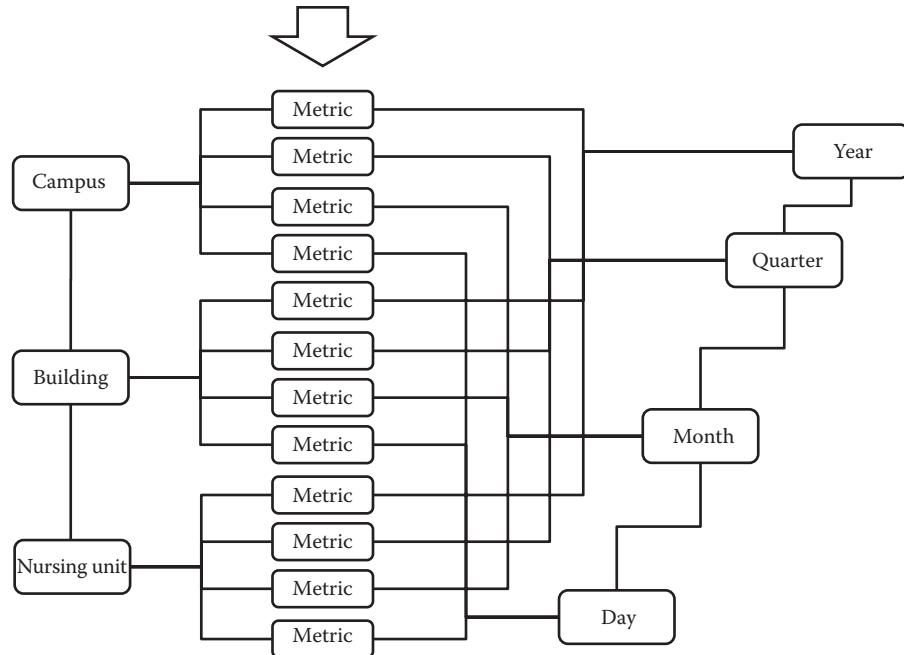


Figure 16.12 Example facility–calendar aggregated metrics from single measure.

Figure 16.12 illustrates the 12 metrics that are created if a 2 dimensional aggregation is conducted across the Facility and Calendar dimensions of the previous examples. Each layer of the facility hierarchy gets a metric value set calculated for each level of the calendar hierarchy.

```

SELECT METRIC,
       FACILITY,
       CALENDAR,
       SUM(MEASURE)           AS SUM,
       COUNT(*)               AS SAMPLE,
       AVG(MEASURE)            AS MEAN,
       MIN(MEASURE)            AS MINIMUM,
       MAX(MEASURE)            AS MAXIMUM
FROM
(
  SELECT META.ENTITY          AS METRIC,
         FH.ANCESTOR_MASTER_ID AS FACILITY,
         CH.ANCESTOR_MASTER_ID AS CALENDAR,
         TO_NUMBER(FACT.VALUE) AS MEASURE
  FROM FACT
  INNER JOIN METADATA_D META ON (FACT.METADATA_MASTER_ID = META.MASTER_ID
                                   AND META.ENTITY IN('Stroke', 'VTE')
                                   AND META.ATTRIBUTE = 'Numerator')
  INNER JOIN FACILITY_B FB ON (FACT.FACILITY_GROUP_ID = FB.FACILITY_GROUP_ID)
  INNER JOIN FACILITY_D FD ON (FB.MASTER_ID = FD.MASTER_ID)
  INNER JOIN FACILITY_H FH ON (FD.MASTER_ID = FH.DESCENDENT_MASTER_ID
                               AND FH.PERSPECTIVE IN ('~SELF~','~NATURAL~'))
  INNER JOIN CALENDAR_B CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
  INNER JOIN CALENDAR_D CD ON (CB.MASTER_ID = CD.MASTER_ID)
  INNER JOIN CALENDAR_H CH ON (CD.MASTER_ID = CH.DESCENDENT_MASTER_ID
                               AND CH.PERSPECTIVE IN ('~SELF~','~NATURAL~'))
)
GROUP BY METRIC, FACILITY, CALENDAR
    
```

Each metric, after aggregation, stands alone for querying and reporting. Whether querying the quarterly numbers for a building or monthly numbers for a campus, the metric fact will provide the sample size and related descriptive statistics for measures that were found within that aggregation. Figure 16.13 illustrates a few aggregated binomial metric values. Because each raw value being aggregated into a binomial metric is either zero or one, certain patterns in aggregated metrics can be seen. For example, the median and mode are always the more common of those zero or one values within the sample, so don't provide any new information.

For binomial metrics, the fact columns of interest are usually the sum, sample, and mean. For compliance metrics, where the metric is defined as the ratio of a numerator and denominator derived from the data, the sum serves as numerator because the sum of a column containing only zero and one values is effectively a count of the ones, so usually serves as a positive assertion toward whatever compliance is being measured by the metric. The sample serves as the denominator because a count of all of the facts, whether zero or one, represent the full number of opportunities being measured by the metric. The mean then represents the ratio of the two.

In Figure 16.14, you can see that 25 facts were aggregated, as evidenced in the sample value for the grand total. These facts were also aggregated at a lower level of the relevant hierarchy, shown in the figure as the X, Y, and Z level totals that aggregated 7, 10, and 8 raw facts, respectively. The X total aggregate included four 1 values (sum) out of the 7 values (sample) for a compliance ratio (mean) of 57%. Each metric fact can be queried in the same manner, and a user need not to know the structure or depth of any hierarchies involved in the aggregations that created those facts.

Figure 16.13 similarly illustrates several nonbinomial aggregated metric facts, this time for an arbitrary aggregation A, B, and C in one of the dimensions. Because a nonbinomial fact can take on any value, all of the columns of the Metric Fact table become more useful, although some of the columns might be illogical depending on what is being measured.

	Sum	Mean	Median	Mode	Sample	Variance	Std. Dev.
A-total	499.0	71.3	77.0	#N/A	7	299.57	17.3
B-total	764.0	76.4	77.0	93.0	10	278.71	16.7
C-total	497.0	62.1	64.5	71.0	8	240.41	15.5
Grand	1760.0	70.4	71.0	69.0	25	287.58	17.5

Figure 16.13 Example of aggregated nonbinomial metric facts.

	Sum	Mean%	Median	Mode	Sample	Variance	Std. Dev.
X-total	4	57	1.0	1.0	7	0.286	0.53
Y-total	7	70	1.0	1.0	10	0.233	0.48
Z-total	5	63	1.0	1.0	8	0.268	0.52
Grand	16	64	1.0	1.0	25	0.240	0.49

Figure 16.14 Example of aggregated binomial metric facts.

The quantitative values in the Metric Fact table can be used to support any routine statistical analyses for the metrics and dimensions involved. A very common use is to display and control median or mean values in scorecards or dashboards. Another common use is to identify subpopulations of the warehouse that can be classified as significantly similar or different based on a student's t-test (or F-test as the sample size grows) on the aggregated variance values. For example, if you aggregate a quantitative value for a set of facts against two cohorts in the Subject dimension or two protocols in the Study dimension, you can statistically decide if the two groups are significantly different based on these facts. While not all desired statistical analyses can be accomplished using this Metric Fact table, I've found that a wide array of common analysis queries can be accomplished, and detailed data can be extracted for more sophisticated analyses in other tools or software packages.

Metric–Control Fact Views

While any of the metric facts and control facts loaded into the data warehouse can be queried independently, the use of controlled metrics to build dashboards and scorecards through the Display subdimension implies a view of these two types of facts joined together. Figure 16.15 illustrates an example of what might be encountered in such a view using the nonbinomial sample data from [Figure 16.13](#). It shows a single control fact set of values that provides control values against the median value in the Metric Fact table.

The desired view for these data joins the Metric Fact table to the Control Fact table in a manner that allows the aggregated metrics to be queried along with their requisite control values, preferably without needing to know what control values have actually been loaded or at what level of the associated hierarchies.

Metric fact							
	Sum	Mean	Median	Mode	Sample	Variance	Std. Dev.
A-total	499.0	71.3	77.0	#N/A	7	299.57	17.3
B-total	764.0	76.4	77.0	93.0	10	278.71	16.7
C-total	497.0	62.1	64.5	71.0	8	240.41	15.5
Grand	1760.0	70.4	71.0	69.0	25	287.58	17.5

Control fact					
LSL	LCL	TGT	UCL	USL	Type
51	58	62	65	72	Median

Metric–control fact (view)						
	VALUE	LSL	LCL	TGT	UCL	USL
A-total	77.0	51	58	62	65	72
B-total	77.0	51	58	62	65	72
C-total	64.5	51	58	62	65	72
Grand	71.0	51	58	62	65	72

Figure 16.15 Example metric–control pivoted view.

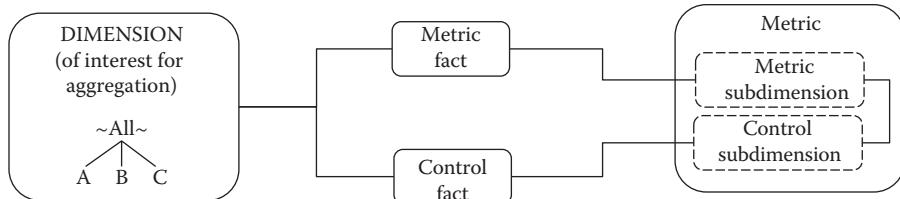


Figure 16.16 Conceptual model for metric and control view query.

The example in the figure doesn't explicitly indicate the level of aggregation that the control values have been defined for. You can't tell if the control values have been loaded only as a single set of values for the entire dimension hierarchy or if they have been defined for each of the A, B, and C branches of the hierarchy separately. The fact that in this example the same control values end up being used for all of the values in the resulting view would be a strong indication that the control values were supplied at the top of the hierarchy. Control values loaded separately for different parts of the hierarchy would be used if different control values were desired. In this example, the same control values are used at all four aggregate nodes, indicating that they were probably entered at the top of the hierarchy. The main point of defining this view is that you don't want your users to have to know the difference. Users can query the view without having to know anything about the sourcing of the metric aggregations nor exactly what controls have been defined.

Figure 16.16 provides a conceptual model for how the metric-control view is defined. You need to join the Metric Fact and Control Fact tables. This example illustrates the process for a single-dimension aggregation, but any number of dimensions can be aggregated simultaneously. For dimensions not being aggregated, the dimensional references can be set to the “~All~” row in the System subdimension.

The logic of this view involves finding metric facts and their related control facts, with the hierarchy in the Metric dimension determining the relationships (right side of Figure 16.16). For each resulting pair, the desired aggregation dimension is joined for each fact, looking for common ancestor entries within the same perspective in the dimension (left side of Figure 16.16). Each row returned through this logic is a valid row in the metric-control view. The value column in the view is set using a case statement to select the appropriate column from the Metric Fact table to match the control column value of the metric for the control values.

```

SELECT *,  

CASE  

    WHEN VALUE > UPPER_SPECIFICATION_LIMIT THEN 'Red'  

    WHEN VALUE > UPPER_DESIGN_LIMIT THEN 'Yellow'  

    WHEN VALUE < LOWER_SPECIFICATION_LIMIT THEN 'Red'  

    WHEN VALUE < LOWER_DESIGN LIMIT THEN 'Yellow'

```

```

        ELSE 'Green'
    END DISPLAY_COLOR
FROM (
    SELECT MF.FACT_ID           AS METRIC_FACT_ID,
           MC.FACT_ID          AS CONTROL_FACT_ID,
           H1.ANCESTOR_MASTER_ID AS DIAGNOSIS_GROUP_ID,
           CASE
               WHEN M2.CONTROL_COLUMN = 'Sum'      THEN MF.SUM
               WHEN M2.CONTROL_COLUMN = 'Mean'     THEN MF.MEAN
               WHEN M2.CONTROL_COLUMN = 'Median'   THEN MF.MEDIAN
               WHEN M2.CONTROL_COLUMN = 'Mode'     THEN MF.MODE
               WHEN M2.CONTROL_COLUMN = 'Minimum'  THEN MF.MINIMUM
               WHEN M2.CONTROL_COLUMN = 'Maximum'  THEN MF.MAXIMUM
               WHEN M2.CONTROL_COLUMN = 'Sample'   THEN MF.SAMPLE
               WHEN M2.CONTROL_COLUMN = 'Variance' THEN MF.VARIANCE
               WHEN M2.CONTROL_COLUMN = 'StdDev'   THEN MF.STDDEV
           END VALUE,
           CF.LOWER_SPECIFICATION_LIMIT,
           CF.LOWER_DESIGN_LIMIT,
           CF.TARGET_VALUE,
           CF.UPPER_DESIGN_LIMIT,
           CF.UPPER_SPECIFICATION_LIMIT
    FROM METRIC_FACT MF
    INNER JOIN METRIC_D M1 ON (MF.METRIC_MASTER_ID = M1.MASTER_ID)
    INNER JOIN METRIC_H MH ON (M1.ANCESTOR_MASTER_ID = M1.MASTER_ID)
    INNER JOIN METRIC_D M2 ON (MH.DESCENDENT_MASTER_ID = M2.MASTER_ID)
    INNER JOIN CONTROL_FACT CF ON (M2.MASTER_ID = CF.METRIC_MASTER_ID)
    INNER JOIN DIAGNOSIS_B B1 ON (MF.MASTER_ID = B1.MASTER_ID)
    INNER JOIN DIAGNOSIS_H H1 ON (B1.MASTER_ID = H1.DESCENDENT_MASTER_ID
                                  OR H1.DESCENDENT_MASTER_ID = 6)
    INNER JOIN DIAGNOSIS_B B2 ON (CF.MASTER_ID = B2.MASTER_ID)
    INNER JOIN DIAGNOSIS_H H2 ON (B2.MASTER_ID = H2.DESCENDENT_MASTER_ID
                                  OR H2.DESCENDENT_MASTER_ID = 6)
    WHERE H1.ANCESTOR_MASTER_ID = H2.ANCESTOR_MASTER_ID
          AND H1.PERSPECTIVE      = H2.PERSPECTIVE
);

```

The code for the view then wraps an additional query around the base query to set the display color column according to how the value column maps to the control values in the view. The example code sets the color as simple text, but you can set the column to an RGB setting if your business intelligence tool supports that usage.

Also notice in the sample code that the joins to the Hierarchy tables include an option to match on the All row in the System subdimension (Master ID = 6). This extra logic assures that there will always be a grand total aggregation set, even if the perspective in the actual dimension hierarchy doesn't include a single value at the top of the hierarchy. Displays of these data can always navigate and drill down from the All value.

Any view you define between the Metric Fact and Control Fact tables will return varying results depending upon the actual metrics and control loaded into the warehouse. If controls have been defined for multiple layers of the hierarchy, the view will return multiple rows for the same metric fact using each set of

control values. Likewise, if there are multiple perspectives in a dimension being aggregated, a different set of combined data will be returned for facts connecting to those entries that are referenced in more than one perspective. If inner joins are used, as in the sample code, there will not be any rows in the view for any Metric Fact values that have no corresponding control values. If you want something different than this, you can add appropriate logic to your view query. Left joins to the Control Fact table can allow Metrics Facts to appear with NULL controls when there are no controls loaded for a given metric or dimensional hierachic level. It is common to add filters to the query to limit the perspectives and even the levels of the hierarchy that appear in the view.

These metric queries are among the more complicated queries you'll need to create against the warehouse. I recommend that you experiment with them for a while before committing to a long-term strategy for delivering metrics and controls to users. This experimentation is particularly important if your organization is fairly new at defining and building metrics or if your metric program is generally immature. Start with only those metrics and controls that are actually being used in your institution. Meet those needs very specifically first and only then work to generalize the metric capability. For me, metrics are the newest piece of this design. I've implemented this functionality in several organizations, but I haven't yet made the process completely generic.

At whatever level of detail and generality you decide to implement metric support, along with the maximization of your sources discussed at the beginning of this chapter, you are now ready to start delivering data to your users. There are some additional query-support functions to be finalized in Chapter 17, and then you'll be ready to end this Gamma phase and move toward production.

Chapter 17

Delivering Data

Delivering data to users requires operationalizing the various data controls and features put into place in the last few chapters against the full Gamma version warehouse that is being loaded. The traditional view of data warehousing as giving users the ability to query and report data in the warehouse is true, but it's a generalization. More specifically, users are given access to subsets of information in the warehouse to which they have legitimate rights of access for well-defined purposes. What the data users see in the warehouse will often differ based on *why* they are looking. Satisfying all of these requirements involves taking advantage of features explicitly included in the warehouse design, as well as capabilities of the operating environment, particularly the database management system. Each warehouse implementation will be somewhat different based on differences in those environmental features, but the underlying requirements being addressed will remain common among the wide range of possible environmental implementations, many of which are unique to biomedicine.

Warehousing Use Cases

Stakeholders participate in your data warehousing program in different ways, each seeking to meet their own requirements that are likely to be much more concrete and specific than any list of general or aggregate requirements for a data warehouse. The warehouse is a tool, and it can be used in a wide variety of ways. Your effort involves making sure you've built a generalized toolset that can be used to meet the varying specific requirements of each stakeholder. These requirements can be grouped by the type of stakeholder that is expected to participate and can be defined for requirement purposes as high-level use cases.

User Analysts

The main outputs of the warehouse, produced for users typically known as analysts, include data marts and the data queries and reports that can be obtained from the warehouse (Figure 17.1). The distinction between whether an analyst queries the warehouse directly, or through some other mechanism to which a data mart has been transferred, is relevant in terms of technologies and tools but not in terms of what each analyst is trying to do. Every participant in your warehousing initiative, regardless of what other specialized roles they might also play, is in the role of user analyst at times. For many stakeholders, analyst is the only role they participate in. This role is the standard user participation that comes to mind when we think about data warehousing.

Whether directly or otherwise, an analyst needs access to the definitions of what has been placed in the warehouse. His or her objective is typically some combination of identifying cohorts to study as well as querying data about those cohorts. Some of that data access becomes routine enough to be defined into regularly scheduled reports. They provide feedback to other stakeholders in the warehouse system, typically both data owners and support staff, and receive training in the warehouse and its tools. In more advanced cases, individual analysts might annotate or curate data in the warehouse.

HIPAA Controller

A special case of data analysis in the warehouse is the role of privacy officer, or HIPAA controller. This role acts as an analyst in querying cohorts and data, but the focus is specialized and important enough to be noted separately. As a

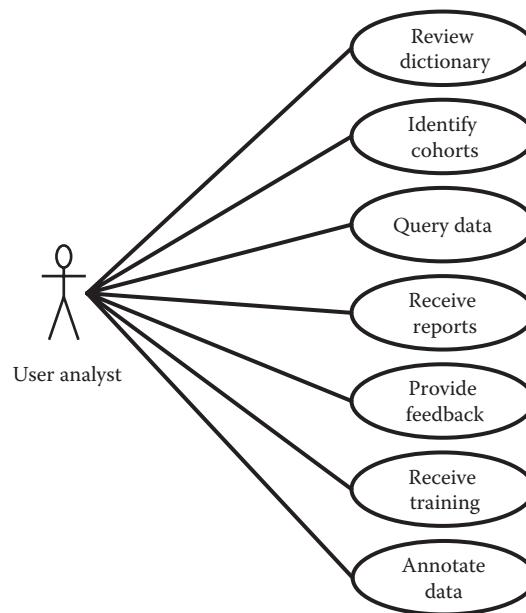


Figure 17.1 User analyst use cases.

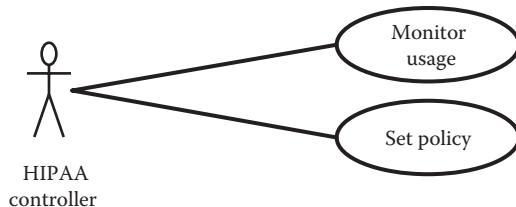


Figure 17.2 HIPAA controller use cases.

result of these queries, specialized control activities might be initiated with data owners and support staff to increase or improve the controls in place around the warehouse. This controller role sets policy for how data are defined and accessed relative to privacy requirements, a specialized annotation usage (Figure 17.2).

Data Owners

The data warehouse is responsible for storing and controlling the data placed within it, but it does not own that data. Ownership of the data rests with the myriad departments and teams throughout the institution that are functionally responsible for that data. The data warehouse acts as a steward of the data on behalf of those owners, so those owners play a pivotal role in the definition and use of their data within the warehouse. The warehouse outputs that go to the data owners include data issues that arise through the loading and use of the data that might require resolution from data owners, as well as identified data hypotheses that offer additional opportunities to data owners for analysis and understanding of their data (Figure 17.3).

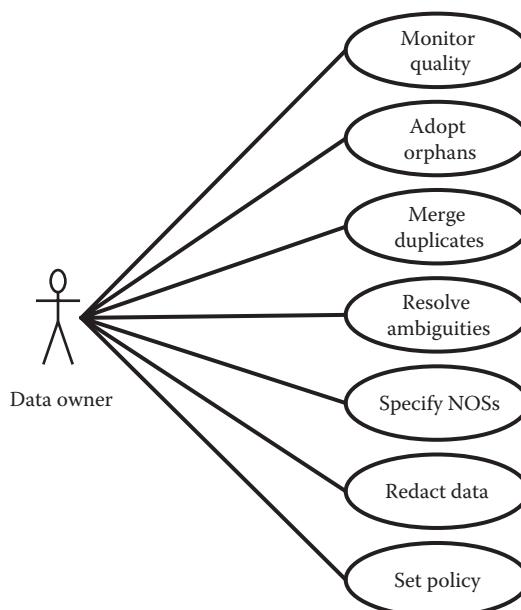


Figure 17.3 Data owner use cases.

Data owners monitor the quality of their data in the warehouse to ensure that it is accurately loaded and understood by any analyst that might use it. That effort includes the adoption of orphan data values that show up in their data from time to time, as well as the consolidation of duplicate synonyms that occur because of previously unrecognized data overlap across multiple source systems. To the extent that individual sources don't provide enough detail in the data to unambiguously differentiate it from similar data arriving from other systems, data owners work to reduce the ambiguity, including specifying values for unspecified data properties where the properties are not provided adequately by source application systems.

It is up to the data owners to decide what will be seen in the data warehouse and under what circumstances. Data can be redacted by data owners, rendering it not visible to user analysts. Overall, data owners set all policies related to the presence, visibility, and usage of data within their ownership, allowing the warehouse technology and staff to remain agnostic regarding what data are available and what they ultimately mean.

Data Governance

A special case of data ownership is the overall governance function around the warehouse. In a certain respect, all of the data owners are part of the governance function, but the governance function also goes beyond simply the aggregation of the effects of all of the data owners. The data governance function oversees the entire warehouse system and guides the behaviors of the data owners (Figure 17.4).

Data governance sets all of the policies that drive the content and behavior of the data warehouse and monitors and approves all exceptions to those policies. Their work includes prioritizing what data are added to the warehouse and in what order. This role is critical in the early years as different data across the institution competes for resources to be loaded into the warehouse. It can take 4–6 years before all institutional data are present in a large-scale warehouse, so the function that prioritizes that timing needs to have serious management clout and should definitely not reside in the data warehousing team.

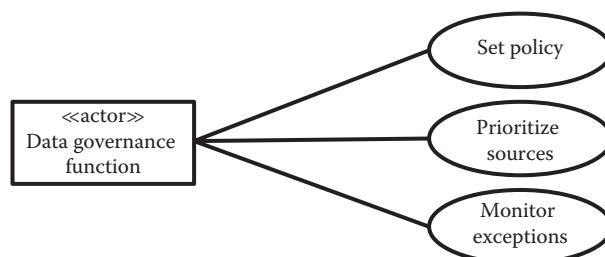


Figure 17.4 Data governance function use cases.

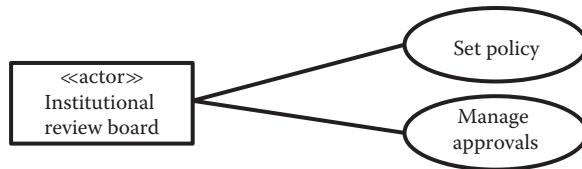


Figure 17.5 Institutional review board use cases.

Institutional Review Board

The organizations Institutional Review Board (IRB) owns behavioral access to the data warehouse, so is integrally involved in defining data elements and their access profiles (Figure 17.5). The IRB sets policy regarding what data can be accessed by what users and under what circumstances and manages all approvals of data access under those policies.

Support Teams

Actually delivering and providing a major data warehouse to an institution takes a lot of resource and involves many people carrying out numerous duties that make the system available and keep it running smoothly (Figure 17.6). To perform this function, these teams need considerable data from the warehouse in order to ascertain where opportunities to improve performance and control will be found.

The support team typically provides for the implementation of policies set by other stakeholders, including IRB approvals for access to data in the warehouse. They work to engage analytical users of the warehouse to assure that they

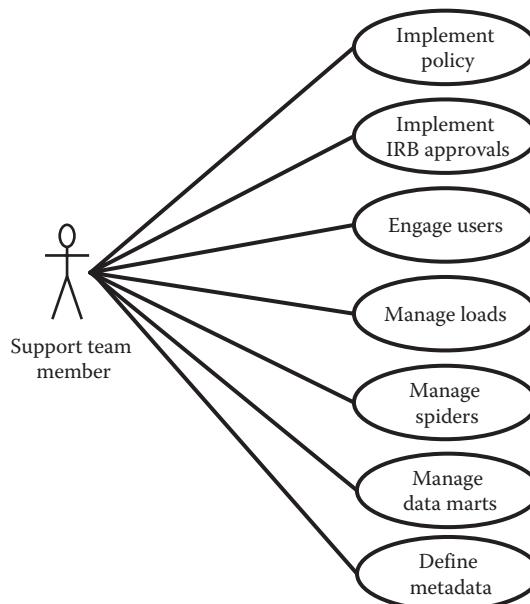


Figure 17.6 Support team member use cases.

receive adequate training and performance to meet their needs. They also manage the data loads that bring data into the warehouse, as well as the numerous spider processes that are typically always running in the warehouse environment. They also construct and provide the independent data marts that analysts need from time to time.

Data Administration

A special case of support resources is the data administrator, someone who specializes in managing data within and across an organization. These individuals typically have responsibility for data administration in the organization generally not just for the data warehouse. Their organizational role is to help ensure that data are well defined and implemented across the entire organization (Figure 17.7).

Typical data administration responsibilities include updating the data dictionaries that are used by analysts and data owners to understand the contents of the warehouse. They also monitor data loads to the warehouse looking for changes in patterns or frequencies of data that might indicate changing circumstances in the institution, monitoring the performance of the warehouse proactively in case changes start to seep in. The data administrators also typically act as warehouse liaison to the data governance bodies, often because they are uniquely situated to speak the languages of both the data warehousing staff and the organization's leadership team.

Auditor

Another critical support team is auditing. Auditors provide an independent control on the warehouse that management can review and communicate with to assure that the warehouse is performing its intended function and is being used according to policy (Figure 17.8). To do this, auditors typically evaluate new and

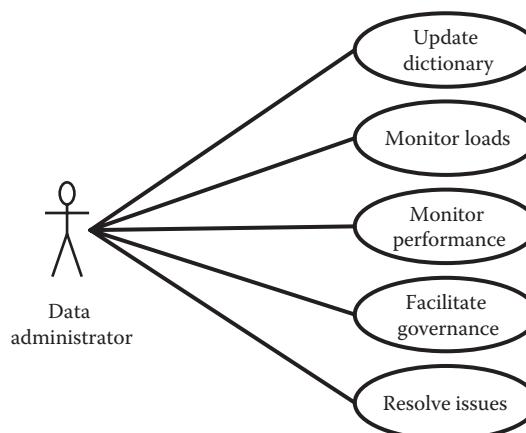


Figure 17.7 Data administrator use cases.

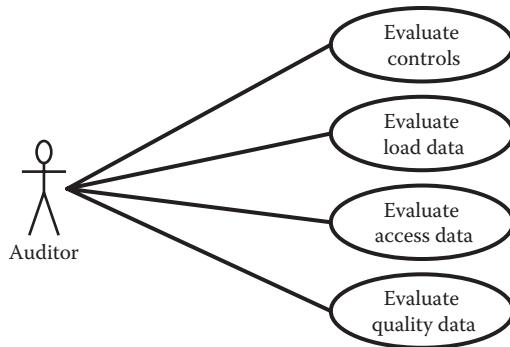


Figure 17.8 Auditor use cases.

existing controls in the data warehouse environment, evaluate all loaded data and query access for policy compliance, and thoroughly review and evaluate the quality of all data in the warehouse. Policy compliance issues are escalated to the governance group for prioritization and action.

Data Sources

Each of the stakeholders described earlier take on the role of both customer and supplier to the data warehouse. Data warehousing is a dynamic process precisely because the customers who receive the outputs of the system are generally also the suppliers providing inputs to the system. The exceptions to this dynamic are the data sources that provide data to the warehouse (Figure 17.9). The various data sources, typically application system data structures, provide both reference and factual data to the warehouse; acting in a passive manner and not typically receiving direct feedback from the warehouse system.

Standards Bodies

A special case of a data source stakeholder is a standards body (Figure 17.10), whether that body articulates data conventions for a sector or carries the weight of a regulator. Standards bodies promulgate the standards against which much of the data in a clinical data warehouse is categorized, so loading their reference data into the warehouse dimensions is an important aspect of creating a highly usable resource. The loading of standard reference data is often complicated by the need to license proprietary aspects of the standards that are involved, so institutions that carry enterprise licenses for this data often find the loading of

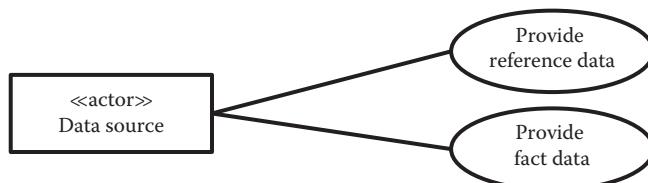


Figure 17.9 Data source use cases.

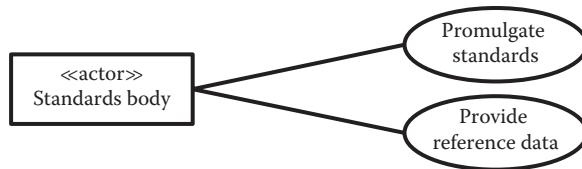


Figure 17.10 Standards body use cases.

standards a bit easier than those organizations that do not have direct licensed access to the needed reference data.

Privacy-Oriented Usage Profiles

Ultimately, delivering data to all warehouse stakeholders across the various warehouse use cases requires that you provide extensive functionality for access and control. Your business intelligence environment, at whatever level of maturity of those tools you have available in your organization, will provide the vast majority of user access. Presuming that your business intelligence suite provides some form of semantic layer to insulate users from the underlying technologies, your users won't be writing SQL code to obtain data but will instead write their queries in the specialized environments you provide. Beyond those standard tools, you will also probably provide certain specialized applications for accessing the warehouse for specific purposes. Most of the issues you'll deal with during your rollout will be independent of whether you rely on your general business intelligence tools or implement specialized applications. Most of the specific applications you build and provide will be in response to requirements that you've already attempted to support in your more general tools. The choice to implement additional applications depends more on your organization's profile and experience in building and servicing application systems than on any direct warehousing implications.

Regardless of whether a specific use case is to be satisfied in your general business intelligence environment, or through more customized or tailored applications, your stakeholders will need access to data in the warehouse for a variety of reasons, in a large number of contexts. A common element among these scenarios will be the need to support patient privacy considerations. All of the standard warehouse tools and custom applications must support patient privacy regulations in a manner, and meeting the requirements of each use case should be treated as secondary to that overall requirement. Every use case ultimately falls under some combination of three main access scenarios (Figure 17.11) that need to be considered when delivering data to users:

- *Deidentified access to data:* Data are provided to users in a manner so as to protect the identity of patients in any query result set, in compliance with the HIPAA Privacy Rule in the United States, and similar related regulations elsewhere. This deidentified access is the most commonly used path to the

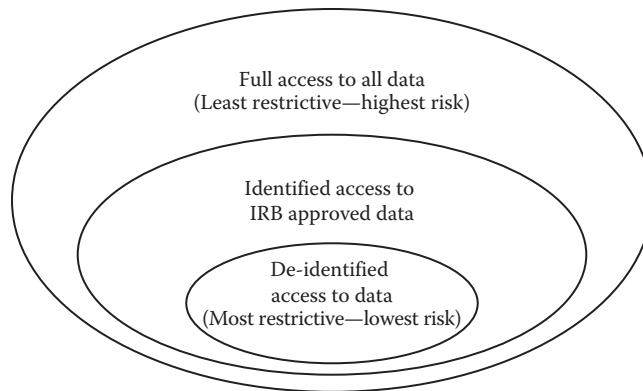


Figure 17.11 Layers of HIPAA-protected and IRB-controlled data access.

warehouse since even those users authorized to see identified data should be using deidentified access for queries that can be accomplished in that manner.

- **Identified access to IRB-approved data:** Data are provided to users who are authorized to have access by the IRB. IRB limitations typically include restrictions that must be enforced, particularly with respect to which patients are included in any approved cohort. Additionally, the IRB approval might only be for certain data during certain time periods.
- **Full access to all data:** Data are provided to users in an unrestricted mode. This full access is typically reserved for users in job functions that require broad access in an *ad hoc* manner, such as quality assurance staff or the information technology support team.

Since these three scenarios represent a decreasing level of data protection, it is important that users be trained to use the most restrictive access path available that meets the requirements for any particular access and that increased training is available to users who will have access through the less restrictive methods. It is not uncommon for queries to involve mixtures of these levels. For example, a researcher might be accessing IRB-approved data about a cohort of their research subjects while simultaneously comparing the cohort to a profile of similar patients obtained through deidentified access to a large population of patients. For example, a quality assurance analyst might use full access to investigate a problem with a particular medication use in the institution, but only after using the deidentified use case to define an *ad hoc* cohort of patients who received that medication during the time periods being investigated.

Designing for these scenarios presents challenges for the warehouse team. There is no single generic design that can be used to implement the levels of required controls. The technologies available vary too widely by organization to be able to recommend any implementation, but there are some commonalities that are frequently used as components of a solution. The full open access doesn't need to be implemented *per se*, so that particular scenario takes care of itself. You will provide a full access view of your warehouse within your business

intelligence layer, even though you'll need to prevent most users from having indiscriminant access to it.

Limited IRB access presents many challenges because you have to be prepared to protect any data at any time without knowing in advance exactly what the IRB might approve. Fortunately, those limited-access views are rarely needed on the warehouse implementation date, so you can defer them until later. Typically, you need to concentrate on the completely deidentified scenario since it is the most common. The deidentified access capability is more predictable than the full IRB view, so you can create it just once for all users. From there, you can back up and apply techniques used in the deidentified scenario to start solving the IRB-approved access problems.

Metadata Browsing

The first functionality any user of the data warehouse needs is the ability to browse through the metadata in the warehouse to gain a better understanding of what data the warehouse makes available. This capability can be included in the general business intelligence environment you supply to your users, but one day, you'll probably build a distinct application for searching and navigating the metadata. Whatever functionality you provide, it should be an integral part of user training.

Browsing metadata in the warehouse involves two levels of query capability: (1) browsing the Metadata dimension to see what data has been defined to the warehouse, and (2) browsing the Data Feed dimension to see what data has been loaded into the warehouse. Facts in the warehouse can be investigated through either of these two pathways. Dimensional data require investigation through both pathways because the link between dimension data values and the Metadata dimension is only implicit. Unlike facts that have an actual foreign key to the Metadata dimension, the connection between dimensions and the Metadata mappings is through the Data Feed dimension that links dimension rows to the sources of the data in those rows.

To plan the functionality of a metadata browser, the important concept to keep in mind is that a single row of metadata defines a link between a specific source column and its precise warehouse destination. It's important to keep that single-row perspective in mind when discussing metadata because that source-to-target linkage is the key to the entire generic ETL architecture you've implemented. As the one-to-one perspective broadens, users will begin to see the many-to-many mappings between complex data sources and the complex combinations of data they include in their queries, but the one-to-one perspective is where you should encourage them to start.

The rows of metadata can be approached from either side of their content: (1) look at all of the data defined in the warehouse and trace that data back to their various sources or (2) look at all the various sources of data for the

warehouse and trace that data forward to their warehouse destinations. Both approaches are valid and actually look at the same metadata. Most users start in the middle of these two views by querying the names you've given your metadata rows, names that are captured in the Source, Event, Entity, and Attribute columns. These four columns are populated by your data analysis team in order to provide user-oriented values for these four key concepts. Because they aren't part of the data structure of either the sources or the warehouse, the values aren't constrained by the names of objects in those environments. They can be any values your analysis team determines would be meaningful to your user community, and the names can be changed at any time if you identify a better way to describe data to support your users. You don't have as much flexibility with other aspects of the metadata where the values often are determined by what data columns are actually named in the various systems and databases involved.

Querying the metadata typically follows a logical path of starting broadly and narrowing to pursue an item of interest. In that respect, the first query I teach users to write is

```
SELECT DISTINCT SOURCE
  FROM METADATA_D
 ORDER BY SOURCE;
```

This query produces a list of source systems from which data are derived. Remember that the values of these fields is user oriented and analyst determined, so "source system" is a loose definition for whatever the analyst decides to place in this column. Typically, the list includes the clinical and operational systems used within the institution, as well as any outside sources of information that are being treated as data sources, even if the actual application systems responsible for those datasets are unknown. To give users a sense of scale in the metadata, these queries are often adjusted to provide counts:

```
SELECT SOURCE, COUNT(*)
  FROM METADATA_D
 GROUP BY SOURCE
 ORDER BY SOURCE;
```

This query gives the user the same list of sources as the first query, but the count of metadata entries for each gives the user a sense of scale regarding how much different metadata has been defined for each source. While the number of metadata entries isn't necessarily a true indicator of how much data exists in the warehouse from that source, the correlation is typically fairly strong: Big complicated sources with lots of data tend to need more metadata to define and map that data into the warehouse. The next layer of metadata browsing typically pursues either an Event (a verb) or Entity (a noun) orientation, with the verb-process orientation more common among

novice users. This deeper exploration can be accomplished for all sources or for a subset of sources of interest as seen in the earlier browsing:

```
SELECT SOURCE, EVENT, COUNT(*)
  FROM METADATA_D
 GROUP BY SOURCE, EVENT
 ORDER BY SOURCE, EVENT;
```

This process-oriented query provides a view into the types of data being received from each source system. For example, if the source is the institution's electronic health record (EHR), the event list might include Admission, Discharge, Transfer, Order, Administration, Accession, Result, and other clinical or administrative EHR events. Conversely, a user might be interested in a list of sources from which the warehouse receives data about each type of event:

```
SELECT EVENT, SOURCE, COUNT(*)
  FROM METADATA_D
 GROUP BY EVENT, SOURCE
 ORDER BY EVENT, SOURCE;
```

Users will mix and match these elements of metadata as they become accustomed to browsing these core columns of the metadata. For example, they might be more interested in seeing all of the main data constructs that are being impacted by data being received. We represent those elements in the Entity column:

```
SELECT ENTITY, EVENT, SOURCE, COUNT(*)
  FROM METADATA_D
 GROUP BY ENTITY, EVENT, SOURCE
 ORDER BY ENTITY, EVENT, SOURCE;
```

This query lists the entities of interest in the warehouse (e.g., Patient, Encounter, Provider, Medication, Facility, Procedure) along with the sourced events that provide data about these entities. By adding the Attribute column, users can see the actual full metadata names for the data of interest:

```
SELECT ENTITY, EVENT, ATTRIBUTE, SOURCE, COUNT(*)
  FROM METADATA_D
 GROUP BY ENTITY, EVENT, ATTRIBUTE, SOURCE
 ORDER BY ENTITY, EVENT, ATTRIBUTE, SOURCE;
```

This query results in the most detailed view users can get of the data in the sources and warehouse using these four columns. Some of the result rows will have counts greater than one, and that's normal because they represent data in the warehouse that can be sourced from more than one place or type of source dataset. If the analysis team assigns very detailed values to these four columns, the counts in this query will tend toward one. If looser or more generic values are assigned, the counts tend to go up as increasingly vague values apply across more data values.

Becoming familiar with the Source, Event, Entity, and Attribute information in the warehouse is a core purpose behind providing metadata browsing capability. These four columns are among the most common query filters that experienced users will use when defining queries against the warehouse, particularly for fact data. To get more detailed insights into the specific columns of information available, users will browse the warehouse *targeting* information in each metadata row:

```
SELECT TARGET_DIMENSION, TARGET_TABLE, TARGET_SUBDIMENSION, TARGET_COLUMN, COUNT(*)
  FROM METADATA_D D
  INNER JOIN METADATA_R R ON (D.MASTER_ID = R.MASTER_ID)
 GROUP BY TARGET_DIMENSION, TARGET_TABLE, TARGET_SUBDIMENSION, TARGET_COLUMN
 ORDER BY TARGET_DIMENSION, TARGET_TABLE, TARGET_SUBDIMENSION, TARGET_COLUMN;
```

This browsing query counts the entries of metadata that target each component of the warehouse. These columns define where data are loaded when processed by the generic ETL. By convention, the list of dimensions includes “Fact” even though the Fact table would not normally be considered a dimension. Additionally, the subdimension is NULL when the table isn’t a Definition table. There is an analogous browser query that looks at the *sourcing* side of each metadata entry:

```
SELECT PHYSICAL_DATASET, LOGICAL_DATASET, FACT_BREAK, SOURCE_COLUMN, COUNT(*)
  FROM METADATA_D D
  INNER JOIN METADATA_R R ON (D.MASTER_ID = R.MASTER_ID)
 GROUP BY PHYSICAL_DATASET, LOGICAL_DATASET, FACT_BREAK, SOURCE_COLUMN
 ORDER BY PHYSICAL_DATASET, LOGICAL_DATASET, FACT_BREAK, SOURCE_COLUMN;
```

The results of this query include every source column that arrives at the generic ETL job stream from any source. A user gains insight into the contents of the warehouse, and the sources of that data, by mixing and matching elements of these sourcing, defining, and targeting perspectives. For example, this query illustrates the subdimensions that are populated with data from each physical source:

```
SELECT TARGET_DIMENSION, TARGET_SUBDIMENSION, SOURCE, PHYSICAL_DATASET, COUNT(*)
  FROM METADATA_D D
  INNER JOIN METADATA_R R ON (D.MASTER_ID = R.MASTER_ID)
 GROUP BY TARGET_DIMENSION, TARGET_SUBDIMENSION, SOURCE, PHYSICAL_DATASET
 ORDER BY TARGET_DIMENSION, TARGET_SUBDIMENSION, SOURCE, PHYSICAL_DATASET;
```

Understanding what data are defined in the warehouse, and the sources of that data, is a key element of user training and the development of effective queries against the data. The more users understand the metadata for the system, the more they’ll be able to query and integrate the data in the warehouse to meet their analytic and reporting requirements. Once they’ve gained an understanding of the data available in the warehouse, they’ll begin querying that data. They’ll begin by identifying the cohort of patients that they are interested in, and they’ll build out their full queries around that cohort.

Cohort Identification

A cohort is a collection of subjects, typically patients. The starting point for every user query against the data warehouse is some cohort of interest. A cohort can be a simple selection based on general or simple criteria, such as all patients discharged in the last 60 days or all patients with certain diagnoses or who have had certain procedures. The definition of the cohort can become more complicated, such as all patients over 50 years of age who have been administered Metformin or Glipizide and have had A1C results in a certain range or all patients with a certain combination of diagnoses and procedures who have not been admitted in the past year. Time is almost always part of a cohort definition. Users rarely query the warehouse for information going back to the beginning of time in searching for patients: The selection of patients for cohorts is usually done with data from the recent past. Once patients are added to a cohort, users sometimes query data about those patients into the remote past, but that remote data are rarely part of the determination of membership in the cohort. The furthest back I've seen data queried as part of a cohort definition was to find a list of patients that had been admitted at least once in each of the last 10 years.

The core warehouse problem with cohort identification is that it is a form of data mining in which we seek information for ourselves, not for the direct benefit of the patients. Because the cohort question is a research question, IRB approval is needed to access much of the data we want to see about these patients, but it's difficult to apply for IRB approval without at least some preliminary access to that data. To help alleviate—but not completely eliminate—this problem, you should include some form of Cohort Query Tool (CQT) in your Gamma version. By definition, the CQT provides a capability with which users generate queries, the output of which are lists of qualified subjects: cohorts. The constraints built into the tool need to ensure that privacy cannot be violated, so no restrictions are placed on the use of the tool. No IRB approval is required. The cohorts defined in this tool are used to help with the IRB approval process, and upon that approval, the cohort membership definitions are then used to further qualify broader warehouse queries as approved by the IRB. Whether a CQT is implemented as a standalone application or as a combination of predefined queries in the organization's business intelligence toolset, the requirements are more or less the same: Data that are protected under privacy regulations can be included in query selection and filtering criteria, because the query tool is specifically designed not to return any of that data. The only returned data are a list of patients (typically Master IDs in the Subject dimension) or just a count of how many patients fit the provided criteria.

Suppose you're interested in defining a cohort of female patients over 50 years of age who have a diagnosis of diabetes and have received Metformin as part of their treatment ([Figure 17.12](#)). You'll be able to filter the Subject dimension on Gender fairly easily, and that dimension can also give you the Date of Birth of the patients, so their ages. Depending upon how other data were loaded into the warehouse, you might or might not have facts that

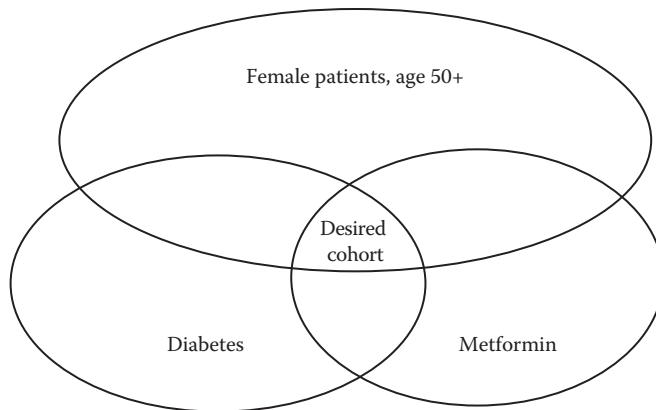


Figure 17.12 Cohort definition as intersection of dimensional filtering.

dimensionalize medications and diagnoses together. Presuming you don't, you recognize that the facts that give you diagnoses and medications are orthogonal—they are dimensionalized differently, so they have to be queried separately. You'll need two subqueries, one to find patients with diabetes, and the other to find patients given Metformin. Both subqueries can filter on Gender and Age to limit their result sets, and an `INNER JOIN` between the results of the subqueries will give a result set of patients who are both diabetic and have received Metformin.

The subqueries are unique but share some characteristics. In each case, you need to decide what facts will represent the information being sought. What factual metadata should be included that would indicate a diagnosis of diabetes? Most facts that include diagnosis would likely be relevant, including admission and discharge diagnoses and problem lists. Family history facts would probably be excluded. What about *working* diagnoses? Should they be included? The Metformin subquery presents the same challenges. Typically medication administrations would be included in the query. What if the only administration fact for Metformin is a not-taken event? Should orders for Metformin be included if administration data are missing? Should medication history recorded on admission be used? These questions affect whether or not a patient will be in your result set. The answers are dependent upon what data are actually loaded and available in the warehouse at any given time and on the user's understanding of the facts available in the warehouse as defined in the metadata.

You also have to be concerned with how these subqueries will qualify the information they are looking for. Metformin comes in several forms under over a dozen brand names, many in combination with other medications. How should the medication in the query be filtered? Assuming you loaded the RxNorm ontology into your Material dimension in such a way that it provides a pathway to the medications you loaded from your clinical systems, you could qualify your ancestor material on the Metformin concept in that ontology (Id 6809), and use the material hierarchy to retrieve all descendants. This would provide a result set that includes any of the medications defined as made with Metformin as part of the formulation.

What about qualifying diabetes in the diagnosis dimension? Depending upon what data have been loaded into the warehouse for diagnosis facts and ontologies, you might limit diagnosis to an ancestor of Diabetes Mellitus in ICD-9 (Id 250) and retrieve all descendent data, or you might filter on the SNOMED concept for Diabetes Mellitus (Id 191044006) knowing that the mappings in SNOMED will allow you to reach both ICD-9 (Id 250) data and ICD-10 (Id E08–E13) data. Keep in mind that the ontologies selected and available are only useful to the extent that your queries will find fact data dimensionalized to the descendent concepts that are reached through the ontologies.

The complexity of this thought process is a good indication of why these queries can be difficult to write generically. Note that the complexity isn't caused by the dimensional star architecture itself. The actual syntax of any final query might be complex because of the orthogonal nature of the facts involved, but even a more traditional query against a normalized relational warehouse would have the same problems. It is difficult to conceptualize some complicated questions. In fact, the introduction of ontologies makes the complexity of the queries in a relational warehouse much more complicated because of the extra normalized tables and relationships required for their mappings. Ontologies add no new tables or relationships to this star-based architecture, so they add no coding complexity to your star-based dimensional queries.

Once the fact metadata and dimensional filters have been specified, the two needed subqueries can be written. Each joins the Fact table to the metadata, qualifying the metadata in each case to the events that were selected in analysis; and to the Subject dimension, qualifying on the patient's gender and age. For consistency, the Subject dimension should also be filtered to the patient role in the bridge, because some of the facts returned might reference other roles and subjects. Each query also joins and filters the Diagnosis or Material dimension for the Diabetes or Metformin concept selected in analysis. Both of these subqueries are written to return a set of distinct Subject Master IDs. Next, the result sets of the two subqueries are INNER JOINed on the returned Master ID to reduce the result set to only those subjects found in the results of both subqueries. This result set is the desired cohort list. The final step in cohort querying is to wrap one last query around all of this code in order to count the rows in the cohort set. This final count is displayed to the user as the result of the cohort query.

There are other factors that might need to be built into this cohort query, but the essentials have been handled. The user might want to limit the factual queries to only active data. If the facts in the warehouse are loaded against hierarchies in the Subject dimension, the patient qualification would need to be on the Person subdimension as an ancestor, to ensure that Master IDs for specimens weren't returned instead of patients. The actual filters required will vary depending on the range and types of data loaded, and the complexity of any hierachic relationships that have been defined against the loaded data.

Timing aspects are also typically added to cohort queries. Depending upon why the cohort is being sought, it might not be useful to identify patients

not seen recently, or for whom the diagnosis and medication were not contemporaneous. At its simplest, a search might be limited to data in the past 2 years. If so, that filter is easily added to the Calendar dimension of each of the subqueries (or directly to the Source Timestamp on the fact, if that approach was selected during design). For more contemporaneous alignment, the subqueries could be written to return the Master ID of the Interaction dimension, so the joining of the subquery data would ensure that the findings occurred in the same encounter. More generally, the subqueries could be written to return fact timestamps so the subquery join would check to see that the diagnosis and medication were within 30 days of each other or that the diagnosis occurred before the medication.

The permutations of qualifications that might be needed in a CQT are almost limitless, so don't fall into the bottomless pit of trying to design for every eventuality because you'll fail. Stick to the more likely and common query logic, and take advantage of the generic nature of this star design to maximize what can be achieved with general logic. Here is what I consider the highlights of a generic query for cohorts:

- A cohort is always a list of distinct patients.
- Membership in a cohort is determined by facts associated with those patients.
- Facts are identified in subqueries by metadata and filtered by some combination of nonorthogonal dimensions, often a single dimension at a time.
- Subqueries always return the Subject Master ID along with any other data needed for any subsequent Boolean joins, typically including Interaction Master ID and Calendar Master ID at the appropriate temporal grain.
- Subquery result sets are combined using joins that implement And, Or, and Not conditions across the result sets, nesting subqueries to lower levels as required by the Boolean conditions needed for their recombination.

A few simple queries built into a toolset at the business intelligence layer of the warehouse implementation can embody these requirements, providing a flexible, efficient, and effective way for users to quickly identify cohorts of interest without needing to have anything more than deidentified access to the data in the warehouse. Cohorts can be saved in the data warehouse by inserting a single Cohort row into the Subject dimension and using the list of patient Master IDs to create hierarchy entries in the Subject dimension. Saving a cohort allows subsequent queries to be defined using cohort as the filtering criteria for the Subject dimension ancestor without needing to know who those patients are or having to rebuild and execute the cohort query. Remember that defining the cohort might have involved accessing data that are not directly available to the user in a deidentified view of the warehouse. Querying deidentified data *about* the cohort requires having the cohort defined *before* accessing the warehouse with deidentified queries.

Over time, users will become more familiar with the data in the warehouse, and their requirements for querying the data will become more mature and more complicated. A common adaptation I see in cohort searching is to add a two-stage model to the queries so a cohort definition can be split into research and control subgroups. An initial-stage search might define female patients over 50 years of age, who have been admitted in the past year and have a diagnosis of diabetes. A second-stage criteria could split this group into a research cohort that has received Metformin, and a control cohort that has not received Metformin. Subsequent queries could then use these two cohorts for data comparison and analysis.

Another commonly used cohort query strategy involves taking advantage of the CQT in the open source i2b2 platform. That query tool requires the data of interest to be loaded into the i2b2 database, which is severely restricting when compared to the general star in this implementation, but it can be done. I wouldn't recommend doing it just to use the cohort tool, but if you end up using i2b2 as part of your data mart delivery strategy, you'll probably include the CQT in your i2b2 usage.

Fact Count Queries

One of the earliest queries any new user of the warehouse should learn to create is the Fact Count by Metadata query. This simple query allows users to see all of the types of facts that exist in the warehouse for some domain or area in which they are interested. It's a great training query because it is very common for individual users not to be aware of all the different types of data in the warehouse. The basic query is a simple join of the Fact table and Metadata dimension in order to count facts by their metadata tags:

```
SELECT SOURCE, EVENT, ENTITY, ATTRIBUTE, COUNT(*)  
  FROM FACT  
 INNER JOIN METADATA_D META  
    ON FACT.METADATA_MASTER_ID = META.MASTER_ID  
 GROUP BY SOURCE, EVENT, ENTITY, ATTRIBUTE  
 ORDER BY SOURCE, EVENT, ENTITY, ATTRIBUTE;
```

This query can easily be adapted to provide a view into the facts associated with any particular area of interest. A common variation is to limit the fact count to only those patients previously identified in a cohort. If a cohort query filtered on a particular time period when it was defined—say the last 2 years—then the fact-count query can be filtered in the same way; so the user can quickly see what kinds of facts are available for a cohort or domain of interest. The types of facts available will often influence what kind of queries are designed and executed next, particularly if the user is surprised by an unanticipated presence or absence of data. A cohort of hundreds of patients could easily have tens of

thousands of facts—or more—and it can be very helpful to know what those fact types are before writing additional detailed queries.

Executing the fact count query for a cohort of patients illustrates the general model for querying the data in the warehouse that should be taught to all users: always start with a subquery to identify your cohort of interest and include the returned identifiers as one of your selection criteria in your main query:

```
SELECT SOURCE, EVENT, ENTITY, ATTRIBUTE, COUNT(*)
  FROM FACT
  INNER JOIN METADATA_D META ON (FACT.METADATA_MASTER_ID = META.MASTER_ID)
  INNER JOIN SUBJECT_B B ON (FACT.SUBJECT_GROUP_ID = B.GROUP_ID)
 WHERE B.MASTER_ID IN
    (SELECT DISTINCT MASTER_ID AS COHORT_MEMBER FROM
     (
      << Cohort Subquery Here >>
     )
    )
 GROUP BY SOURCE, EVENT, ENTITY, ATTRIBUTE
 ORDER BY SOURCE, EVENT, ENTITY, ATTRIBUTE;
```

If the functional capabilities of the implemented CQT include saving the resulting cohorts in the Subject dimension, the need to reproduce the cohort search query as a subquery is eliminated, instead allowing you to filter your main query using a join to the cohort definition:

```
SELECT SOURCE, EVENT, ENTITY, ATTRIBUTE, COUNT(*)
  FROM FACT
  INNER JOIN METADATA_D META ON (FACT.METADATA_MASTER_ID = META.MASTER_ID)
  INNER JOIN SUBJECT_B B ON (FACT.SUBJECT_GROUP_ID = B.GROUP_ID)
  INNER JOIN SUBJECT_H H ON (D.MASTER_ID = H.DESCENDENT_MASTER_ID)
  INNER JOIN SUBJECT_D D ON (H.ANCESTOR_MASTER_ID = D.MASTER_ID)
 WHERE D.SUBDIMENSION = 'Cohort' AND D.MASTER_ID = 123456
 GROUP BY SOURCE, EVENT, ENTITY, ATTRIBUTE
 ORDER BY SOURCE, EVENT, ENTITY, ATTRIBUTE;
```

The general fact count query is a common convention with which users can become familiar with the data contained in a cohort of interest. It defines a subset of the data warehouse that is of interest, a form of *ad hoc* data mart for analysis. If the user receives IRB approval for identified data access to the cohort, the range of data available against the cohort expands, and subsequent queries shift to an identified view of the data.

Timeline Generation

After a user has some idea of the types of facts that are available in an area of interest, the next step in anticipating query strategies is to look at some of the details of those facts or a selected subset of those facts that might be of interest at any particular time. The generic query I recommend for general exploration is

the timeline, a query in which facts are sorted by Time within Interaction within Subject. Each result row includes the fact type, value (possibly redacted) and unit of measure:

```

SELECT FACT.SUBJECT_GROUP_ID AS SUBJECT,
       FACT.INTERACTION_GROUP_ID AS INTERACTION,
       CALENDAR_D.SORT_DATE AS CALENDAR,
       CLOCK_D.SORT_TIME AS CLOCK,
       META.SUBJECT,
       META.EVENT,
       META ENTITY,
       META.ATTRIBUTE,
       CASE
           WHEN META.REDACTION_CONTROL_FLAG = 'N'
               THEN FACT.VALUE
           ELSE '~Redacted~'
       END AS VALUE,
       UOM_D.ABBREVIATION AS UOM

FROM FACT, METADATA_D META,
     UOM_D, UOM_B,
     CALENDAR_D, CALENDAR_B,
     CLOCK_D, CLOCK_B

WHERE FACT.METADATA_ID      = META.MASTER_ID
    AND FACT.CALENDAR_GROUP_ID = CALENDAR_B.GROUP_ID
    AND CALENDAR_B.MASTER_ID  = CALENDAR_D.MASTER_ID
    AND CALENDAR_B.ROLE       = 'Event'
    AND FACT.CLOCK_GROUP_ID   = CLOCK_B.GROUP_ID
    AND CLOCK_B.DIMENSION_ID = CLOCK_D.DIMENSION_ID
    AND CLOCK_B.ROLE         = 'Event'
    AND FACT.UOM_GROUP_ID    = UOM_B.GROUP_ID
    AND UOM_B.MASTER_ID      = UOM_D.MASTER_ID
    AND UOM_B.ROLE            = 'Value'

ORDER BY FACT.PATIENT_GROUP_ID,
        FACT.INTERACTION_GROUP_ID,
        CALENDAR_D.SORT_DATE,
        CLOCK_D.SORT_TIME,
        META.SUBJECT, META.EVENT, META.ENTITY, META.ATTRIBUTE;

```

Note that I cheated a little in that query. I'll explain why so you can choose your own strategy for developing queries quickly. I selected the Subject Group ID and Interaction Group ID as proxies for my Subject Master ID and Interaction Master ID because I know the groups created in those two dimensions only rarely have more than one bridge entry. I used the proxies to avoid the extra joins. I don't lose any facts through this shortcut, but I could inadvertently treat two Group IDs that resolve to the same Master ID as though they are distinct Subject or

Interaction entries. If this were a concern (i.e., if I expected many transplant-related facts), I'd fill in the missing joins to the rest of the Subject and Interaction dimension tables in order to resolve the distinct entries in the sort order.

Likewise, I didn't include a lot of extra dimensional qualifications that are needed to ensure correct dimension connections as the data get more complicated. When a query qualifies a bridge entry on a role, it is customary to also qualify the rank to ensure a single result. I omitted it because I know the roles I am qualifying on would not have more than one rank. I also omitted the qualification criteria to ensure that whenever I joined from a bridge to a definition, the query would only return either the current Type 2 variant of the definition or the variant active at the time of the fact. I know Type 2 changes occur only very rarely to the Calendar, Clock, and Unit of Measure (UOM) dimensions, and I believe I'd spot any Cartesian that showed up in the result as a double entry of facts.

Lastly, I didn't fully redact the Fact value. I blocked the value if the redaction flag in the metadata was on because that means someone in the governance structure decided to protect the value. I also should have redacted the value if its length was excessive (as defined by the governance function) since I don't know what patient-identifying data might exist in the fact value. The Redaction Control Flag should be on for longer text values in the UOM dimension for controlling values longer than the threshold set by the governance group (e.g., 15 characters in the code example). Here's the corrected query:

```

SELECT FACT SUBJECT_GROUP_ID AS SUBJECT,
       SUBJECT_B.ROLE AS SUBJECT_ROLE,
       FACT INTERACTION_MASTER_ID AS INTERACTION,
       INTERACTION_B.ROLE AS INTERACTION_ROLE,
       CALENDAR_D.SORT_DATE AS DATE,
       CLOCK_D.SORT_TIME AS TIME,
       META SUBJECT,
       META EVENT,
       META ENTITY,
       META ATTRIBUTE,
       CASE
           WHEN META.REDACTION_CONTROL_FLAG = 'N'
               THEN WHEN UOM_D.REDACTION_CONTROL_FLAG = 'Y'
                   AND LENGTH(FACT.VALUE) > 15
                       THEN '~Redacted~'
                   ELSE FACT.VALUE
               ELSE '~Redacted~'
           END AS VALUE,
       UOM_D.ABBREVIATION AS UOM
FROM FACT, METADATA_D META,
     SUBJECT_B, INTERACTION_B,
```

```

UOM_D, UOM_B,
CALENDAR_D, CALENDAR_B,
CLOCK_D, CLOCK_B

WHERE FACT.METADATA_ID
    AND FACT.SUBJECT_GROUP_ID
    AND FACT.INTERACTION_GROUP_ID
    AND FACT.CALENDAR_GROUP_ID
    AND CALENDAR_B.MASTER_ID
    AND CALENDAR_B.ROLE
AND CALENDAR_B.RANK
    AND CALENDAR_D.STATUS_INDICATOR
    AND FACT.CLOCK_GROUP_ID
    AND CLOCK_B.DIMENSION_ID
    AND CLOCK_B.ROLE
AND CLOCK_B.RANK
    AND CLOCK_D.STATUS_INDICATOR
    AND FACT.UOM_GROUP_ID
    AND UOM_B.MASTER_ID
    AND UOM_B.ROLE
AND UOM_B.RANK
    AND UOM_D.STATUS_INDICATOR
        = META.MASTER_ID
        = SUBJECT_B.GROUP_ID
        = INTERACTION_B.GROUP_ID
        = CALENDAR_B.GROUP_ID
        = CALENDAR_D.MASTER_ID
        = 'Event'
        = 1
        = 'A'
        = CLOCK_B.GROUP_ID
        = CLOCK_D.DIMENSION_ID
        = 'Event'
        = 1
        = 'A'
        = UOM_B.GROUP_ID
        = UOM_D.MASTER_ID
        = 'Value'
        = 1
        = 'A'

ORDER BY SUBJECT_B.MASTER_ID, INTERACTION_B.MASTER_ID,
CALENDAR_D.SORT_DATE, CLOCK_D.SORT_TIME,
META.SUBJECT, META.EVENT, META ENTITY, META.ATTRIBUTE;

```

This corrected query is marginally more complicated than the earlier version, but that added complexity is straightforward and predictable. I only take shortcuts to save time when I'm typing queries manually, knowing that the reporting layer of my business intelligence toolset will eventually take care of most of that extra connections and controls for me in production. The general timeline query is intended to provide a quick snapshot of the facts that are associated with some area of interest. Let's take a look at an execution of the basic query that returns 52 facts from a single patient encounter ([Figure 17.13](#)). This patient was preadmitted for gangrene, probably by a phone call from his personal physician, just after 5:00 p.m. on January 31, 2011, with an expected arrival the next morning. The following evening, the patient was admitted as an inpatient with an admitting diagnosis of cellulitis. Over the next few hours, laboratory results and vital signs were recorded for the patient. Medications were ordered and administered after midnight, and a charge showed up in the SAP finance system about an hour later. This is only a fragment of the data that might be returned by the basic timeline query, but notice its richness. A great deal can be discerned about the patient, clinically and operationally, by looking at these simple data.

A user who receives this kind of result will typically want to refine the query to see a bit more detail. This is usually done by adding some dimensions to

1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Expected Arrival Date	02/1/2011	DATE
2/1/2011	18:40	Cerner	Admission Inpatient (A01)	Patient Visit (PV2)	Admit Reason	CELLULITIS	LONG TEXT
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	52	MM HG
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	140	MEQ/L
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	36	MEQ/L
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	25	MM HG
2/1/2011	21:19	Quest	Specimen Received	Lab Test	Factless	~Factless~	Factless
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	6.4	%
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	82.8	%
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	1.0	x10 3/uL
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	16.1	%
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	3.77	x10 6/uL
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.00	%
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	8.4	x10 3/uL
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	10.1	x10 3/uL
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	11.9	G/DL
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.4	%
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.3	%
2/1/2011	22:27	Quest	Final Result	Lab Test	Clinical Result	33.5	%
2/1/2014	22:46	Quest	Final Result	Lab Test	Clinical Result	5.9	MG/DL
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	51	SECONDS
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	25.6	SECONDS
2/1/2014	23:39	Cerner	Allergy Assessment	No Known Allergy	Factless	~Factless~	Factless
2/1/2014	23:46	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	7	PAIN NOW
2/1/2014	23:47	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	20	BREATHES PER M
2/1/2014	23:48	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	YES	YES/NO
2/1/2014	23:49	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	79	BEATS PER MINU
2/1/2014	23:50	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	72	MM HG
2/1/2014	23:51	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	2 MODERATE REL	WITH MEDICATION
2/1/2014	23:52	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	35.9	oC
2/1/2014	23:53	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	136	MM HG
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	150	MG
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	25	MG
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	0.4	MG
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	40	MG
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG
2/2/2014	00:21	Cerner	Medication Order	New Order	Medication Route	30	MG
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	150	MG
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	25	MG
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	0.4	MG
2/2/2014	00:32	Cerner	Medication Order	New Order	Requested Give Unit	5	MG
2/2/2014	00:34	Cerner	Medication Administration	Given	Administered Unit	5	MG
2/2/2014	00:46	Cerner	Medication Order	New Order	Requested Give Unit	2	TABS
2/2/2014	00:51	Cerner	Medication Administration	Not Given	Administered Unit	5	MG
2/2/2014	01:03	Cerner	Medication Administration	Given via PYSIS	Administered Unit	2	TABS
2/2/2014	01:45	SAP	Charge	Unbilled - Inpatient	Charge Amount	38.00	US DOLLAR
2/2/2014	01:45	SAP	Charge	Unbilled - Inpatient	Charge Amount	124.50	US DOLLAR
2/2/2014	03:12	Cerner	Lab Test	New Order	Factless	~Factless~	Factless

Figure 17.13 Generic patient timeline.

the query in order to see more of the clinical context within which these facts occurred. This can be done by adding all the dimensions of interest to the query at once or by selectively adding one or more dimensions at a time, to explore details of interest.

Suppose the user, looking at the basic facts, notes there are several laboratory results displayed. To discover what these results were measuring, the query could be executed again after adding a join to the Procedure dimension. The results might look like the data shown in Figure 17.14. This result includes the Procedure description for every fact. Some of the facts list the procedure as Not Applicable because the fact wasn't dimensionalized against that dimension. The user might then filter out those Not Applicable entries and add a join to the Quality

Date	Time	Subject	Event	Entity	Attribute	Value	Units	Procedure Description
1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Admit Reason	GANGRENE	LONG TEXT	-Not Applicable-
1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Expected Arrival Date	02/1/2011	DATE	-Not Applicable-
2/1/2011	18:40	Cerner	Admission Inpatient (A01)	Patient Visit (PV2)	Admit Reason	CELLULITIS	LONG TEXT	-Not Applicable-
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	52	MM HG	pCO2 - VEN (POCT) [40-50]
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	140	MEQ/L	WB NA - VEN (POCT) [135-145]
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	36	MEQ/L	HCO3 - VEN (POCT) [20-27]
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	25	MM HG	pO2 - VEN (POCT) [20-25]
2/1/2011	21:19	Quest	Specimen Received	Lab Test	Factless	-Factless-	Factless	-Not Applicable-
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	6.4	%	MONOCYTE % [2.0-11.0]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	82.8	%	NEUTROPHIL % [40.0-74.0]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	1.0	x10 3/uL	LYMPHOCYTE # [1.0-4.5]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	16.1	%	RED DISTRIB. WIDTH [11.5-14.5]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	3.77	x10 6/uL	RED BLOOD CELL [4.50-6.00]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.00	%	NUCLEATE RBC% [0.0-0.0]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	8.4	x10 3/uL	NEUTROPHIL # [1.9-8.0]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	10.1	x10 3/uL	WHITE BLOOD CELL [5.0-11.0]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	11.9	MG/DL	HEMOGLOBIN [13.9-16.3]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.4	%	EOSINOPHIL % [1.0-7.0]
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.3	%	BASOPHIL % [0.0-1.0]
2/1/2011	22:27	Quest	Final Result	Lab Test	Clinical Result	33.5	%	HEMATOCRIT [42.0-55.0]
2/1/2014	22:46	Quest	Final Result	Lab Test	Clinical Result	5.9	MG/DL	WB CREATININE - VEN [0.7-1.5]
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	51	SECONDS	APTT - SL [21-34]
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	25.6	SECONDS	PROTHROMBIN TIME - SL [13.0-15.0]
2/1/2014	23:39	Cerner	Allergy Assessment	No Known Allergy	Factless	-Factless-	Factless	-Not Applicable-
2/1/2014	23:46	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	7	PAIN NOW	PAIN NOW
2/1/2014	23:47	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	20	BREATHES PER M Respirations	
2/1/2014	23:48	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	YES	YES/NO	PAIN LEVEL IS ACCEPTABLE TO PATIENT
2/1/2014	23:49	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	79	BEATS PER MINU	Pulse, Radial
2/1/2014	23:50	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	72	MM HG	Diastolic Blood Pressure
2/1/2014	23:51	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	2	MODERATE REL WITH MEDICATION	WITH MEDICATION
2/1/2014	23:52	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	35.9	oC	Temperature, Oral
2/1/2014	23:53	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	136	MM HG	Systolic Blood Pressure
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	-Not Applicable-
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	150	MG	-Not Applicable-
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	25	MG	-Not Applicable-
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	0.4	MG	-Not Applicable-
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	40	MG	-Not Applicable-
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	-Not Applicable-
2/2/2014	00:21	Cerner	Medication Order	New Order	Medication Route	30	MG	-Not Applicable-
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	150	MG	-Not Applicable-
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	25	MG	-Not Applicable-
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	0.4	MG	-Not Applicable-
2/2/2014	00:32	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	-Not Applicable-
2/2/2014	00:34	Cerner	Medication Administration	Given	Administered Unit	5	MG	-Not Applicable-
2/2/2014	00:46	Cerner	Medication Order	New Order	Requested Give Unit	2	TABS	-Not Applicable-
2/2/2014	00:51	Cerner	Medication Administration	Not Given	Administered Unit	6	MG	-Not Applicable-
2/2/2014	01:03	Cerner	Medication Administration	Given via PYSISIS	Administered Unit	2	TABS	-Not Applicable-
2/2/2014	01:45	SAP	Charge	Unbilled - Inpatient	Charge Amount	38.00	US DOLLAR	-Not Applicable-
2/2/2014	01:45	SAP	Charge	Unbilled - Inpatient	Charge Amount	124.50	US DOLLAR	-Not Applicable-
2/2/2014	03:12	Cerner	Lab Test	New Order	Factless	-Factless-	Factless	PROTHROMBIN TIME
2/2/2014	03:40	Quest	Lab Test	Specimen Received	Factless	-Factless-	Factless	PROTHROMBIN TIME
2/2/2014	04:10	Quest	Lab Test	Final Result	Clinical Result	21.7	SECONDS	PROTHROMBIN TIME - SL [13.0-15.0]

Figure 17.14 Patient timeline with Procedure descriptions.

Date	Tim	Subje	Event	Entity	Attribute	Value	Units	Procedure Description	Quality Descript
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	52	MM HG	pCO2 - VEN (POCT) [40-50]	Above high normal
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	140	MEQ/L	WB NA - VEN (POCT) [135-145]	Normal
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	36	MEQ/L	HCO3 - VEN (POCT) [20-27]	Above high normal
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	25	MM HG	pO2 - VEN (POCT) [20-25]	Normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	6.4	%	MONOCYTE % [2.0-11.0]	Normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	82.8	%	NEUTROPHIL % [40.0-74.0]	Above high normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	1.0	x10 3/uL	LYMPHOCYTE # [1.0-4.5]	Normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	16.1	%	RED DISTRIB. WIDTH [11.5-14.5]	Above high normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	3.77	x10 6/uL	RED BLOOD CELL [4.50-6.00]	Below low normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.00	%	NUCLEATE RBC% [0.0-0.0]	Normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	8.4	x10 3/uL	NEUTROPHIL # [1.9-8.0]	Above high normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	10.1	x10 3/uL	WHITE BLOOD CELL [5.0-11.0]	Normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	11.9	G/dL	HEMOGLOBIN [13.9-16.3]	Below low normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.4	%	EOSINOPHIL % [1.0-7.0]	Below low normal
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.3	%	BASOPHIL % [0.0-1.0]	Normal
2/1/2011	22:27	Quest	Final Result	Lab Test	Clinical Result	33.5	%	HEMATOCRIT [42.0-55.0]	Below low normal
2/1/2014	22:46	Quest	Final Result	Lab Test	Clinical Result	5.9	MG/dL	WB CREATININE - VEN [0.7-1.5]	Above high normal
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	51	SECONDS	APTT - SL [21-34]	Above high normal
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	25.6	SECONDS	PROTHROMBIN TIME - SL [13.0-15.0]	Above high normal
2/1/2014	23:46	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	7	PAIN NOW	PAIN NOW	~Not Applicable~
2/1/2014	23:47	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	20	BREATHES PER M/Respirations		~Not Applicable~
2/1/2014	23:48	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	YES	YES/NO	PAIN LEVEL IS ACCEPTABLE TO PATIEN	~Not Applicable~
2/1/2014	23:49	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	79	BEATS PER MINU	Pulse, Radial	~Not Applicable~
2/1/2014	23:50	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	72	MM HG	Diastolic Blood Pressure	~Not Applicable~
2/1/2014	23:51	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	2	MODERATE RELWITH MEDICATION WITH MEDICATION		~Not Applicable~
2/1/2014	23:52	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	35.9	oC	Temperature, Oral	~Not Applicable~
2/1/2014	23:53	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	136	MM HG	Systolic Blood Pressure	~Not Applicable~
2/2/2014	03:12	Cerner	Lab Test	New Order	Factless	~Factless~	Factless	PROTHROMBIN TIME	~Not Applicable~
2/2/2014	03:40	Quest	Lab Test	Specimen Received	Factless	~Factless~	Factless	PROTHROMBIN TIME	~Not Applicable~
2/2/2014	04:10	Quest	Lab Test	Final Result	Clinical Result	21.7	SECONDS	PROTHROMBIN TIME - SL [13.0-15.0]	Above high normal

Figure 17.15 Patient timeline with Procedure and Quality descriptions.

dimension to see if the facts included additional qualitative entries (Figure 17.15). These new results include an Abnormalcy indicator that was loaded as a Fact quality against the laboratory tests, although an experienced clinician could predict those values using the Reference Range values included in the Procedure descriptions.

As with the lab results, the user might also note the presence of medication facts in the original query and want to rerun the query with a join to the Material dimension to see what medications were involved (Figure 17.16). This display provides the medication names involved in those medication order and administration facts. As with the lab result, many of the facts don't have a material associated with them and are listed as Not Applicable in the material description. The user could filter these out and once again look for qualitative data on these facts in the Quality dimension (Figure 17.17). The user can now see the medication route associated with each medication fact. Lastly, the user might be interested in knowing more about the diagnosis recorded on admission. The query can be rerun with a join to diagnosis for those facts (Figure 17.18). This query shows that the ICD-9 440.24 diagnosis recorded as part of the preadmit process was changed to ICD-9 682.9 on admission.

All of these data, including the Procedure, Material, and Diagnosis dimensions, could have been added to the base timeline query at once. The result set gets very wide, and there are lots of values that appear as Not Applicable, but it gives a very comprehensive view of the facts all at once. Some users will jump to that full query very quickly. Other users will want to explore the data a bit more and will choose this incremental approach. Both approaches work and will serve users well. In the business intelligence environment, either query approach will

Date	Time	Subject	Event	Entity	Attribute	Value	Units	Material Description
1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Admit Reason	GANGRENE	LONG TEXT	~Not Applicable~
1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Expected Arrival Date	02/1/2011	DATE	~Not Applicable~
2/1/2011	18:40	Cerner	Admission Inpatient (A01)	Patient Visit (PV2)	Admit Reason	CELLULITIS	LONG TEXT	~Not Applicable~
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	52	MM HG	~Not Applicable~
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	140	MEQ/L	~Not Applicable~
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	36	MEQ/L	~Not Applicable~
2/1/2011	21:16	Quest	Final Result	Lab Test	Clinical Result	25	MM HG	~Not Applicable~
2/1/2011	21:19	Quest	Specimen Received	Lab Test	Factless	~Factless~	Factless	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	6.4	%	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	82.8	%	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	1.0	x10 3/uL	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	16.1	%	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	3.77	x10 6/uL	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.00	%	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	8.4	x10 3/uL	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	10.1	x10 3/uL	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	11.9	GR/DL	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.4	%	~Not Applicable~
2/1/2011	22:25	Quest	Final Result	Lab Test	Clinical Result	0.3	%	~Not Applicable~
2/1/2011	22:27	Quest	Final Result	Lab Test	Clinical Result	33.5	%	~Not Applicable~
2/1/2014	22:46	Quest	Final Result	Lab Test	Clinical Result	5.9	MG/DL	~Not Applicable~
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	51	SECONDS	~Not Applicable~
2/1/2014	23:07	Quest	Final Result	Lab Test	Clinical Result	25.6	SECONDS	~Not Applicable~
2/1/2014	23:39	Cerner	Allergy Assessment	No Known Allergy	Factless	~Factless~	Factless	~Not Applicable~
2/1/2014	23:46	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	7	PAIN NOW	~Not Applicable~
2/1/2014	23:47	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	20	BREATHE PER MIN	~Not Applicable~
2/1/2014	23:48	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	YES	YES/NO	~Not Applicable~
2/1/2014	23:49	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	79	BEATS PER MINU	~Not Applicable~
2/1/2014	23:50	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	72	MM HG	~Not Applicable~
2/1/2014	23:51	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	2 MODERATE REL WITH MEDICATION	~Not Applicable~	
2/1/2014	23:52	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	35.9	oC	~Not Applicable~
2/1/2014	23:53	Cerner	Vital Sign (RAS X02)	Final Result	Clinical Result	136	MM HG	~Not Applicable~
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	GLIPIZIDE XL TAB
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	150	MG	METOPROLOL XL TABS
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	25	MG	CAPTOPRIL TAB
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	0.4	MG	TAMSULOSIN CAP
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	40	MG	ESOMEPRAZOLE CAP
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	FINASTERIDE TAB
2/2/2014	00:21	Cerner	Medication Order	New Order	Medication Route	30	MG	CINACALCET TAB
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	150	MG	METOPROLOL XL TABS
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	25	MG	CAPTOPRIL TAB
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	0.4	MG	TAMSULOSIN CAP
2/2/2014	00:32	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	METOPROLOL TARTRATE INJ
2/2/2014	00:34	Cerner	Medication Administration	Given	Administered Unit	5	MG	FINASTERIDE TAB
2/2/2014	00:46	Cerner	Medication Order	New Order	Requested Give Unit	2	TABS	OXYCODONE/ACETAMINOPHEN 5/325
2/2/2014	00:51	Cerner	Medication Administration	Not Given	Administered Unit	5	MG	METOPROLOL TARTRATE INJ
2/2/2014	01:03	Cerner	Medication Administration	Given via PYSISIS	Administered Unit	2	TABS	OXYCODONE/ACETAMINOPHEN 5/325
2/2/2014	01:45	SAP	Charge	Unbilled - Inpatient	Charge Amount	38.00	US DOLLAR	~Not Applicable~
2/2/2014	01:45	SAP	Charge	Unbilled - Inpatient	Charge Amount	124.50	US DOLLAR	~Not Applicable~
2/2/2014	03:12	Cerner	Lab Test	New Order	Factless	~Factless~	Factless	~Not Applicable~
2/2/2014	03:40	Quest	Lab Test	Specimen Received	Factless	~Factless~	Factless	~Not Applicable~
2/2/2014	04:10	Quest	Lab Test	Final Result	Clinical Result	21.7	SECONDS	~Not Applicable~

Figure 17.16 Patient timeline with Material descriptions.

Date	Time	Subject	Event	Entity	Attribute	Value	Units	Material Description	Quality Description
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	GLIPIZIDE XL TAB	PO Daily
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	150	MG	METOPROLOL XL TABS	PO
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	25	MG	CAPTOPRIL TAB	PO BID
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	0.4	MG	TAMSULOSIN CAP	PO Daily
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	40	MG	ESOMEPRAZOLE CAP	PO Daily
2/2/2014	00:21	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	FINASTERIDE TAB	PO Daily
2/2/2014	00:21	Cerner	Medication Order	New Order	Medication Route	30	MG	CINACALCET TAB	PO Daily
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	150	MG	METOPROLOL XL TABS	PO GIV
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	25	MG	CAPTOPRIL TAB	PO GIV
2/2/2014	00:28	Cerner	Medication Administration	Given	Administered Unit	0.4	MG	TAMSULOSIN CAP	PO GIV
2/2/2014	00:32	Cerner	Medication Order	New Order	Requested Give Unit	5	MG	METOPROLOL TARTRATE INJ/IV Push STAT	
2/2/2014	00:34	Cerner	Medication Administration	Given	Administered Unit	5	MG	FINASTERIDE TAB	IV PUSH
2/2/2014	00:46	Cerner	Medication Order	New Order	Requested Give Unit	2	TABS	OXYCODONE/ACETAMINOPH Q4H	
2/2/2014	00:51	Cerner	Medication Administration	Not Given	Administered Unit	5	MG	METOPROLOL TARTRATE INJ/NOT GIV	NPO
2/2/2014	01:03	Cerner	Medication Administration	Given via PYSISIS	Administered Unit	2	TABS	OXYCODONE/ACETAMINOPH PO	

Figure 17.17 Patient timeline with Material and Quality descriptions.

Date	Tim	Subje	Event	Entity	Attribute	Value	Units	Diag_Ver	Diagnosis Description
1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Admit Reason	GANGRENE	LONG TEXT	440.24	ATHEROSCLEROSIS OF NATIVE ARTERIES OF THE EXTREMITIES WITH GANGREN
1/31/2011	17:10	Cerner	Pre-admit Patient (A05)	Patient Visit (PV2)	Expected Arrival Date	02/1/2011	DATE	440.24	ATHEROSCLEROSIS OF NATIVE ARTERIES OF THE EXTREMITIES WITH GANGREN
2/1/2011	18:40	Cerner	Admission Inpatient (A01)	Patient Visit (PV2)	Admit Reason	CELLULITIS	LONG TEXT	682.9	CELLULITIS AND ABSCESS OF UNSPECIFIED SITES

Figure 17.18 Patient ADT timeline with Diagnosis descriptions.

only take a few seconds to carry out, the differences being only when to drag in a few extra descriptive objects. The query that the toolset generates will take this form:

```

SELECT FACT.SUBJECT_MASTER_ID AS SUBJECT,
       SUBJECT_B.ROLE AS SUBJECT_ROLE,
       FACT.INTERACTION_MASTER_ID AS INTERACTION,
       INTERACTION_B.ROLE AS INTERACTION_ROLE
      ,CALENDAR_D.SORT_DATE AS DATE,
       CLOCK_D.SORT_TIME AS TIME,
       META.SUBJECT,
       META.EVENT,
       META ENTITY,
       META.ATTRIBUTE,
       CASE
          WHEN META.REDACTION_CONTROL_FLAG = 'N'
              THEN WHEN UOM_D.REDACTION_CONTROL_FLAG = 'Y'
                  AND LENGTH(FACT.VALUE) > 15
                  THEN '~Redacted~'
                  ELSE FACT.VALUE
          ELSE '~Redacted~'
       END AS VALUE,
       UOM_D.ABBREVIATION AS UOM,
       CASE
          WHEN PROCEDURE_D.REDACTION_CONTROL_FLAG = 'N'
              THEN PROCEDURE_D.DESCRIPTION
              ELSE '~Redacted~' AS PROCEDURE_DESCRIPTION,
       CASE
          WHEN MATERIAL_D.REDACTION_CONTROL_FLAG = 'N'
              THEN QUALITY_D.DESCRIPTION
              ELSE '~Redacted~' AS MATERIAL_DESCRIPTION,
       CASE
          WHEN QUALITY_D.REDACTION_CONTROL_FLAG = 'N'
              THEN QUALITY_D.DESCRIPTION
              ELSE '~Redacted~' AS QUALITY_DESCRIPTION,
       CASE
          WHEN DIAGNOSIS_D.REDACTION_CONTROL_FLAG = 'N'
              THEN DIAGNOSIS_D.VERNACULAR_CODE
              ELSE '~Redacted~' AS DIAGNOSIS_CODE,
       CASE
          WHEN DIAGNOSIS_D.REDACTION_CONTROL_FLAG = 'N'
              THEN DIAGNOSIS_D.DESCRIPTION
              ELSE '~Redacted~' AS DIAGNOSIS_DESCRIPTION

```

```

FROM FACT, METADATA_D META,
      SUBJECT_B, INTERACTION_B,
      UOM_D, UOM_B,
      CALENDAR_D, CALENDAR_B,
      CLOCK_D, CLOCK_B,
      PROCEDURE_B, PROCEDURE_D,
      MATERIAL_B, MATERIAL_D,
      QUALITY_B, QUALITY_D,
      DIAGNOSIS_B, DIAGNOSIS_D

WHERE FACT.METADATA_ID
      AND FACT.SUBJECT_GROUP_ID
      AND FACT.INTERACTION_GROUP_ID
      AND FACT.CALENDAR_GROUP_ID
      AND CALENDAR_B.MASTER_ID
      AND CALENDAR_B.ROLE
      AND CALENDAR_B.RANK
      AND CALENDAR_D.STATUS_INDICATOR
      AND FACT.CLOCK_GROUP_ID
      AND CLOCK_B.DIMENSION_ID
      AND CLOCK_B.ROLE
      AND CLOCK_B.RANK
      AND CLOCK_D.STATUS_INDICATOR
      AND FACT.UOM_GROUP_ID
      AND UOM_B.MASTER_ID
      AND UOM_B.ROLE
      AND UOM_B.RANK
      AND UOM_D.STATUS_INDICATOR
      AND FACT.PROCEDURE_GROUP_ID
      AND PROCEDURE_B.MASTER_ID
      AND PROCEDURE_D.STATUS_INDICATOR
      AND FACT.MATERIAL_GROUP_ID
      AND MATERIAL_B.MASTER_ID
      AND MATERIAL_D.STATUS_INDICATOR
      AND FACT.QUALITY_GROUP_ID
      AND QUALITY_B.MASTER_ID
      AND QUALITY_D.STATUS_INDICATOR
      AND FACT.DIAGNOSIS_GROUP_ID
      AND DIAGNOSIS_B.MASTER_ID
      AND DIAGNOSIS_D.STATUS_INDICATOR

ORDER BY SUBJECT_B.MASTER_ID, INTERACTION_B.MASTER_ID,
      CALENDAR_D.SORT_DATE, CLOCK_D.SORT_TIME,
      META.SUBJECT, META.EVENT, META.ENTITY, META.ATTRIBUTE;

```

The timeline query, with varying levels of dimension inclusion, is the most popular query style in this star architecture design. It allows a great deal of data to be viewed very quickly, without having to know or anticipate what that data might be or how much of it might be returned. Because each returned row is based on a single fact, it is easy to add filters that vary what is returned. Even for

users who expect to eventually define and build more sophisticated queries of the data involved, the basic timeline is the common starting point. Many users, even those who build more elaborate queries, agree that the timeline query solves many of their problems by answering their questions without having to build those more complicated alternatives. Timelines are the core tool in the star-schema data warehouse.

Business Intelligence

For most warehouse users, particularly non-IT users, nontrivial queries are carried out in some form of business intelligence toolset that provides a semantic layer between user and database (Figure 17.19). These toolsets allow users to access data in the warehouse without having to possess technical knowledge of the database tables in which data are stored, nor knowledge of what every database column might contain.

In essence, the business intelligence layer allows the warehouse support team to add rules to the warehouse that support user requirements without needing to implement all of those rules in the database technology. This creates a decision point in your design as certain rules can be implemented in either the semantic layer or the database. When either is an option, I recommend implementing rules in the database. This choice is particularly important when users might be accessing the database using tools other than the business intelligence environment, or through different business intelligence environments. Rules implemented in the database affect all users of the database. Rules implemented in a business intelligence layer have to be redundantly implemented in all of the available business intelligence interfaces.

Figure 17.19 illustrates the warehouse and any associated data mart at the database level. I recommend that all user access to the warehouse be through a view of the warehouse database that can be used to support your control requirements. Giving a user direct access to the database effectively gives them a fully identified

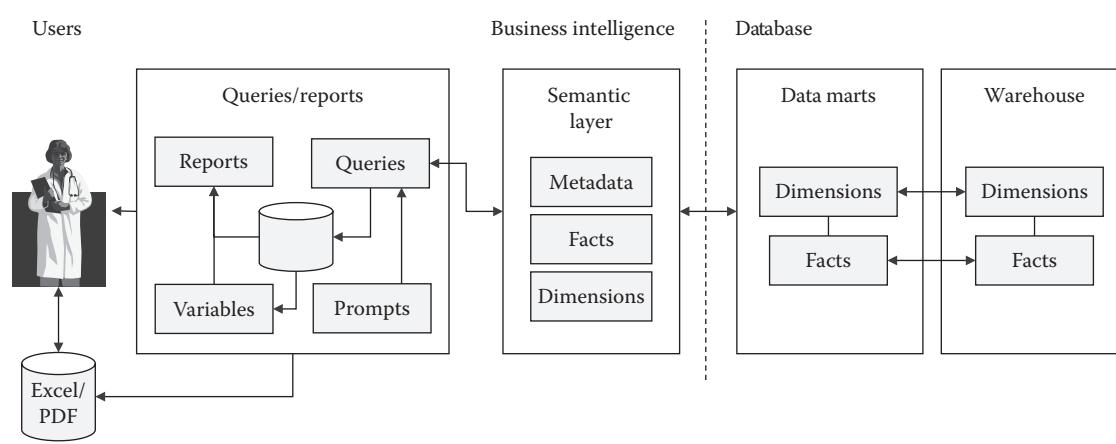


Figure 17.19 Technology pathway between users and warehouse.

and unconstrained view of the data. Almost no users should be given fully unqualified access. Even for users who are perceived to be in positions that would give them full access to the data, there will still be certain controlled conditions where you'll want to avoid providing some data. For example, you'll eventually want to start deleting some information in the warehouse. You'll probably do this logically, by tagging data to be considered deleted rather than physically deleting it from the database. When this occurs, you'll want that data hidden from all warehouse users, even users with full access. If all access is through a data mart view, you'll always have an available layer in which to place logical restrictions.

The first three data mart views I recommend providing are the three views implied by your redaction controls: (1) fully identified, (2) controlled with redaction flags, and (3) redacted. Authorized users can build queries against the needed view, and further logical data mart construction can be done as new views off the database, or as additional views of one of the three baseline data marts.

Characteristics that are desired by users at the query level, but that don't impact the range of tables or columns accessed in the warehouse, are typically implemented in the semantic layer of the business intelligence toolset. These include any alternatively named or constrained columns that users would like to include in their queries. I recommend that you be very careful in setting up these characteristics in your semantic layer because their use can have unexpected consequences. For example, a user might request a semantic object for Drug Name that provides the Description column in the Material dimension where the Subdimension is Drug. Users request these kinds of objects all the time because it's easier to include them in a query than to include the Description column and have to provide a filter on Subdimension. In situations where the Subdimension filter is included in the query, using either the base Description column or the Drug Name object provides the same results. Problems arise when the Drug Name object is built into a query but the Subdimension filter is not. If the facts being queried are expected to be associated with Drug entries in the dimension, users will often omit the Subdimension filter because they believe it is implied by the fact types being queried (e.g., Medication Administrations, or Drug Allergies). In these cases, the results received can be very different depending upon whether the Description column or Drug Name object was used in the query. The Drug Name object includes a constraining filter of which users might be unaware. The effect would be to drop facts from the result set that aren't actually pointing to Material entries in the Drug subdimension. These omissions could include Drug Allergy fact types that were received erroneously against other types of nondrug allergens, and Medication Administrations if some were recorded against entries in the Vaccine subdimension. Facts that had an erroneous key in the Material dimension would also be omitted because they are referencing the Unknown row in the System subdimension. It's possible that users would not want these extra facts within the context of the query being performed, but it's just as likely they would want them and aren't even aware that they have been omitted from the results.

I find semantic objects that provide these hidden constraining rules problematic for these reasons. The definitions of these objects typically presume a high level of data quality that often doesn't exist within the actual data. I know users will request complicated semantic objects in many situations, and you'll provide them as a service to your user base, but be on the lookout for errors and omissions that might result from their use. I would rather have my users get too many results and then filter some out, than to receive too few and not realize it. Views and semantic objects introduce risks, so I recommend seeking approval from the governance body for any new views or objects that are more than trivial in their implementation and impact.

Alternative Data Views

Because the star-schema architecture tends to store data that are deep rather than wide, you will receive many requests to provide views into the data that are flattened (i.e., made wide rather than deep). I agree that flattening can be helpful in certain circumstances; but it opens the door to normalized relational thinking rather than star schema thinking, so there is a risk that users who demand flattened views are actually resisting adapting to a true star architecture. I recommend that you manage against this risk by getting your data governance group involved whenever a user community requests new nontrivial views of the data.

When warranted, flattened views are relatively easy to create because all of the necessary data exist in the warehouse and can be navigated generically through the Metadata dimension. The forms of flattening I see most often involve deep groups and deep fact sets.

Flattened Groups

Some users find it desirable—and helpful—to have the deep bridge entries flattened out so they can be queried as though the multiple bridges are available as a single row. For example, suppose a dataset is being loaded into the warehouse that contains laboratory results that are being qualitatively characterized based on their determined abnormalcy and their result status. The data analyst might have defined metadata that maps each column of source data as reference entries in the Quality dimension:

<i>Reference Column</i>	<i>Value</i>
CONTEXT_KEY_01 (Physical Dataset)	Lab Results
CONTEXT_KEY_02 (Logical Dataset)	~All~
CONTEXT_KEY_03 (Fact Break)	~All~
CONTEXT_KEY_05 (Target Table)	QUALITY_R
CONTEXT_KEY_06 (Target Column)	CONTEXT_KEY_01

Because you're mapping fact data, you need bridge parameters, and because there are two entries for the same dimension, you can't rely on the default one-bridge group. You have to specify bridge parameters for each of the source columns:

<i>CONTEXT_KEY_04 (Source Column)</i>	<i>CONTEXT_KEY_07 (Target Role)</i>	<i>CONTEXT_KEY_08 (Target Rank)</i>	<i>CONTEXT_KEY_09 (Target Weight)</i>	<i>MASTER_ID</i>
ABNORMALCY_INDICATOR	Abnormalcy	1	0.0	634
RESULT_STATUS	Status	1	1.0	635

This situation results in facts that are dimensionalized to two different entries in the Quality dimension ([Figure 17.20](#)).

Users who want to query laboratory results that are abnormal or preliminary would filter their query on the appropriate bridge role. However, users who want to query results that are abnormal *and* preliminary would need a pair of orthogonal subqueries. Orthogonal queries present problems in most business intelligence semantic layers. To solve this common situation, I recommend you implement multiple pathways between the Fact table and each Bridge table in your semantic layer. Two semantic pathways would support the abnormalcy and status query without an orthogonal requirement. The question is: How many pathways should you provide? To completely avoid any orthogonal query requirement, you would have to include a number of pathways equal to the number of bridges in your largest anticipated group in a dimension. You can determine this number by counting the grouped entries in the metadata or by actually counting entries in the Bridge table by Group ID. Counting metadata gives you the largest group currently *anticipated*, and counting the actual data only gives you the largest group currently *stored*. Because new groups need to be anticipated in the metadata before they can be stored in the warehouse, counting metadata provides a leading indicator for any new dataset being sourced that would require even larger bridge entries.

A solution based on raising the group sizes to every increasing numbers is unsustainable, so you'll probably need to implement some flattened views at times. I find supporting two or three pathways against each dimension in the semantic layer very helpful, and it precludes the need to create the vast majority of flattened views that users are likely to request. However, you'll eventually need to create a view, and the algorithm for doing so is very generic: In a flattened

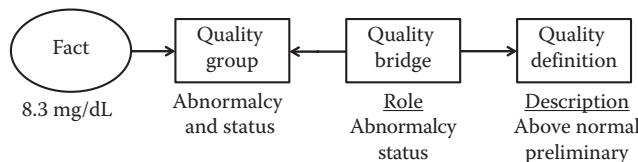


Figure 17.20 Baseline two-quality fact characterization.

group, all or some bridge elements are prepivoted into a view where they appear to be sibling columns in the dimension group table. For the abnormalcy and status example in the Quality dimension, your users might want to see the descriptions of both the abnormalcy and status characteristics in the same row of a table so they can easily filter on both values at the same time. This requirement can be met with a view that pivots the desired data into separate sibling columns of an outrigger table on the Quality group. As an outrigger, the identifier of a row will be the same Group ID that identifies groups in the primary Bridge and Group tables, so any joins that will work for those tables will work against the view. Since the table flattens multiple Definition entries, a query can't pass through the outrigger to get additional data from the Definition, Hierarchy, Reference, or Context tables. Any data needed in the flattened view from those tables must be obtained and included in the outrigger view at the time it is created. This example builds a simple flattened outrigger that includes only the descriptions of the two relevant Definition rows:

```

SELECT GROUP_ID,
       ABNORMALCY_DESCRIPTION,
       STATUS_DESCRIPTION
FROM
  (
    SELECT GROUP_ID,
           DESCRIPTION          AS VALUE,
           ROLE+'_DESCRIPTION' AS TARGET_COLUMN
      FROM QUALITY_D D
     INNER JOIN QUALITY_B B ON B.MASTER_ID = D.MASTER_ID
                           AND B.ROLE IN ('Abnormalcy', 'Status')
                           AND B.RANK = 1
  ) PVT
PIVOT
  (
    MIN(VALUE)
    FOR TARGET_COLUMN IN (
      ABNORMALCY_DESCRIPTION,
      STATUS_DESCRIPTION
    )
  ) P

```

This query implements the orthogonal nature of the flattening through the PIVOT command. For very simple views, this can also be done through a simple two-path orthogonal query, but I recommend staying with the pivot-based option for all your flattening. The single-path pivot design makes it very easy to add more data columns that might be identified for the flattened view at a later time. For example, to add the Vernacular Code to the example view, a new SELECT statement can be added as a UNION to the existing extraction, allowing the same PIVOT to complete the flattening:

```

SELECT GROUP_ID,
       ABNORMALCY_VERNACULAR_CODE,
       ABNORMALCY_DESCRIPTION,
       STATUS_VERNACULAR_CODE,
       STATUS_DESCRIPTION

```

```

FROM
(
    SELECT GROUP_ID,
           DESCRIPTION          AS VALUE,
           ROLE+'_DESCRIPTION' AS TARGET_COLUMN
      FROM QUALITY_D D
     INNER JOIN QUALITY_B B ON B.MASTER_ID = D.MASTER_ID
                           AND B.ROLE IN ('Abnormalcy', 'Status')
                           AND B.RANK = 1
UNION
    SELECT GROUP_ID,
           VERNACULAR_CODE        AS VALUE,
           ROLE+'_VERNACULAR_CODE' AS TARGET_COLUMN
      FROM QUALITY_D D
     INNER JOIN QUALITY_B B ON B.MASTER_ID = D.MASTER_ID
                           AND B.ROLE IN ('Abnormalcy', 'Status')
                           AND B.RANK = 1
)
) PVT
PIVOT
(
    MIN(VALUE)
    FOR TARGET_COLUMN IN (
        ABNORMALCY_VERNACULAR_CODE,
        ABNORMALCY_DESCRIPTION,
        STATUS_VERNACULAR_CODE,
        STATUS_DESCRIPTION
    )
)
) P

```

The outrigger table view created by this pivot architecture makes wider data rows available to users without having to add more pathways to your semantic layer (Figure 17.21).

Note that the query in this example is very precise in its focus. It only creates outrigger rows for facts loaded against groups that include the two roles in the filtering. A few other characteristics are significant. Note that the join to the Quality Bridge table constrains the Rank to 1. There is no indication in the metadata that there might be any more than one entry per Role, but it is always advisable to specify the Rank for any Role to assure uniqueness. Also, note how the Target Column was generated in the query. If you have roles with multiple ranks, you can add Rank to that concatenation, making the view column names a bit clumsier, but guaranteed to be unique.

Let's take a look at another example, this one from Diagnosis. Users are often interested in filtering queries on more than one role in the Diagnosis dimension,

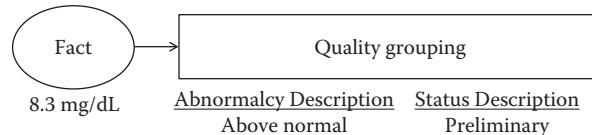


Figure 17.21 Flattened group outrigger pairing Abnormalcy with Status.

and these queries raise the same orthogonal conditions as any other multibridge qualification. The metadata might include the following:

Reference Column	Value
CONTEXT_KEY_01 (Physical Dataset)	ADT
CONTEXT_KEY_02 (Logical Dataset)	Discharge
CONTEXT_KEY_03 (Fact Break)	~All~
CONTEXT_KEY_05 (Target Table)	DIAGNOSIS_R
CONTEXT_KEY_06 (Target Column)	CONTEXT_KEY_01

Because you're still mapping fact data, you need bridge parameters, and because there are two entries for the same dimension, you can't rely on the default one-bridge group. You have to specify the bridge parameters for each of the source columns as you did in the Quality dimension:

CONTEXT_KEY_04 (Source Column)	CONTEXT_KEY_07 (Target Role)	CONTEXT_KEY_08 (Target Rank)	CONTEXT_KEY_09 (Target Weight)	MASTER_ID
PRIMARY_DIAG	Primary	1	1.0	721
DISCHARGE_DRG	DRG	1	0.0	722

This situation results in facts that are dimensionalized to two different entries in the Diagnosis dimension (Figure 17.22).

The query to implement the flattened outrigger view that provides for both DRG and Primary Diagnosis in the same row is

```
SELECT GROUP_ID,
       PRIMARY_VERNACULAR_CODE,
       DRG_VERNACULAR_CODE,
FROM
(
  SELECT GROUP_ID,
         VERNACULAR_CODE AS VALUE,
         ROLE+_VERNACULAR_CODE' AS TARGET_COLUMN
  FROM DIAGNOSIS D D
```

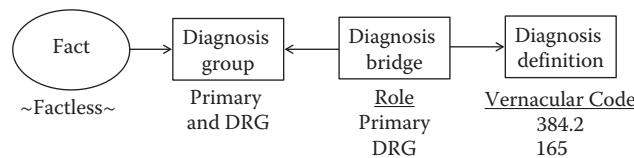


Figure 17.22 Two-bridge grouping for Primary and DRG diagnoses.

```

        INNER JOIN DIAGNOSIS_B B ON B.GROUP_ID = D.GROUP_ID
                                AND B.ROLE IN ('Primary', 'DRG')
                                AND B.RANK = 1
    ) PVT
PIVOT
(
    MIN(VALUE)
    FOR TARGET_COLUMN IN (
        PRIMARY_VERNACULAR_CODE,
        DRG_VERNACULAR_CODE
    )
) P

```

The generic nature of the flattening problem, along with the generic architecture for the warehouse dimensions, makes this query almost exactly like all other group-flattening queries. The resulting view provides users with a pair of columns that can be filtered or displayed as needed (Figure 17.23).

If the requirement were for the flattened view to include both the codes and descriptions, the query would be a little bigger, but not much more complicated:

```

SELECT GROUP_ID,
       PRIMARY_VERNACULAR_CODE,
       PRIMARY_DESCRIPTION,
       DRG_VERNACULAR_CODE,
       DRG_DESCRIPTION,
FROM
(
    SELECT GROUP_ID,
           VERNACULAR_CODE                      AS VALUE,
           ROLE+'_VERNACULAR_CODE'              AS TARGET_COLUMN
      FROM DIAGNOSIS_D D
     INNER JOIN DIAGNOSIS_B B ON B.GROUP_ID = D.GROUP_ID
                                AND B.ROLE IN ('Primary', 'DRG') AND B.RANK = 1
UNION
    SELECT GROUP_ID,
           DESCRIPTION                         AS VALUE,
           ROLE+'_DESCRIPTION'                 AS TARGET_COLUMN
      FROM DIAGNOSIS_D D
     INNER JOIN DIAGNOSIS_B B ON B.GROUP_ID = D.GROUP_ID
                                AND B.ROLE IN ('Primary', 'DRG') AND B.RANK = 1
) PVT

```

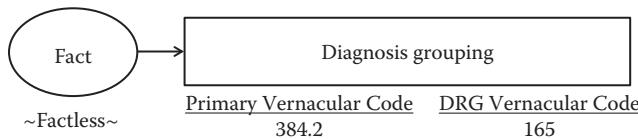


Figure 17.23 Paired Primary ICD-9 and DRG diagnosis.

```

PIVOT
(
    MIN (VALUE)
    FOR TARGET_COLUMN IN (
        PRIMARY_VERNACULAR_CODE,
        PRIMARY_DESCRIPTION,
        DRG_VERNACULAR_CODE,
        DRG_DESCRIPTION
    )
) P

```

Flattened group views provide wider data access to users who might be struggling with deep data, and they also prevent you from adding more semantic pathways to the dimensions in your semantic layer. I strongly recommend that they be used carefully and be kept as generic as possible.

Flattened Facts

The other table that you'll find needs to be flattened at times is the Fact table. Conceptually, the issues involved in providing flattened facts are the same as for dimension groups. In practice, however, the range of combinations of possible widened facts is enormous compared to the range of bridges that might be encountered in a dimension group. If you can identify a cluster of facts that is useful to query as a wider pivoted view, the logic of the view is the same as for flattened dimension groups. For example, this flattened view provides time-stamped vital signs data for patients that were loaded into the warehouse through six source columns in the Vitals logical dataset of the Nursing physical dataset ETL load:

```

SELECT MAX(FACT_ID) AS WIDE_FACT_ID
SUBJECT_GROUP_ID,
CALENDAR_GROUP_ID,
CLOCK_GROUP_ID,
EHR_VITALS_PATIENT_SYSTOLIC,
EHR_VITALS_PATIENT_DIASTOLIC,
EHR_VITALS_PATIENT_TYMPANIC,
EHR_VITALS_PATIENT_ORALTEMP,
EHR_VITALS_PATIENT_RESPIRATION,
EHR_VITALS_PATIENT_PAIN
FROM
(
    SELECT FACT_ID,
    SUBJECT_GROUP_ID,
    CALENDAR_GROUP_ID,
    VALUE,
    CONCAT(SOURCE, '_', EVENT, '_', ENTITY, '_', ATTRIBUTE) AS TARGET_COLUMN
    FROM FACT
    INNER JOIN METADATA_R META ON FACT.METADATA.MASTER_ID = META.MASTER_ID
    AND CONTEXT_KEY_01 = 'Nursing'
    AND CONTEXT_KEY_02 = 'Vitals'
    AND SOURCE = 'EHR'
    AND EVENT = 'Vitals'
)

```

```

        AND ENTITY          = 'Patient'
        AND ATTRIBUTE
            IN ('Systolic', 'Diastolic', 'Tympanic', OralTemp', 'Respiration', 'Pain')
        ) PVT
PIVOT
(
    MIN(VALUE)
    FOR TARGET_COLUMN IN (
        EHR_VITALS_PATIENT_SYSTOLIC,
        EHR_VITALS_PATIENT_DIASTOLIC,
        EHR_VITALS_PATIENT_TYMPANIC,
        EHR_VITALS_PATIENT_ORALTEMP,
        EHR_VITALS_PATIENT_RESPIRATION,
        EHR_VITALS_PATIENT_PAIN
    )
) P

```

Taken to an extreme, these flattened views can ultimately present your warehouse to users or tools that need to see your star schema architecture presented as a relational warehouse model. The Source, Event, Entity, and Attribute columns in the metadata, when combined, provide a consistent way to name the resulting relational columns in the various created views. Providing maximum flexibility in how data are seen in the warehouse allows you to minimize how often you need to export data to an external data mart, but that need arises eventually.

External Data Marts

No matter how many views of the data you build in your warehouse, at either the database or semantic layer levels, you'll eventually need to externalize some of your warehouse data. These *external* data marts share many characteristics with the internal data mart views you built into the warehouse, the primary difference is you give up control over the data and their use when you externalize them into nonwarehouse datasets. Your exports to the external data marts can include much of the warehouse control data themselves, but your target environment probably won't know how to use that data to provide the same controls available in your data warehouse. This means you will lose orphan, unexpected, and undesired row-level controls and, most importantly, automatic redaction of privacy-protected data. Depending upon the design of the external data mart, you are likely to also lose much of your UOM and Data State controls. If those risks are acceptable, the remaining larger problem you have to deal with is in negotiating the amount and complexity of data to be exported.

i2b2

In Chapter 2, the i2b2 data warehouse structure was used to illustrate an intermediate design point between a completely generic one dimensional star schema and the functionally-specific 26 dimensional biomedical warehouse you

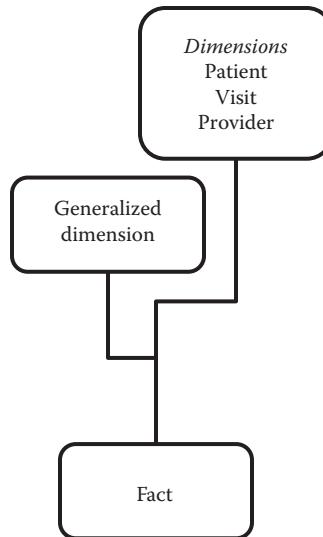


Figure 17.24 i2b2 structure as external data mart candidate.

are implementing (Figure 17.24). The i2b2 warehouse model is widely used across biomedical communities and interfaces with many commonly used and powerful research tools. If your governance group decides to authorize an export of a subset of your data warehouse into an i2b2 warehouse for use by members of your user community, the data export you design can be very specific or generic depending on how often you expect to implement exports. One-time or throw-away exports can be done very specifically. The more often you plan to export data, the more generic you should make your export tools to support that activity. The key design parameters tend to be (1) the number of explicit dimensions available, (2) the availability of at least one generic dimension to handle all the dimensional data beyond the list of explicit dimensions, (3) the complexity of the fact tables, and (4) whether there are any hidden dimensions in the fact tables.

The i2b2 star schema is a nominal four dimensional warehouse that includes three explicit dimensions (e.g., Patient, Visit, and Provider) and one generic dimension (e.g., Concept). The explicit i2b2 dimensions should be able to handle your Subject, Interaction, and Caregiver data without much modification. The i2b2 Patient dimension includes properties of your Geopolitics dimension, so it can be used to dimensionalize your address-related patient facts. Most of your dimensional data, that aren't handled by the three explicit dimensions or the Fact table, will go in the Concept dimension. The i2b2 Fact table includes several implicit dimensions. The Start Time can serve as your Calendar and Clock dimensions, and the End Time that should be able to simulate some aspects of your Data State dimensions, if you export Superseded facts. A Units Code that is available for the Fact table will serve your UOM dimension needs, if you don't export the "Expected" UOM error and warning conditions from your warehouse. The Modifier Code in the key of the Fact table can be used as a proxy for your Metadata dimension.

The i2b2 structure has only a limited set of columns, so each time you load data you have to decide how to use each column to meet your needs. Each time you design an export to an i2b2-based data mart you might end up with a different set of data. You can modify or extend the i2b2 table design, but that will usually defeat your purpose in choosing the i2b2 data mart to get your warehouse out to a place where it can be accessed by the wide range of applications and tools that are able to use the i2b2 data structures. Instead, you will need to develop appropriate algorithms for mapping the data you want to extract from your warehouse into the external data mart of your choice. I have found that mapping data into i2b2 is good practice for data mart extraction generally. Other external data marts are typically much simpler than i2b2, so can usually reuse techniques I've adapted from my i2b2 scenarios.

Provider Dimension

A good place to start i2b2 mapping is the Provider dimension because it is a very simple dimension conceptually. The dimension supports a Provider ID and a character Name column, neither of which serves as the key to the dimension. The key is the Provider Path, and the configuration of a path is central to using the i2b2 database. If you conceptualize each Provider ID and Name as a different provider in the dimension, the Path key represents the different pathways to that provider in your data. It will contain any of the aggregating characteristics from your Caregiver dimension that you will want to use to access the provider, typically drawing on columns in your Reference, Definition, Hierarchy, and Bridge tables. For example, you might decide you want your data mart Provider dimension to be very detailed; so you not only know the provider for each fact, but also use the warehouse Bridge table to establish a deep context for each fact-to-provider connection in the data mart. The deepest possible connection would include enough Bridge information to be able to isolate the combinations of providers who were involved in the same original fact in your warehouse.

To obtain that level of detail, your i2b2 Provider dimension needs to include the Role, Rank, Weight, and Group ID from your Caregiver Group table. In my experience, the Role property alone is sufficient for most data mart purposes. For example, each Provider in your Caregiver dimension could be exported to i2b2 in each Role in which they appear in the facts that will be exported. Alternatively, the list of roles exported might be limited to a subset that you decide is more relevant to the purpose of the exported data mart. For example, many data mart extracts limit provider connections to only those providers in the Attending role. You won't be able to duplicate all of the data capabilities of your warehouse in i2b2, so you'll have to develop an intuition for which features are necessary and which can be omitted. I've never encountered a problem remodeling the data without the Rank, Weight, or Group ID. I have

found that if I omit Role from Provider, the queries that can be written become limiting during data analysis. With Role as the deeper grain for Provider, you next have to choose your aggregation levels. Initial aggregations are always chosen from your Definition table. I typically choose the Subdimension and Category columns for aggregation. These two columns provide for groupings of definitions in the warehouse Definition tables, so they are a consistent way to carry the data over into an i2b2 pathway. If your implementation isn't using the Category column or if you only have one subdimension scoped for extraction, either can be omitted.

You might also have some user-defined columns in your Definition table that are being used in query filtering that can be used as pathway aggregations. Some of these might also be defined in your Hierarchy table, so select whichever ones best fit your extract design approach. For example, you might have a Clinical Specialty column in the Definition table that is also accessible through your Hierarchy table. If the only thing you need for your pathway is the value of the Clinical Specialty column, extracting it from the same row from which you obtain the rest of your data is fine. However, if you extract the Clinical Specialty by navigating through the dimension hierarchy, you gain the ability to further aggregate the data by the Subdimension and Category of the clinical specialty definition row. I recommend this latter approach if also higher-order entries will be placed into the dimension in such a way that certain facts will be dimensionalized to those higher-order entries. If both levels of entries in the i2b2 dimension are built using the same warehouse data, you'll find it easier to integrate the various data after they are placed into the i2b2 data mart.

Your algorithm for extracting data to be loaded into the i2b2 Provider pathway will result in a set of aggregating concepts, such as

```
Clinician
Clinician\Cardiologist
Clinician\Cardiologist\Provider
Clinician\Cardiologist\Provider\Hospitalist
Clinician\Cardiologist\Provider\Hospitalist\NID:0123456
Clinician\Cardiologist\Provider\Hospitalist\NID:0123456\Admitting
Clinician\Cardiologist\Provider\Hospitalist\NID:0123456\Attending
Clinician\Cardiologist\Provider\Hospitalist\NID:0123456\Consulting
```

The last four rows of conceptual pathways represent the detailed definitions required to load data from your warehouse into i2b2 based on extracting facts using the Admitting, Attending, Consulting, or Caregiver roles in your Caregiver Bridge table. The Caregiver role is stripped away into an aggregate concept without any role because it served as a default single-bridge Role in the warehouse and didn't actually indicate any functional role.

Beyond mapping entries directly from your Caregiver dimension into the i2b2 Provider dimension, you should consider other information in your warehouse that might be needed in the i2b2 Provider dimension to meet the expectations of your anticipated users. It's not uncommon for queries to aggregate provider data by the organization in which the provider is practicing. Some of the facts in your warehouse might completely lack detailed provider data and only dimensionalize to an organization. Your warehouse facts might treat Caregiver as Not Applicable entirely or as temporarily Undetermined. Using these facts in i2b2, particularly aggregating these facts, requires higher-level concepts to be added to your provider pathways.

If you have facts in your warehouse that assign caregivers to organizational departments, they can be used to extend the pathways for the caregivers being extracted. Through these facts, you can extract the department the caregiver is assigned to, and any aggregating organization in which that department is defined in the organization Hierarchy table. The resulting set of concepts is then enriched to include that data in each pathway:

```
Hospital
Hospital\St. Joseph's Hospital
Hospital\St. Joseph's Hospital\Department
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist\Provider
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist\Provider\Hospitalist
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist\Provider\Hospitalist\NID:0123456
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist\Provider\Hospitalist\NID:0123456\Admitting
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist\Provider\Hospitalist\NID:0123456\Attending
Hospital\St. Joseph's Hospital\Department\Emergency Department\
    Clinician\Cardiologist\Provider\Hospitalist\NID:0123456\Consulting
```

The conceptual pathways in a dimension can get very complicated if you have a lot of rich data in your warehouse dimensions. If your warehouse data are defined in a manner where everything rolls up into clean, nonorthogonal hierarchies, at least the complexity remains algorithmically linear. If your warehouse hierarchies are based on more complex direct-acyclic-graph perspectives, or multiple orthogonal hierachic perspectives, the possible permutations of aggregating concepts grow geometrically. In theory, it can all be exported and loaded into i2b2, but in practice, you'll have to reduce and disaggregate your warehouse data to simplify the i2b2 loads.

Patient Dimension

The i2b2 Patient and Visit dimensions present a different set of challenges. Entries in both dimensions are identified using an integer surrogate, so you can use your warehouse identifiers in i2b2. I recommend using the Master ID from your Subject and Interaction dimensions since i2b2 doesn't directly support slowly-changing dimensions. If instead, you choose to use the warehouse Definition ID as an identifier, you'll end up with a different set of i2b2 dimension rows for each slowly-changing variant of the Master ID entry in your warehouse, and those entries will be difficult to connect within i2b2. I would only suggest that path if a cross analysis of dimensional characteristics is the main point of your i2b2 data mart application. Generally, your data mart will be built for the biomedical facts it will contain, and losing some of your slowly-changing variations in the dimensions won't be considered problematic.

You will load the i2b2 Patient dimension from your Subject dimension, using the Master ID as your i2b2 identifier. If you are using your Subject Hierarchy to establish a relationship between your patients and their specimens, you'll probably just extract the Person rows in the dimension for loading into i2b2. If so, remember to navigate the hierarchy to get the Master ID of the *ancestor* Person when extracting facts that have been dimensionalized to Specimen rows so they'll load correctly into i2b2. The i2b2 Patient dimension includes columns from your Geopolitics dimension, so you'll need to extract that data through one of your patient address facts as you extract your patient data. If you join the dimension through the address fact to the Geopolitics dimension, your extract columns will look very similar to the target i2b2 Patient dimension columns.

Visit Dimension

Your Interaction dimension will serve as your source for the i2b2 Visit dimension. The exact grain you extract for visits will depend on your data mart requirements, but typically you can pull all Interaction dimension entries and store them in i2b2 by Master ID. The Visit dimension does support a pathway hierachic capability, so it is possible to store slowly-changing variations of Interaction within the Visit dimension. If you choose to do so, you should use the warehouse Definition ID to identify the Visit in i2b2 and use the detailed end of the visit pathway to group each Definition ID into the appropriate Master ID. Note that disaggregating slowly-changing data through an i2b2 pathway gets complicated. In particular, the Visit dimension includes a Start Time and an End Time. If you populate those two columns with the Admission and Discharge timestamps from the warehouse, they will carry NULL values in some rows but not others, based on the temporality of updates in the warehouse. This choice will impact the availability of any associated facts whenever the Start and End times are used in an i2b2 query as selection or filtering criteria. You might want

to consider not loading those columns into i2b2 as slowly-changing data in order to prevent this problem.

Regardless of your decision to include slowly-changing data in your Interaction dimension, you'll use the pathway data in the i2b2 Visit dimension to provide for the grouping of data into aggregating hierarchies. As with the Provider dimension, I recommend that you always consider the Subdimension and Category columns as pathway elements. These two columns provide some minimum context for visit data that have been pulled from your Interaction dimension:

```
Encounter
Encounter\Emergency
Encounter\Emergency\01234
Encounter\In-Patient
Encounter\In-Patient\12345
Visit
Visit\Homecare
Visit\Homecare\23456
Visit\Homecare\34567
Visit\Office
Visit\Office\45678
Visit\Office\56789
Visit\Clinic
Visit\Clinic\67890
```

Extracting additional columns from the Interaction dimension can add richness to the Visit dimension. These columns can be specific to the Subdimension or Category entries since those two columns already exist as higher elements of the targeted pathways:

```
Encounter
Encounter\Emergency
Encounter\Emergency\Ambulance
Encounter\Emergency\Ambulance\01234
Encounter\In-Patient
Encounter\In-Patient\12345
Research
Research\Participation
Research\Participation\ABC1242
Visit
Visit\Homecare
Visit\Homecare\Emergency
Visit\Homecare\Emergency\23456
Visit\Homecare\Scheduled
Visit\Homecare\Scheduled\34567
Visit\Office
Visit\Office\Appointment
Visit\Office\Appointment\45678
```

```
Visit\Office\Walk-In  
Visit\Office\Walk-In\56789  
Visit\Clinic  
Visit\Clinic\67890
```

As with the Provider dimension, you might want to extend your aggregation of Visit pathways to include institutional data. You can extract data for each Interaction through certain facts that also dimensionalize to the Organization dimension, so you can group Visits into continually higher-level aggregates:

```
Wilson Hospital  
Wilson Hospital\Emergency  
Wilson Hospital\Emergency\Encounter  
Wilson Hospital\Emergency\Encounter\Emergency  
Wilson Hospital\Emergency\Encounter\Emergency\Ambulance  
Wilson Hospital\Emergency\Encounter\Emergency\Ambulance\01234  
Wilson Hospital\Cardiology  
Wilson Hospital\Cardiology\Encounter\In-Patient  
Wilson Hospital\Cardiology\Encounter\In-Patient\12345  
Wilson Hospital\Medical School  
Wilson Hospital\Medical School\Research  
Wilson Hospital\Medical School\Research\Participation  
Wilson Hospital\Medical School\Research\Participation\ABC1242  
Care Management Practice  
Care Management Practice\Visit  
Care Management Practice\Visit\Homecare  
Care Management Practice\Visit\Homecare\Emergency  
Care Management Practice\Visit\Homecare\Emergency\23456  
Care Management Practice\Visit\Homecare\Scheduled  
Care Management Practice\Visit\Homecare\Scheduled\34567  
Cardiology Associates  
Cardiology Associates\Visit\Office  
Cardiology Associates\Visit\Office\Appointment  
Cardiology Associates\Visit\Office\Appointment\45678  
Cardiology Associates\Visit\Office\Walk-In  
Cardiology Associates\Visit\Office\Walk-In\56789  
Southside Clinic  
Southside Clinic\Visit  
Southside Clinic\Visit\Clinic  
Southside Clinic\Visit\Clinic\67890
```

If the facts between the Interaction and Organization dimensions in your warehouse offer many permutations or the hierarchies in either dimension contain orthogonal perspectives, you'll need to select the most useful pathways from among the available alternatives. There will be times when the explosion of permutations implied by some of your aggregations becomes too extensive, either because the data are too numerous and cumbersome to manage, or the complexity of the pathways themselves becomes too complex or confusing for

the users of the data mart to query effectively. Your choices in those situations will be to scale back the data you want to load into the data mart or split the entries in the dimension so each smaller component has a simpler pathway. If the Interaction and Organizational components of your extracts were not combined into a single pathway, the Visit data might look like this

```
Interaction
Interaction\Encounter
Interaction\Encounter\Emergency
Interaction\Encounter\Emergency\Ambulance
Interaction\Encounter\Emergency\Ambulance\01234
Interaction\Encounter\In-Patient
Interaction\Encounter\In-Patient\12345
Interaction\Research
Interaction\Research\Participation
Interaction\Research\Participation\ABC1242
Interaction\Visit
Interaction\Visit\Homecare
Interaction\Visit\Homecare\Emergency
Interaction\Visit\Homecare\Emergency\23456
Interaction\Visit\Homecare\Scheduled
Interaction\Visit\Homecare\Scheduled\34567
Interaction\Visit\Office
Interaction\Visit\Office\Appointment
Interaction\Visit\Office\Appointment\45678
Interaction\Visit\Office\Walk-In
Interaction\Visit\Office\Walk-In\56789
Interaction\Visit\Clinic
Interaction\Visit\Clinic\67890
Organization
Organization\Wilson Hospital
Organization\Wilson Hospital\Emergency
Organization\Wilson Hospital\Cardiology
Organization\Wilson Hospital\Medical School
Organization\Care Management Practice
Organization\Cardiology Associates
Organization\Southside Clinic
```

I include the name of the warehouse dimension as an additional highest order pathway entry whenever I combine multiple warehouse dimensions into a single pathway construction so data mart users know exactly what they are seeing in their results. The impact of splitting the pathways in the dimension is that I need to potentially double the facts being stored since each fact I would have stored against the combined pathway now needs to be stored against each of the split pathways. There are exceptions that prevent a complete doubling of facts, however. If, in the warehouse, either dimension is marked as Not Applicable, that version of the fact can be omitted in the data mart. Also, the

range of facts required for the secondary pathway dimension (in this case, Organization) is often much smaller than the full range of facts being loaded into the data mart. For example, millions of clinical facts might be brought into the warehouse; but only several thousand admission–discharge–transfer (ADT) facts might need the Organizational pathway, so data mart users could easily limit queries to only clinical facts for visits with ADT facts against a particular organization of interest.

With the dimension pathways simplified by the splitting of different perspectives, other details might be added that would be overwhelming in the single merged set of pathways. For example, here is the subset of Visits that represent participation in research:

```
Interaction\Research  
Interaction\Research\Participation  
Interaction\Research\Participation\ABC1242
```

For research visits, I often recommend extending the pathway to include the Protocol from the Study dimension, making it much easier for users to query visits related to particular studies of interest:

```
Interaction\Protocol  
Interaction\Protocol\PR12345  
Interaction\Protocol\PR12345\Research  
Interaction\Protocol\PR12345\Research\Participation  
Interaction\Protocol\PR12345\Research\Participation\ABC1242
```

The combinations and permutations of warehouse dimension data that can be exported to construct dimensions and pathways in i2b2 is enormous, and care must be exercised to prevent an unnecessary and counterproductive explosion of data in i2b2 that just gets in the way of meeting the requirements for which the i2b2 data mart is being provided. To avoid that pitfall, model the extraction of facts first and only include dimension data from the warehouse that are actually referenced by the facts being extracted.

Concept Dimension

Every fact extracted from the warehouse will be unique to the requirements of the data mart being provided. The breadth and depth of required data can be any combination of data across any of the dimensions and metadata fact types in your warehouse. By whatever criteria are relevant, you eventually identify the set of facts that you want to extract. Before you treat them as a fact extract, however, you first treat the list of facts as the criterion for extracting your dimensional data. Based on the choices you made in planning your Patient, Visit, and Provider dimensions and pathways, you can then extract the fact dimensionalization to

Subject, Interaction, and Caregiver (along with some Geopolitics and Study, as well) to produce the distinct combinations of entries for those i2b2 dimension loads. This dimensionalized fact extract still has a lot of dimensional data associated with it that needs to be extracted and sent to i2b2. Since the only i2b2 dimension that hasn't been designed yet is the Concept dimension, all of your remaining dimension data will end up in some form in the i2b2 Concept dimension.

The analysis you conduct in order to define Concept dimension entries is analogous to the analysis you performed against the other i2b2 dimensions. The Concept dimension is identified by a pathway in the same manner as the Provider dimension, so the analytical techniques you developed with that simpler dimension will apply to the Concept dimension. The difficulty in determining Concepts is in handling of combinations and permutations of dimension data that might be relevant to the facts you'll extract. I recommend starting with each dimension as a standalone set of concepts so you can determine the entries and pathways needed just at that level. Subsequently combining dimensions makes the concept data more complicated and can't really be explored or defined until the data of each single dimension are modeled.

As with Provider, the Concept pathways that might be needed are drawn from the Reference, Definition, Hierarchy, and Bridge tables in the warehouse dimensions. An example set of concept pathways that could be modeled in Diagnosis based on ICD-9 coding includes

```
Diagnosis
Diagnosis\390-459 Circulatory
Diagnosis\390-459 Circulatory\410-414 Ischemic heart disease\
    410 Acute myocardial infarction
Diagnosis\390-459 Circulatory\410-414 Ischemic heart disease\
    410 Acute myocardial infarction\
        410.0 Acute myocardial infarction of anterolateral wall
Diagnosis\390-459 Circulatory\410-414 Ischemic heart disease\
    410 Acute myocardial infarction\
        410.0 Acute myocardial infarction of anterolateral wall\
            410.01 Acute myocardial infarction, of anterolateral wall,
                initial episode of care
Diagnosis\390-459 Circulatory\410-414 Ischemic heart disease\
    410 Acute myocardial infarction\
        410.0 Acute myocardial infarction of anterolateral wall\
            410.01 Acute myocardial infarction, of anterolateral wall,
                initial episode of care\Primary
Diagnosis\390-459 Circulatory\410-414 Ischemic heart disease\
    410 Acute myocardial infarction\
        410.0 Acute myocardial infarction of anterolateral wall\
            410.01 Acute myocardial infarction, of anterolateral wall,
                initial episode of care\Working
```

Similar analysis could generate this example of pathways for an NDC medication identifier in your warehouse Material dimension:

```
Material
Material\Medication
Material\Medication\Anti-infectives
Material\Medication\Anti-infectives\Antifungals
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\Nystatin
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\
    Nystatin\00005542918
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\
    Nystatin\00005542918\Administered
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\
    Nystatin\00005542918\Dispensed
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\
    Nystatin\00005542918\Orderred
```

In each of these analytical examples, you are generating permutations of concepts as they exist in hierarchic stacks in your dimensions, drilling down to the Roles the entries play against facts. The number of combinations algorithmically generated can be staggering. Even these examples hide some of the detailed complexity: They are sufficient if your facts tend only to be recorded against the leaves of your dimensional hierarchies. Facts are often recorded against intermediate levels in dimensional hierarchies, so the pathway concepts created by adding Role data from our Bridges usually has to be generated at intermediate points in the pathways as well. For example, if your extract includes data against the Nystatin example, but your facts don't always include a specific NDC code, some additional pathways are needed to provide for role-based involvement of the hierarchic intermediate:

```
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\Nystatin\Administered
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\Nystatin\Dispensed
Material\Medication\Anti-infectives\Antifungals\Misc Antifungals\Nystatin\Orderred
```

The number of entries and pathways you'll require in the i2b2 Concept dimension can be extremely high for most of the data marts you might be asked to build. To keep the volume more manageable, you need to ensure that you completely understand the requirements that the new data mart is intended to satisfy. If you omit data from your extracts that isn't actually needed in the data marts, the Concept dimension gets easier to define and load.

There are still permutations of data in your warehouse that aren't represented by the Concept pathways illustrated. Suppose the requirements for the data mart include cross-dimensional issues, such as knowing which diagnosis a medication was ordered against, or knowing what facility a procedure was performed in. Even if you loaded all of your Diagnosis, Material, Facility, and Procedure data into the Concept dimension, you still have to address how the cross-dimensional requirements will be met. Depending on how you load your facts into i2b2, some of the requirements might be met by designing sets of facts you load. A warehouse fact dimensionalized against a procedure and a facility can load

into i2b2 as two facts: one against the facility concept, and the other against the procedure concept. The medication order fact will load into i2b2 as a fact against the material concept for the medication, and again as another fact for the diagnosis concept. Whether these pairs of facts can be aligned in a query depends on how they can be recognized as a likely pair, and on the density of other related facts in the fact table. It's easier to match pairs of facts in sparse fact tables than in dense ones. Either way, relying on matching can introduce risk into the meeting of cross-dimensional requirements.

An alternative to the paired-fact design is to create combined Concept entries for each of the required combinations of Concepts. An artificial Concept is loaded into i2b2 for each allowed combination of Diagnosis–Material and Facility–Procedure, and individual facts are dimensionalized to the new combo-Concepts rather than creating two paired facts. This alternative enables a great deal of flexibility in supporting requirements; but at a cost of a major expansion in Concepts, as well as increased difficulty in querying and understanding results in the large or more complex i2b2 queries. Most i2b2 data marts reach a maximum size and complexity, not because of the number of facts included, but because of the increasing complexity of the Concept dimension. Anything you can do to keep the Concept complexity under control will ultimately allow you to bring over more facts, and satisfy more requirements.

Fact Table

The extraction of facts from the warehouse drives the load of both facts and dimensions in the i2b2 data mart. The facts are extracted and loaded last, but they are analyzed first so the dimensional data loaded into i2b2 can be constrained to the data needed to support the actual facts being brought into the data mart. While you have the option of doing full dimension extracts from your warehouse into the i2b2 dimensions, the reality is that you only need to transfer dimensional data that are needed to dimensionalize the facts you'll be bringing over. Facts determine your scope in the data mart.

Any data mart in which you transfer data will typically be a subset of the data in your warehouse, usually a significantly small subset. When asked to extract data for a new data mart, the analytical challenge you face can be termed *data reduction*. You want to reduce the scope and scale of data you extract in order to expedite the loading of the new data mart and reduce the exposure involved in allowing the data to leave the controlled environment of the warehouse. Your biggest reduction will occur when you identify the cohort of patients to be included in your extract. Keep in mind that one of the more powerful tools in i2b2 is the CQT. Researchers use i2b2 to explore and identify cohorts for their studies. In that sense, they'll be identifying cohorts of patients within the cohort that you extract. As a result, the cohort you extract will tend to be very large, encompassing all of the types of cohorts that the users of the data mart might want to explore. Your cohort could be all patients admitted in the past

2 years for any reason, or all patients in a given geographic area. These are large cohorts, but are still very small compared to the range of data you have in your warehouse. The more specific you can be about the cohort of interest for extract, the smaller your resulting extract and i2b2 load will be.

Within the anticipated cohort, you'll continue reductions by looking at the range of facts available for the patients in your cohort and potentially eliminating many of them from your extract scope. If the i2b2 focus is clinical, you should omit many administrative, operational, or financial details. Some fact types could need more analysis before being excluded. For example, your users might want laboratory results and medication administrations but not the corresponding orders. Among those clinical results and administrations, you might omit lab results that were errored-out due to contamination or medications not given. Allergy assessments with positive findings are probably needed but not routine negative findings (e.g., NKAs). For patients with extensive vital signs data, an extract could be limited to only one set of results per hour. Eliminating facts that aren't needed reduces the scale of the extract and makes the data mart more usable. Unneeded facts in the data mart can clutter the queries and cause the data mart to be less useful than you intended.

Once fact types have been selected for extract, the next step in data reduction is to evaluate the usefulness of the different dimensions that the facts reference. A clinical extract often omits the Facility dimension. Unless financial facts are planned for extraction, the Accounting dimension is typically not extracted for data marts. Each dimension is evaluated separately on a case-by-case basis for the different facts that are anticipated. Including a dimension in the extract for one set of fact types doesn't require that all fact types dimensionalize to that dimension data in the data mart, even if they do in the warehouse. For dimensions that are selected for extraction, further data reductions can be made by reducing the group cardinality of the extracted data. For example, some of the bridge entries in some of the dimensions could be omitted from the extract. This could be accomplished explicitly (e.g., only extract Attending caregivers) or more implicitly (e.g., only extract roles with Weight greater than zero).

When analysis for your data mart extract is complete, you can implement the extraction queries that fulfill your design. Most data mart extracts follow a fairly standard pattern, but each is distinct because of the choices you made in your analysis. The first stage is to establish the facts in the warehouse that you want in your data mart, by Fact ID. For very simple data marts, this can be accomplished in one query:

```
SELECT FACT_ID
  FROM FACT F
INNER JOIN CALENDAR_B B ON (F.CALENDAR_GROUP_ID = B.GROUP_ID AND B.ROLE = 'Event')
INNER JOIN CALENDAR_D D ON (B.MASTER_ID = D.MASTER_ID
                           AND D.CALENDAR_DATE > '2012-01-01'
                           AND < metadata entries chosen for extract> ;
```

This query pulls the identifier for all facts that actually occurred from January 1, 2012 onward. Note that the filter on the Event role prevents older facts loaded more recently from being extracted. For simple data marts, this query might be sufficient to define the entire extract strategy. For more complicated extract designs, you might need to create multiple orthogonal queries of this type, each with different selection criteria. As long as you can union all of those queries into a single distinct list of Fact IDs, you will have correctly defined your extract scope. The number of Fact IDs in the list is an indicator of what will be loaded into the data mart, but it is not the actual fact count that will be loaded. The final fact count in the data mart will be significantly larger than the number of facts being extracted from your warehouse. Even with data reduction during your analysis, the facts you are extracting are still likely to be connected to many dimension entries. Each of those entries will become a separate Concept in i2b2 that will trigger a separate fact in i2b2. A single warehouse fact will often load into i2b2 as 5–20 separate conceptualized facts. Keeping dimensions and roles within your extract scope increases the resulting fact count as each warehouse fact is joined in a Cartesian product to each of those extracted Concepts in i2b2.

Once your Fact ID list is identified for the extract, you can begin extracting the dimension references for those facts. You should filter each dimension extract according to your chosen strategy for using that dimension in i2b2. The basic query always involves a filter to ensure only the desired Fact IDs are included in the extract, and each dimension entry is only extracted once. This is accomplished by choosing in advance which Context entry will be used for each dimension row and including a filter to that context. Many dimension entries in the warehouse have multiple identifiers across many contexts, but since the data mart only supports one, it creates unnecessary redundant concepts in i2b2 if all contexts are extracted. The basic query selects the three warehouse columns that will form the Concept key in i2b2:

```

SELECT DISTINCT
    F.FACT_ID,
    C.CONTEXT_NAME      AS I2B2_CONCEPT_ID_1,
    R.REFERENCE_COMPOSITE AS I2B2_CONCEPT_ID_2,
    B.ROLE              AS I2B2_CONCEPT_ID_3
FROM FACT F
INNER JOIN CAREGIVER_B B ON (F.CAREGIVER_GROUP_ID = B.GROUP_ID)
INNER JOIN CAREGIVER_R R ON (B.MASTER_ID = R.MASTER_ID)
INNER JOIN CAREGIVER_C C ON (R.CONTEXT_ID = C.CONTEXT_ID
                            AND CONTEXT_NAME = '<selected extract context>')
WHERE F.FACT_ID IN ( <> fact-cohort selection query >> );

```

The DISTINCT clause in the query is needed because a single entry in a dimension could be referenced multiple times by a single fact in the same Role. That condition is rare but possible. If other analytical conditions need to be included as part of the design strategy, they can be added to this basic query.

If some of the conditions are orthogonal, the query can be split into multiple queries as long as they all produce the same output columns.

Some of your extraction queries might be more complicated depending upon what data you have in your warehouse. One common situation you need to be able to handle is when your Reference table contains aliases for a Definition in the same Context as the base reference. An example is when you've done patient merges in your Subject dimension resulting in many patients having multiple MRNs defining them. Only one MRN is current in the EHR, but your warehouse has all of them to support historical data. When this occurs, your extract will erroneously create multiple output rows. Depending upon how you've previously decided to handle and load this kind of event, the logic you have to add to the query to get the correct MRN will be unique to your design. I usually use the Vernacular Code column in the Definition table to store the most commonly used identifier so it is easily accessible to users. When I implement patient merge loads, I keep the current active MRN in the Vernacular Code. My Reference table has multiple MRNs per patient, but the Definition table has only one. Changing the extract to obtain a distinct MRN from the Definition table rather than from the Reference table solves the problem in my typical designs:

```
SELECT DISTINCT
    F.FACT_ID,
    '<selected context>' AS I2B2_CONCEPT_ID_1,
    D.VERNACULAR_CODE AS I2B2_CONCEPT_ID_2,
    B.ROLE AS I2B2_CONCEPT_ID_3
    FROM FACT F
    INNER JOIN SUBJECT_B B ON (F.SUBJECT_GROUP_ID = B.GROUP_ID)
    INNER JOIN SUBJECT_D D ON (B.MASTER_ID = D.MASTER_ID AND D.STATUS_INDICATOR = 'A')
    WHERE F.FACT_ID IN ( <> fact-cohort selection query >> );
```

Since this revised extract query obtains the required identifier from the Definition table, the Reference and Context tables can be omitted from the JOINS. The Context Name that would have been placed in the filter on the Context JOIN can be placed as a literal among the selected columns. If you aren't using the Vernacular Code column to resolve alias issues within your extracted context, you'll have to build a more complicated extraction query to avoid redundant entries.

If your design of Concepts in i2b2 includes cross-dimensional Concepts, you'll have to provide extract queries that pull the cross-dimensional pairings from the warehouse. A common cross-dimensional Concept set is the combination of Procedures and Diagnoses:

```
SELECT DISTINCT
    F.FACT_ID,
    PC.CONTEXT_NAME AS I2B2_CONCEPT_ID_1A,
    PR.REFERENCE_COMPOSITE AS I2B2_CONCEPT_ID_2A,
    PB.ROLE AS I2B2_CONCEPT_ID_3A,
    DC.CONTEXT_NAME AS I2B2_CONCEPT_ID_1B,
    DR.REFERENCE_COMPOSITE AS I2B2_CONCEPT_ID_2B,
```

```

DB.ROLE                                AS I2B2_CONCEPT_ID_3B
FROM FACT F
INNER JOIN PROCEDURE_B PB ON (F.PROCEDURE_GROUP_ID = PB.GROUP_ID)
INNER JOIN PROCEDURE_R PR ON (PB.MASTER_ID = PR.MASTER_ID)
INNER JOIN PROCEDURE_C PC ON (PR.CONTEXT_ID = PC.CONTEXT_ID
                             AND PC.CONTEXT_NAME = '<selected extract context>')
INNER JOIN DIAGNOSIS_B DB ON (F.DIAGNOSIS_GROUP_ID = DB.GROUP_ID)
INNER JOIN DIAGNOSIS_R DR ON (DB.MASTER_ID = DR.MASTER_ID)
INNER JOIN DIAGNOSIS_C DC ON (DR.CONTEXT_ID = DC.CONTEXT_ID
                             AND DC.CONTEXT_NAME = '<selected extract context>')
WHERE F.FACT_ID IN ( << fact-cohort selection query >> );

```

This query extracts a result row for each combination of Procedure and Diagnosis associated with the desired facts. Your load strategy for i2b2 will determine how you'll combine the six result identifiers to create the combined identifier you've chosen to include as an i2b2 Concept entry.

The data marts to which you deliver data from your warehouse are *extensions* of your warehouse. On a case-by-case basis, you'll have to decide how often you refresh the data you extract and send, and whether those refreshes will completely replace the original extracts or provide incremental updates. You'll also need to establish the level of control you give up in sending data to the mart. The more your warehouse controls are included within the extracted data, the less risk you incur in the use of the data in the target data mart. I recommend creating all extracts from the warehouse using a redacted or completely deidentified view of the warehouse unless your governance group specifically authorizes certain privacy-controlled data to be included. Under this scenario, your data mart users might prefer that any concepts extracted as Redacted be omitted from the extract. Either way, you've provided some additional protection. The protection of data identifiers causes additional problems since they can't be redacted because of the nature as identifiers in the data mart. A compromise that sometimes helps is to extract your Master ID from the Subject dimension rather than the privacy-controlled identifier. Sending the Master ID instead of an MRN improves control, because the Master ID only identifies the patient within the context of your warehouse, and you control who has that access. A Master ID can't be used outside of the warehouse the way an MRN can. If the owners of the data mart need to merge in other data from other sources, they are likely to insist on being given the MRN for each patient. Therefore, I also recommend that patients in privacy-protected classes be omitted from extraction cohorts unless specifically authorized by the IRB. As long as your governance group approves the extract, which also implies that the data mart owner has IRB approval to see the data in question, you will have done as much as you can.

Chapter 18

Finalizing Gamma

Verification and validation (V&V) of the final warehouse implementation represents an expanded challenge from the functional V&V that finalized the Beta version and will actually include all of that capability as well. You'll add criteria to the V&V strategy to encompass all functions and features implemented since the Beta version, with emphasis on operationalizing the verification and validation process so it becomes a regular operating component of the production warehouse, with data and results of the V&V process becoming further input into the warehouse support, data administration, and data governance functions in production. Testing isn't the end of the development of the warehouse; it's the start of continuous operation, with all the attendant problems of technical complexity and organizational psychology that you began to address in the Beta version.

Business Requirements

While each organization has different, and sometimes unique, requirements for its informatics toolset, a verification and validation strategy built around categories of universal requirements should be useful to every organization. To properly validate your warehouse implementation, it's a good idea to quickly revisit the business requirement categories we explored in Chapter 1:

- *Integrated data repository.* A warehouse that allows us to integrate our data, not just combine it. Just combining data wouldn't need a new data warehouse.
- *New and diverse access paths.* Allow access to data through pathways that might not traditionally be supported in operational healthcare systems, supporting pathways diverse enough to meet any need.
- *Enhanced data quality.* Improve our data as we bring it into the warehouse, reducing or eliminating data problems encountered in our source systems that are difficult or impossible to fix within our current constraints.

- *Early access to complicated data.* Gain access to complicated data quickly, preferably without having to carry out complicated projects. Data in hours or days, not months or years, are needed.
- *Scalable to include more sources.* Add more sources into the future without having to reinvest or redesign the warehouse structures or processes.
- *Semantic pathways into data.* Access data through meanings we add to the data in the warehouse rather than just through keys and pathways provided by our sources. Syntactic keys and text should be supplemented with semantic concepts and language.
- *Longitudinal phenotypic data.* Look at patient phenotypes over the long haul, not just within the internal or external windows provided by individual source systems or data feeds.

Implementing this warehouse design satisfies some of these requirements directly and enables the conformance to the rest, if the warehouse is used effectively.

I urge you to schedule and conduct frequent requirements and design reviews with key stakeholders throughout your project life cycle. These seven requirements should form a core part of the agenda in every session, constantly challenging you to demonstrate how and where your implementation is meeting these requirements. By the end of this Gamma stage, every key stakeholder should easily be able to point to the various functions and features of your implementation that drive compliance for these requirements. This issue goes beyond the requirements conformance itself to the greater advocacy among your stakeholders that is generated when they can clearly see how your new warehouse enables the core requirements that justified the initiative in the first place.

Technical Challenges

There was a time in my career when I considered myself fairly technical with respect to software systems and their operations, but that time has long passed. Putting the warehouses I've developed into production and operation is now always left to others who are more technical and qualified. By observation, though, I've been able to identify some of the key issues and challenges those people have faced, so I can point you in a general direction in this chapter. As you work with your technical staff toward some form of production turnover for Release 1.0, include these technical challenges in your discussions so your turnover team is aware of them and can consider them in their planning.

Configuration Management

Throughout the life cycle of your initiative, you've moved components of your system between your development and test environments, and now you'll move your system to a production environment. While your information technology

function probably has many tools to support this turnover process, it might not have tools to support the movement of your configuration data among these environments. These data include elements of your warehouse that you probably created manually in your development environment and might not have an automated source for loading to your production system.

The data that typically need to be transferred between environments include the contents of all your Context tables and the metadata and data state dimensions, as well as any system-level entries you included in your unit of measure dimension. There might be additional data if you developed any manual data structures in the dimension sets for which you have no source datasets. These might include customized code sets, hierarchy relationships, or specialized subdimensions that you might have created to solve specific problems during the earlier stages.

This issue is not unique to your Release 1.0 production turnover. You most likely encountered this need when transferring your Alpha version from the development to the test environment. You definitely should have encountered the issue when transferring your Beta version from development to test. If you kept your implementation very simple (which is what I've hoped for in keeping each stage down to a short period of months), it's possible that you transferred versions from development to test by simply copying the entire warehouse database from one environment to another. That process would have expedited your project, but it might leave you without having dealt with this data configuration issue until very late in your development.

I urge you to include this issue in your planning for your Beta stage so you have plenty of time to get it right. If you move an early iteration of your Beta system from development to test somewhere around the midpoint of the stage, you can practice a configured turnover at the end of Beta. You'll want to be able to move the second iteration of the Beta system into test without repeating the reloads of anything that is already in test. This will require you to think about how to move the net changing configuration data from one environment to the other in a managed way. You then have the remaining Gamma stage of the project to repeat the process several times to assure that your turnover strategy and tools are effective. I've seen two basic strategies for configuration data management used effectively in warehouses: direct data transfer, and resourcing of configuration data.

Direct data transfer involves directly transferring the rows of interest in any of the tables into the database of another environment. This approach can be effective if the data being transferred are straightforward, meaning it's tied to its own Master IDs and no others. The data can be read from one environment and inserted directly into the next environment. To avoid collisions on table keys, this approach requires that the seed value for the sequence generator in the target environment be set high enough that the generation of Master IDs in that environment won't reuse one of the key values that you transferred directly. This approach breaks down if part of your configuration involves Master IDs that

aren't within your configuration data, such as specialized hierarchy entries you might have created in some of the dimensions. The Master IDs of the ancestors or descendants might be different in the new environment, so data can't just be copied. It would have to be resourced in order to establish values using the new target Master IDs.

Resourcing is much safer than trying to direct data transfer of rows between different database environments, but it involves more development work on the front end to enable all of the needed transfers. This approach results in having a specialized ETL job that reads configuration data from one environment and loads into another environment (e.g., development to test, test to production). Much of the data that needs to be transferred isn't within the scope of data targeted by your generalized ETL subsystem, so specialized workflows are required. Because all of these data are within your control, I recommend developing the ETL transfer code at a capability roughly matching the work you did in the Alpha version. The Context tables, metadata and data state dimensions, and standard rows from the unit of measure dimension can be sourced and loaded in the same way you loaded early data in the Alpha version. The transfer is typically only conducted once per release configuration, so it doesn't justify building the capability into your generalized ETL subsystem.

Additional resourced configuration data can be loaded using the capabilities of the standard ETL jobs of the target environment. These data usually include subdimension, hierarchy, and codeset data that target tables in the design that are covered by your generalized logic. Creating Alpha-style code for these loads would be more work. The biggest advantage to resourcing over direct transfer is the reduced risk of data and key collisions because the Master IDs for the target environment are regenerated on the load, eliminating the possibility that a key from one environment will collide with a key from another.

If you focus attention on this data configuration issue at the end of Beta, it won't present major problems for you when Gamma is complete and you want to go into production. This ensures that once you move on to monthly or quarterly new releases of the warehouse, your configuration turnover activities will be carried out smoothly.

Job Scheduling

In addition to data configuration, job scheduling is often a key issue in production turnover of the new warehouse. This area is extremely dependent on the tools and practices available in your overall information system environment. The easy part is the actual scheduling and sequencing of the various jobs into which you've broken up your ETL subsystem. The difficulty is usually in integrating your strategy for trapping errors and recovering from problems into the job scheduler, and making sure that job failures in your production runs are visible to your warehouse support team.

I recommend tying your Beta version into your job scheduler so you have time throughout the project to test your scheduling and recovery integration. Your warehouse design is meant to be flexible enough that it should never fail during operation. Errors are trapped as logical control data within the warehouse (e.g., orphans, system rows, UOM conditions) rather than risking a job failure that might invalidate an entire daily load. One condition that can easily cause job failures at the database level is when database administrators slip database controls that should not be present into the design. Referential integrity controls are the most common point of failure during testing. The DBAs who insert these controls into the database usually mean well, but don't understand the generic nature of the ETL workflows that often insert data into the database in nonobvious ways. For example, if you've chosen to load your Fact table first because it is the largest load of the night, that load will fail if referential integrity has been turned on between the Fact table and the dimension Group tables. By the end of the runs, all referential integrity constraints are satisfied, but not at checkpoints within those runs. By integrating job control into the Beta version, these kinds of errors are identified and eliminated early.

Another aspect of the design to be considered during job scheduling is that all the ETL workflows are expected to commit data to the warehouse database as late as possible. This simplifies backup and restart planning and allows opportunities for your DBAs to consider converting some of those database insert routines into bulk loads. Because a bulk loading strategy carries implications and trade-offs involving dropping and creating indexes, I can't offer a universal recommendation on its viability. Each scenario needs to be analyzed within the context of your database and your environment. This requires a knowledgeable DBA, and it helps if she or he is available to the project team throughout the design and development life cycle.

Database Tuning

In addition to helping with restart and bulk loading decisions, your database administrator also plays a key role in tuning your warehouse database, and this often is needed with any significant upgrade or expansion of the warehouse. While your development, test, and production environments will be logically equivalent, they will differ significantly in many physical aspects. Using the same database configurations in all three environments will usually result in vast differences in operational performance, and these differences need to be balanced out by continual tuning.

In my experience, the servers on which you run your production warehouse are always more powerful than those on which you conduct your development and testing. That extra power doesn't necessarily result in better performance because it depends on what you are doing to improve the performance of your development and test environments. In development, extra indexes placed into databases to improve their performance on servers that have less

than the amount of required memory can speed up data access by providing the database optimizers alternative paths into commonly used tables. In production, commonly used tables can be pinned into the much larger memory space available on production servers, allowing very fast access to those tables directly. If not viewed broadly, the extra indexes added to the development environment end up slowing down the production environment because pinned tables need fewer indexes, and the extra indexes carry their own overhead.

I've also observed that databases in warehouses generally are tuned for optimal query performance, which makes sense given that the main use of the warehouse is for querying by users. On a day-to-day basis, this performance bias isn't a problem because the volume of daily ETL loads is small compared to the overall volume of data (e.g., typically millions of facts). However, historical loading of data has a very different profile (e.g., often hundreds of millions of facts), and it is historical loading that often dominates the first few days or weeks of a new warehouse release. Your database administrator should develop plans for varying the performance options placed in the warehouse between short-term (e.g., historical loads in new releases) and long-term (e.g., day-to-day operational loading and querying) processing stages.

Functional Challenges

Beyond the technical challenges of a production implementation, there are certain functional issues that arise during validation and turnover that need to be addressed in order to instill user confidence and provide effective control.

Patient Merge/Unmerge

One of the most important data sources to your new warehouse are data that signal either a patient merge is being conducted or a previous patient merge needs to be reversed. The merge action is not difficult to do, but is complicated by the fact that merges must be reversible. Another complication arises because merge and unmerge transactions are administrative transactions within the healthcare institution and can't be processed as simple data updates in the warehouse in the manner in which most sourced data are processed. I recommend building an additional processing pipe into your ETL environment to handle these transactions, as well as other functionally specific ETL transaction types that will emerge over time.

The source data that drive these transactions can take many forms, including several specific HL7 messages intended for patient and encounter merge transactions. Regardless of the type of source you obtain, the data will ultimately consist of a transaction with two patient identifiers—one indicated as the source of the merge or unmerge, and the other indicated as the target. There might be other data associated with the transaction that are also available in the source.

All of these source data can be loaded into the warehouse as a traditional Fact load, with one set of metadata defined for merge facts and another set of data defined for unmerge facts. In this manner, the warehouse includes a fact for each transaction, which provides some traceability and transparency to the processing that has been carried out. These facts don't require any special ETL capabilities beyond your standard fact load.

The same source data can also be sourced as a second logical dataset that includes only the two patient identifiers. The bridge role column in the metadata is used to identify which of the two identifiers is the source for the transaction and which is the target. The ETL transaction in this logical dataset is set to either merge or unmerge. This logical dataset passes through the ETL Reference pipe normally, providing for the resolution of the appropriate Master IDs for the two patient identifiers. However, the Definition and Fact pipes ignore these two transaction types, and they pass into your new ETL pipe for functionally specific transactions.

The transaction pathway for the merge updates the Reference table for the source identifier to point to the Master ID of the target patient. First, copy all of the facts associated with the source patient (in their current data state) to the target patient. Next, update all of the facts associated with the source patient to reflect the merged data state. To facilitate a possible future unmerge, the data states used to indicate that the facts have been merged should be specific to the original data states (e.g., MergedActive, MergedSuperseded) rather than just a general merged state since you will need to be able to determine later what the state of each fact was at the time of the merge. Finally, execute your superseding scripts against the target patient so the data inherited during the merge can result in some of the data for the original target patient changing to a superseded state.

The transaction pathway for unmerge updates reverses what the merge transaction implemented. To start, query the Fact table to ensure that the merge transaction that is being reversed actually occurred in the warehouse. One indication that this is so is that both source and target patient identifiers resolved to the same Master ID. If not, an unmerge transaction is logically impossible. If valid, query the Subject Definition table to find the row that has the target patient identifier in the Vernacular Code column to obtain the target Master ID for the unmerge target. You can also validate that this Master ID is correct by querying to confirm that there is no current Reference entry in the patient identifier context pointing to this Master ID. Once confirmed, you can update the Reference entry that contains the target patient identifier to point to this target Master ID.

The previously merged Fact table then needs to be reversed. Each fact under the target patient is reactivated by updating the data state based on the distinct data states that were used during the merge transaction process (e.g., MergedActive to Active, MergedSuperseded to Superseded). For each of those facts, the corresponding fact under the source patient must be updated by changing its data state to unmerged. This process could potentially leave a fact type without an active instance under the source patient if, at some point, one of

the unmerged facts resulted in the superseding of another fact under that patient so only the superseded fact now remains. You can fix this by rerunning your superseded scripts over the source patient, although you might have to tell that script to process both active and superseded data instead of the normal pathway of only looking at active data.

Merging and unmerging of patients is very complicated and is prone to error when other administrative actions have been taken against those patients during the time periods in question. This processing in a separate functional ETL pipe is an example of a general transaction type known as a *surrogate merge*. Patient merges are usually the first surrogate merges (and unmerges) you need to be able to handle, and their importance to the institution makes them critically important to your stakeholders' perceptions of the warehouse. Surrogate merges can occur in any dimension for a wide variety of reasons, so eventually you'll generalize this capability in your transactional ETL pipe. The patient merge and unmerge transactions are so central to your initial implementation that I don't recommend trying to implement a completely generic version of this transaction type until you've gained some experience with these two essential transactions.

HIPAA Redaction

The functions for redacting data in queries were covered in Chapter 17. One main issue that must be resolved before you implement the warehouse into production is the length of textual data that can be considered *nonidentifying* versus the length beyond which all textual data must be presumed to be identifying. Your governance group (discussed in Chapter 20) will set that value for you. In the absence of a governance decision, I recommend that you set the control value at three characters. A one-, two-, or three-character value is extremely unlikely to be patient identifying, yet will allow for the display of many fact values in the warehouse. Presuming that your governance group will set the control at a longer length, you need to implement it in Gamma before you go live, or else much textual data in the warehouse won't display during queries to the de-identified warehouse. Keep in mind that the two most common values in the Fact table are likely to be "Negative" and "Yellow." Those values won't display with a control length of three characters. Health data are dominated by textual values, so this capability must be resolved and implemented immediately.

While the value you use for control is a governance issue and must be decided by your governance function working with your HIPAA Controller, I recommend a control value of 10 characters. Most single-word simple value facts in the warehouse will consist of 10 or fewer characters. Values longer than 10 tend to be larger text descriptions or comments that are *much* longer, and we want these longer descriptions or comments redacted. In situations where shorter values might introduce risk, the Redaction Control Flag can be turned on to force redaction. If you treat this value as a parameter in your query toolset, the logic of redaction can be made independent of the value, and you will have an

opportunity to experiment with different control values to help the governance function select the most responsible, yet flexible, value for your warehouse.

IRB Integration

One final critical function that you need to provide as you move toward your Release 1.0 warehouse is the ability to manage which users are permitted to query identifiable patient data. For those granted access, you also need some control over which patients and properties they can see. While many issues need to be considered, and eventually designed, your initial focus should be on implementing the cohort definitions to which your IRB grants access permission to projects or individuals. Alternatively, you could identify a data source of patient identifiers who have signed consent for their data to be used for particular purposes. Both of these pathways result in knowing a list of patients—a cohort—to which identified access should be granted to an individual using the warehouse query toolset.

One strategy is to keep those capabilities externalized in your network security, database access security, and query toolset. A different semantic abstraction of the warehouse can be developed for each study cohort, such that only those patients in the cohort would be viewable through the query layer when viewing identifiable data. Another option is to implement different projects as logical database views of the warehouse that restrict access to only those patients for which access permission has been granted by the IRB. Protection can also be provided by offering a fully identified view of the warehouse under strict network and database security controls. A combination of these controls can be used to meet the immediate need in the initial warehouse, with more control or different controls being added later, as needed.

While these controls can be implemented without adding any data to the warehouse, there is an option that allows the warehouse to be used to, at least partially, generalize these solutions across different types of scenarios and purposes. If you can obtain a list of patients for an approved IRB project, I recommend loading that list as a cohort in your Subject dimension. The cohort identifier can be your IRB project identifier, and the list of patients is implemented as a dimension hierarchy, with each individual patient as a child of the new cohort row. Then, all that is needed is for your security or sign-on routine of your query toolset to pass an IRB project identifier as a parameter to the query session. Many of the controls required for IRB cohort management can be built directly into the standard query interface so the interface can automatically add filters to any generated query to ensure that identified data are only displayed for patients within approved cohorts.

Cohort display restrictions can take many forms. The simplest is to add an inner-join `WHERE` clause to the generated query, so only facts that are dimensionalized against the authorized patient identifiers can be returned. A more refined variant is to add the controlling `WHERE` clause only if one or more protected columns (as indicated by the values you have set in your

Redaction Control Flags) is included in the query. An even more flexible option is to add the controlling `WHERE` clause as a left-join condition, using your standard redaction capability to hide data for patients not in your cohort, while allowing all data to be displayed for patients in the project cohort. This last option is the most flexible for users who are querying their project cohort data while also looking for control group samples from outside of their cohort.

The privacy protection of data in the warehouse is very complex. The main complexity is that the IRB can approve access to data using almost any criteria that can be described. The most common I've seen is the cohort-based lists of patients, but sometimes there are restrictive time periods, or a specific list of procedures or results that can be reviewed. No matter what kind of flexibility you build into your IRB controls, in time, you'll encounter situations that you haven't seen before. Eventually you declare that you just can't automate it all. When this happens, you'll move toward an external data mart that you'll extract and supply to the project. Since the data mart can be defined to extract only the project's authorized data—using any arbitrary or specialized logic—the users on that project can be given unrestricted access to the data in their data mart.

Don't adopt the data mart strategy as your *default* strategy for supporting IRB projects. With some creative thinking, most project constraints can be built into your general model. Providing an external data mart, even if it conforms to the standard design of your main warehouse, adds support requirements for update and synchronization that can detract from your capacity to support the main warehouse. Only choose that path when the need can't be reasonably met with the generic functions. Based on my experience, I expect you'll build one or two data marts per hundred cohort-based projects you implement in the warehouse. Supporting a handful of data marts is preferable to supporting hundreds of them. I anticipate that the field-level security and control required of privacy protections in healthcare will increasingly become standard functions in the common database management environments, making IRB protection requirements less foreboding in the future than they seem today.

Going Live

It's important that you go live at this point in your initiative. The issues discussed in this chapter are among the last few significant capabilities or controls that you should deal with during the Gamma stage. Implement the Release 1.0 warehouse now. I've seen too many initiatives bog down at this point looking at all the remaining issues that *could* be addressed rather than stopping after the core issues that *must* be addressed. No matter how many issues you resolve now, you'll see more once you've gone live. Your immediate priority needs to be implementing the warehouse, so your user community can begin actively using it, and deriving value for the organization. Release 1.1 is around the corner. Those other issues can wait until then.

RELEASE 1.0

IV

The Release 1.0 warehouse is placed into production as a fully functional and secure data warehouse, with all scoped data up to the implementation date loaded, and an operating ETL environment that will keep the data current as you move forward. Finalizing the release typically takes a couple of months, depending upon local requirements and processes for setting up and finalizing the operational environment. The launch of the production warehouse is typically about a year after the project started.

Chapter 19

Knowledge Synthesis

Your data warehouse has been designed and built to be more than the sum of its parts, especially more than just the data loaded into it. Used to its fullest, the warehouse contains knowledge not available within any of its individual source systems. Some of that knowledge is generated by combining source system data in novel ways, and some knowledge needs to be coaxed out of the warehouse by adding semantic linkages that provide connectivity and visibility into the data. Whether explicitly connected or implicitly coaxed, this new knowledge is the primary purpose for the data warehouse, and much of it was enabled by the generic nature of the warehouse design. Nothing introduced into the design of the warehouse should inhibit these novel combinations and interconnections of data precisely because you can't predict where new knowledge will arise.

Knowledge synthesis and generation represents what you want to do with your warehouse once you've completed your implementation. Although we can't necessarily predict all of the queries that users of the warehouse will want to perform against the new Release 1.0 data warehouse, there are certain patterns that can be predicted in anticipation of many of the more common query and reporting scenarios.

This chapter covers some of the most common patterns that I see in usage of new data warehouses. I use them to help me explore the data in the warehouse, reinforce my understanding of how those data have been mapped into the generic architecture, and really start to see and use the data in novel and creative ways. These techniques are not exhaustive of what can be done with this new data resource, but they're representative of wide classes of activity that can become the core of a new informatics culture around the data warehouse.

Fact Counts

Among the first queries I write when I encounter a newly loaded warehouse is a set of fact count ranking queries. These are also the first queries I teach to users. Fact count rankings specifically serve as excellent training tools because they use

the internal structure of the dimensions and they combine data across sources in ways that many users don't intuitively do in their early queries. The results obtained by these queries are often of a type and complexity that could not be obtained through any other system in the institution. It is precisely this novelty that makes them so powerful as training tools.

If new users query the production warehouse to get the data they've always gotten from other sources, they could find their experience with the warehouse unsatisfying. There's a learning curve required to master the new data warehouse, and if users come up that curve just to see the same data they've always seen, they'll quickly tire of the extra effort. These fact counts are useful because they quickly combine data in ways that couldn't be accomplished in other systems, while being easy to query precisely because they directly take advantage of the generic structure of the warehouse design. Users learn to write solid queries, and they see new data in novel ways. Their positive reactions create enthusiasm for the new warehouse, and those users become your best advocates for selling the warehouse across the organization.

Fact Counts by Metadata

A fact count query is what its name implies, a simple count of facts in the Fact table. The queries are different to the extent that those counts are aggregated by different aspects of the warehouse model. The first query everyone writes is the fact count by metadata, a simple join of the facts in the Fact table to the metadata that defines them:

```
SELECT SOURCE, EVENT, ENTITY, ATTRIBUTE, COUNT(*)
  FROM FACT
INNER JOIN METADATA_D META ON FACT.METADATA_MASTER_ID = META.MASTER_ID
 GROUP BY SOURCE, EVENT, ENTITY, ATTRIBUTE
 ORDER BY SOURCE, EVENT, ENTITY, ATTRIBUTE;
```

You've been using queries like this one throughout your verification and validation activities in the Alpha, Beta, and Gamma versions. It's a very flexible way to see what's going on in the Fact table at any given time because additional conditions can be added to the simple core query to see what is in the warehouse from different vantage points. These variations get a bit more complicated in two basic ways: (a) by filtering the data to look at only selected subsets of the facts or (b) by adding properties to the grain of the counts to get more detailed views.

Among the commonly used filters, users might only look at data from certain source systems by filtering on the Source column, or a particular set of loads by filtering on the Datafeed dimension. Additional grain might include the unit of measure (UOM) to see how many different ways each particular fact was loaded or the data state to see the impact that different transactions have on the proliferation of fact variants. Aspects used for filtering can also be exchanged

with aspects used for grain, with filtering affecting the breadth of the query, while grain affecting the depth of the results. For example, a filter might constrain data to only quantitative units of measure, while adding data state to the grouped SELECT provides a deeper level of detail for those quantitative facts. Useful permutations are numerous, so I concentrate on making sure users understand how to build and use general queries. I'm often surprised by the creativity they put into these simple exploratory views of the data.

Ranked Dimensioned Facts

The fact count by metadata is a special case of counting of facts against a single dimension. There's no reason why the counts have to be against metadata, except that it is a good way for users to get accustomed to the idea that metadata defines the data in the Fact table. Counts of facts against other dimensions get more interesting and begin to have clinical relevance to many users. This query gives the simplest version of a fact count by diagnosis, with the counts appearing in descending order:

```
SELECT D.DESCRIPTION, COUNT(*)
  FROM FACT
  INNER JOIN DIAGNOSIS_B B ON (FACT.DIAGNOSIS_GROUP_ID = B.GROUP_ID)
  INNER JOIN DIAGNOSIS_D D ON (D.MASTER_ID = D.MASTER_ID
                                AND D.STATUS_INDICATOR = 'A')
 GROUP BY D.DESCRIPTION
 ORDER BY COUNT(*) DESCENDING;
```

The count at the top of the list is the most active row in the dimension. Notice that I didn't say it is the most active *diagnosis*. Predicting what this query will produce requires anticipating what kinds of facts have been loaded into the warehouse. Becoming familiar with the warehouse data, typically through metadata-fact-count queries, allows users to design more meaningful queries.

One thing I can predict with reasonable certainty at this point is that the top entry in the query results will be the Not Applicable system row in the dimension. Typically, there are more facts in the Fact table that didn't dimensionalize to *any* Diagnosis than did dimensionalize to any *particular* row. This is likely to be true for any of the dimensions, although there will be exceptions. For example, only rare facts don't make use of the Subject or Organization dimensions. Most facts involve the Interaction dimension. Even the Diagnosis dimension could be an exception if you've chosen to load only data that use Diagnosis, but I've never seen anyone get past the Gamma version with that staying true.

Likewise, if you've loaded many different kinds of data into the Diagnosis dimension, your ranked list could be a kaleidoscope of different entries. In order

to better organize all of these results, we typically modify the base ranking query to account for system rows and subdimensions:

```
SELECT D.SUBDIMENSION, D.DESCRIPTION, COUNT(*)
  FROM FACT
  INNER JOIN DIAGNOSIS_B B ON (FACT.DIAGNOSIS_GROUP_ID = B.GROUP_ID)
  INNER JOIN DIAGNOSIS_D D ON (D.MASTER_ID = D.MASTER_ID
                                AND D.STATUS_INDICATOR = 'A'
                                AND D.SUBDIMENSION <> 'System')
 GROUP BY D.SUBDIMENSION, D.DESCRIPTION
 ORDER BY COUNT(*) DESCENDING;
```

Including subdimension in the query results provides more context for each count, particularly for novice users who are less familiar with data in the warehouse. Whether users want to filter on a particular subdimension, when running the query, is a function of what they are looking for. I find that ranking the usage of one subdimension at a time is most meaningful to novice users. Seeing the top 10 diagnoses carries more “Wow!” factor than seeing the top 10 list from a combination of diagnoses, pathologies, problems, or reactions. The extent of any differences across subdimensions is purely an accidental function of the range of data that is being placed in the dimension.

So far, the ranked lists being generated by these queries are curiosities. Novice users probably find them interesting, and experienced user can probably make clinical sense of what they are seeing, but we want these queries to be generally more powerful. As an example, let’s look at a simple set of hierarchic relationships that might exist among diagnosis entries in the dimension for an organization using ICD-9 codes (Figure 19.1).

In the ranked counts produced so far, the top two entries in this hierarchy don’t appear at all because they are not diagnosis codes that would show up as the dimensionalization of a fact. They are *aggregates*. The lower three codes are likely in the result list, but they are reported as individual distinct diagnoses. The notion that the count for ICD-9 code 410 should include any counts for codes 410.0 or 410.01 is not present in the query. As a result, the 410 diagnosis code shows up in the ranked list well *below* the level at which a clinical user would expect it. This is

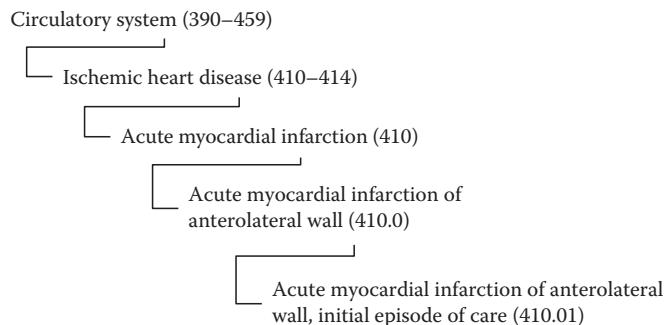


Figure 19.1 Example five-tier diagnostic hierarchy with aggregates.

the opposite of what a user would need. The 410.0 and 410.01 diagnoses are more specific than the more general 410, but they are in fact also 410 diagnoses.

We can easily fix these problems by querying the fact counts across the dimension hierarchy, so the counts automatically aggregate up into their appropriate summary codes:

```
SELECT A.SUBDIMENSION, A.DESCRIPTION, COUNT(*)
  FROM FACT
INNER JOIN DIAGNOSIS_B B ON (FACT.DIAGNOSIS_GROUP_ID = B.GROUP_ID)
INNER JOIN DIAGNOSIS_D D ON (B.MASTER_ID = D.MASTER_ID
                             AND D.STATUS_INDICATOR = 'A'
                             AND D.SUBDIMENSION <> 'System')
INNER JOIN DIAGNOSIS_H H ON (B.DESCENTENT_MASTER_ID = D.MASTER_ID
                             AND H.STATUS_INDICATOR = 'a')
INNER JOIN DIAGNOSIS_D A ON (A.MASTER_ID = B.ANCESTOR_MASTER_ID
                             AND A.STATUS_INDICATOR = 'A'
                             AND A.SUBDIMENSION <> 'System')
GROUP BY A.SUBDIMENSION, A.DESCRIPTION
ORDER BY COUNT(*) DESCENDING;
```

The result of this revised query is a ranked list much closer to what a clinically oriented user would want and expect. Filtering the query is important now because without filtering, the hierarchies will dominate the results. With filtering, the top aggregate in our “Circulatory system (390–459)” example will, by definition, have the highest count of the five entries in [Figure 19.1](#). The top ranked diagnosis will always be an aggregate. As dimension hierarchies get richer and more complicated, users will typically need to be slightly more specific in filtering these count-based queries. A common variant of the query is to filter on only the specific ICD-9 coded diagnosis subdimension, omitting the aggregate entries, so the most common ICD-9 diagnosis (including subdiagnosis variants) will be at the top of the ranked list. This fits with user expectations that a list of the most common diagnoses will be a list of three-digit ICD-9 codes without decimal qualifiers.

Depending upon what you’ve loaded into your warehouse, there might still be a lot of noise in your ranked list query results. If you loaded multiple hierarchies into the dimension, you might be getting double or triple counting across certain descendent-to-ancestor pairs. Users can add filters to this query to include only the perspectives and relationships they are interested in. Likewise, if a large variety of facts is dimensionalized to this dimension, particularly if there are a lot of roles defined in the bridge structures, they should filter their queries on either roles of interest in their rankings or metadata of interest to look at only diagnostically related facts of interest. If they filter out the obvious things that can be excluded, their results will start to look more meaningful. Remember that all we’re doing here is counting facts based on some very simple criteria. Precision isn’t necessary for these ranked lists to get users very excited.

One final filter that I add to the basic ranking query is a calendar filter, typically on the calendar year of the fact. So far, our list is of ranked diagnoses

back to the beginning of whatever time periods are loaded into the warehouse. Ranked lists are helpful in seeing how the list itself changes with time, so *time* should be part of the core query. Let's try limiting the query so our counts rank the results by calendar year:

```
SELECT CD.CALENDAR_YEAR, DA.SUBDIMENSION, DA.DESCRIPTION, COUNT(*)
  FROM FACT
  INNER JOIN CALENDAR_B CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
  INNER JOIN CALENDAR_D CD ON (CD.MASTER_ID = CB.MASTER_ID
                               AND CD.STATUS_INDICATOR = 'A')
  INNER JOIN DIAGNOSIS_B DB ON (FACT.DIAGNOSIS_GROUP_ID = DB.GROUP_ID)
  INNER JOIN DIAGNOSIS_D DD ON (DB.MASTER_ID = DD.MASTER_ID
                               AND DD.STATUS_INDICATOR = 'A'
                               AND DD.SUBDIMENSION <> 'System')
  INNER JOIN DIAGNOSIS_H DH ON (DB.DESCENDENT_MASTER_ID = DD.MASTER_ID
                               AND DH.STATUS_INDICATOR = 'A')
  INNER JOIN DIAGNOSIS_D DA ON (DA.MASTER_ID = DB.ANCESTOR_MASTER_ID
                               AND DA.STATUS_INDICATOR = 'A'
                               AND DA.SUBDIMENSION <> 'System')
 GROUP BY DA.SUBDIMENSION, DA.DESCRIPTION
 ORDER BY CD.CALENDAR_YEAR DESCENDING, COUNT(*) DESCENDING;
```

This revised query provides a ranked list for each year in which there are facts to be counted. Be careful not to overinterpret these results. Historical fact counts are typically lower because you loaded less data in earlier years. The result rankings might or might not be impacted based on a historical load bias.

When the basic ranking query is working effectively, you can make queries even more interesting with some creative filtering. The rankings we've seen so far are limited only by the filtering we've built into the query on the dimension of interest. Suppose we want to go a bit deeper and see not just the top 10 diagnoses of 2014, but the top 10 diagnoses by patient gender. All we need to do to get that result is add *patient gender* to the count aggregation:

```
SELECT CD.CALENDAR_YEAR, SD.GENDER, DA.SUBDIMENSION, DA.DESCRIPTION, COUNT(*)
  FROM FACT
  INNER JOIN SUBJECT_B SB ON (FACT.SUBJECT_GROUP_ID = SB.GROUP_ID)
  INNER JOIN SUBJECT_D SD ON (SD.MASTER_ID = SB.MASTER_ID
                             AND SD.STATUS_INDICATOR = 'A')
  INNER JOIN CALENDAR_B CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
  INNER JOIN CALENDAR_D CD ON (CD.MASTER_ID = CB.MASTER_ID
                             AND CD.STATUS_INDICATOR = 'A'
                             AND CD.CALENDAR_YEAR = '2014')
  INNER JOIN DIAGNOSIS_B DB ON (FACT.DIAGNOSIS_GROUP_ID = DB.GROUP_ID)
  INNER JOIN DIAGNOSIS_D DD ON (DB.MASTER_ID = DD.MASTER_ID
                             AND DD.STATUS_INDICATOR = 'A'
                             AND DD.SUBDIMENSION <> 'System')
  INNER JOIN DIAGNOSIS_H DH ON (DB.DESCENDENT_MASTER_ID = DD.MASTER_ID
                             AND DH.STATUS_INDICATOR = 'A')
  INNER JOIN DIAGNOSIS_D DA ON (DA.MASTER_ID = DB.ANCESTOR_MASTER_ID
                             AND DA.STATUS_INDICATOR = 'A'
                             AND DA.SUBDIMENSION <> 'System')
 GROUP BY DA.SUBDIMENSION, DA.DESCRIPTION
 ORDER BY CD.CALENDAR_YEAR DESCENDING, SD.GENDER, COUNT(*) DESCENDING;
```

Alternatively, we might want to see the top 10 diagnoses by *encounter type*, to look at differences in diagnostic patterns between different types of encounters:

```

SELECT CD.CALENDAR_YEAR, ID.ENOUNTER_TYPE, DA.SUBDIMENSION, DA.DESCRIPTION,
COUNT(*)
  FROM FACT
INNER JOIN INTERACTION_B IB ON (FACT.INTERACTION_GROUP_ID = IB.GROUP_ID)
INNER JOIN INTERACTION_D ID ON (ID.MASTER_ID = IB.MASTER_ID
                                 AND ID.STATUS_INDICATOR = 'A')
INNER JOIN CALENDAR_B CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
INNER JOIN CALENDAR_D CD ON (CD.MASTER_ID = CB.MASTER_ID
                             AND CD.STATUS_INDICATOR = 'A'
                             AND CD.CALENDAR_YEAR = '2014')
INNER JOIN DIAGNOSIS_B DB ON (FACT.DIAGNOSIS_GROUP_ID = DB.GROUP_ID)
INNER JOIN DIAGNOSIS_D DD ON (DB.MASTER_ID = DD.MASTER_ID
                             AND DD.STATUS_INDICATOR = 'A'
                             AND DD.SUBDIMENSION <> 'System')
INNER JOIN DIAGNOSIS_H DH ON (DB.DESCENDENT_MASTER_ID = DD.MASTER_ID
                             AND DH.STATUS_INDICATOR = 'A')
INNER JOIN DIAGNOSIS_D DA ON (DA.MASTER_ID = DB.ANCESTOR_MASTER_ID
                             AND DA.STATUS_INDICATOR = 'A'
                             AND DA.SUBDIMENSION <> 'System')
GROUP BY DA.SUBDIMENSION, DA.DESCRIPTION
ORDER BY CD.CALENDAR_YEAR DESCENDING, ID.ENOUNTER_TYPE, COUNT(*) DESCENDING;

```

The results of this version of our ranking query vary based on how our organization has populated characteristics like encounter type, but different results for inpatient, ambulatory, or emergency encounters are common. Usually there are several columns of data in the encounter that can be used to make even more interesting combinations of these results depending upon what's populated. For training purposes, I often use inpatient labor and delivery encounters precisely because the diagnoses differ from the other lists in very predictable and understandable ways. Another interesting variant, assuming there are discriminating data in a dimension property, is to look at top diagnoses against traffic or workplace accident encounters.

These kinds of fact count ranking queries can be built for any of the dimensions in the warehouse, filtering on any property that helps make the resulting lists clinically meaningful. Ranking lists based on unfiltered data can easily be created, but the meaning behind the results will not necessarily be satisfying unless some thought is put into the design of the filters and aggregates, by someone qualified to interpret the data. As a software engineer, my exposure to biomedical data was nonexistent until about a decade ago when I started working in healthcare. With several years of experience, now I feel comfortable designing some fairly meaningful queries in the healthcare and biomedical arena, but I still rely heavily on specialists in the field for the most meaningful and valuable queries. Fortunately, due to the generic structure of the data warehouse, the jump from basic, mediocre queries to advanced, insightful ones isn't difficult. Even the simple queries I feel competent to write can form a backbone for more expanded usage by knowledgeable users.

Correlation Analysis

My experience is that new users of the warehouse really enjoy writing ranked dimension queries. It gives them a chance to explore the data, learn the tools, and sometimes gain some insight into their data in ways they've never been able to do before. After a while they start to look for even more creative ways to review their data. I find that once they've got their ranked lists, correlation analysis typically comes next.

Having built dimensional ranking queries against Diagnosis, Procedure, and Material dimensions, users start to ask combination questions. Instead of the top 25 diagnoses, procedures, and materials for a certain analytical frame, they start asking what the top 5 procedures were for each of the top 25 diagnoses, or they want to know the top 3 drugs for each of the top 10 diagnoses. They're asking for correlations across dimensions, but they don't necessarily know it yet.

These queries are easy to write using the standard framework for ranked list queries. If you want to know the top procedures for each of the top diagnoses, make the diagnosis ranking query a subquery to the high-level procedure ranking query, where the higher-level query connects the two pieces of interest: diagnosis and procedure.

How these two query halves can get connected depends upon what data were loaded into the warehouse and how it was dimensionalized. If there are facts that are dimensionalized to both diagnosis and procedure, the query is a very simple modification of the ranking queries we've been using:

```

SELECT CD.CALENDAR_YEAR,
       DA.SUBDIMENSION, DA.DESCRIPTION, DIAG_COUNT,
       PA.SUBDIMENSION, PA.DESCRIPTION, COUNT(*)
  FROM FACT
 INNER JOIN CALENDAR_B   CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
 INNER JOIN CALENDAR_D   CD ON (CD.MASTER_ID = CB.MASTER_ID
                               AND CD.STATUS_INDICATOR = 'A')
 INNER JOIN PROCEDURE_B  PB ON (FACT.PROCEDURE_GROUP_ID = DB.GROUP_ID)
 INNER JOIN PROCEDURE_D  PD ON (DB.MASTER_ID = DD.MASTER_ID
                               AND DD.STATUS_INDICATOR = 'A'
                               AND DD.SUBDIMENSION <> 'System')
 INNER JOIN PROCEDURE_H  DH ON (DB.DESCENTANT_MASTER_ID = DD.MASTER_ID
                               AND DH.STATUS_INDICATOR = 'A')
 INNER JOIN PROCEDURE_D  PA ON (DA.MASTER_ID = DB.ANCESTOR_MASTER_ID
                               AND DA.STATUS_INDICATOR = 'A'
                               AND DA.SUBDIMENSION <> 'System')
 INNER JOIN DIAGNOSIS_B  DB ON (FACT.DIAGNOSIS_GROUP_ID = DB.GROUP_ID)
 INNER JOIN DIAGNOSIS_D  DD ON (DB.MASTER_ID = DD.MASTER_ID
                               AND DD.STATUS_INDICATOR = 'A'
                               AND DD.SUBDIMENSION <> 'System')
 INNER JOIN DIAGNOSIS_H  DH ON (DB.DESCENTANT_MASTER_ID = DD.MASTER_ID
                               AND DH.STATUS_INDICATOR = 'A')
 INNER JOIN DIAGNOSIS_D  DA ON (DA.MASTER_ID = DB.ANCESTOR_MASTER_ID
                               AND DA.STATUS_INDICATOR = 'A'
                               AND DA.SUBDIMENSION <> 'System'),
       (SELECT CD.CALENDAR_YEAR          AS DIAG_YEAR,

```

```

DA.SUBDIMENSION      AS DIAG_SUBMENSION,
DA.DESCRIPTION       AS DIAG_DESCRIPTION,
COUNT(*)            AS DIAG_COUNT
FROM FACT
INNER JOIN CALENDAR_B CB ON (FACT.CALENDAR_GROUP_ID = CB.GROUP_ID)
INNER JOIN CALENDAR_D CD ON (CD.MASTER_ID = CB.MASTER_ID
                             AND CD.STATUS_INDICATOR = 'A')
INNER JOIN DIAGNOSIS_B DB ON (FACT.DIAGNOSIS_GROUP_ID = DB.GROUP_ID)
INNER JOIN DIAGNOSIS_D DD ON (DB.MASTER_ID = DD.MASTER_ID
                             AND DD.STATUS_INDICATOR = 'A'
                             AND DD.SUBDIMENSION <> 'System')
INNER JOIN DIAGNOSIS_H DH ON (DB.DESCENTDENT_MASTER_ID = DD.MASTER_ID
                             AND DH.STATUS_INDICATOR = 'A')
INNER JOIN DIAGNOSIS_D DA ON (DA.MASTER_ID = DB.ANCESTOR_MASTER_ID
                             AND DA.STATUS_INDICATOR = 'A'
                             AND DA.SUBDIMENSION <> 'System')
GROUP BY DA.SUBDIMENSION, DA.DESCRIPTION ) DIAG
WHERE CD.CALENDAR_YEAR = DIAG_YEAR
  AND DA.SUBDIMENSION = DA.SUBDIMENSION
  AND DA.DESCRIPTION = DA.DESCRIPTION
GROUP BY DA.SUBDIMENSION, DA.DESCRIPTION
ORDER BY CD.CALENDAR_YEAR,
        DIAG_COUNT DESCENDING, DA.SUBDIMENSION, DA.DESCRIPTION,
        COUNT(*) DESCENDING, PA.SUBDIMENSION, PA.DESCRIPTION;

```

The linking of two ranked lists presents a form of very rudimentary correlation. The most common procedures used for the most common diagnoses are a ranked list of procedures within the already ranked list of diagnoses. Two variables are correlated if one can be used to predict the other. A given diagnosis will be associated with many procedures, but some diagnoses will have ranked procedure lists where the counts fall dramatically after only three or four procedures, while others will have counts that remain stable across dozens and dozens of procedures. Clinical users easily spot and appreciate these differences.

Derivative Data

Novice warehouse users tend to perform queries that directly use the data in the warehouse, often in ways that seem quite natural given the types and volumes of data that have been loaded. More experienced users, who have gained experience and advanced knowledge of the data, tend to perform more sophisticated queries, often involving complicated subqueries that derive additional data from the core data. By monitoring this advanced usage, you can discern additional data that might be valuable, and make it available directly by deriving it explicitly and then loading it into the warehouse as new facts. In this way, the power of that derivative data is available for novice users, and the automatic derivation simplifies querying for more advanced users.

A simple definition of derivative data is that it includes any data facts that are sourced exclusively from the warehouse itself in specialized source-data-intake

jobs designed for that purpose. Since the range of data that *can* be derived is almost infinite, you'll want to define your derivative data based on actual usage of *your* data by *your* expert users, making every implementation different because it is grounded in that usage. The warehouse support team will continuously derive data commonly subqueried by users. As users look at data in novel ways not otherwise possible in the original source systems, their usage will uncover additional possible data derivations in a continuous cycle.

Inter-event Timings

A common derived value is the response time between two related events. Suppose you observe that many expert query writers are deriving the number of minutes that elapse between the placing of a lab order and the recording of the results for that order. The warehouse contains facts associated with both the order and the results, which are dimensionalized to the Calendar and Clock dimensions. The number of minutes between these two events is a simple calculation, but one that adds complexity to a query that includes it because the related facts typically need to be selected separately in subqueries, with the necessary temporal calculation of the gap as part of the main query join.

The classic inter-event timing requirement in healthcare is the length of stay associated with a hospital encounter (defined as a metric value back in Chapter 16). For an inpatient stay, the length of stay is the time between the encounter's admission timestamp and the encounter's discharge timestamp. If both timestamp values were loaded to the definition table of the Interaction dimension, the calculation is routine. If the two timestamp values are not in the dimension, you have to query the admission and discharge facts in order to obtain timestamps for each. The two individual timestamps obtained are pivoted into a single row for calculating the length of stay:

```

INNER JOIN INTERACTION_B IB ON IB.GROUP_ID = F.INTERACTION_GROUP_ID
INNER JOIN INTERACTION_D ID ON ID.MASTER_ID = IB.MASTER_ID
    AND ID.STATUS_INDICATOR = 'A'
INNER JOIN CALENDAR_B CB     ON CB.GROUP_ID = F.CALENDAR_GROUP_ID
    AND CB.ROLE = 'Event'
INNER JOIN CALENDAR_D CD     ON CD.MASTER_ID = CB.MASTER_ID
    AND CD.STATUS_INDICATOR = 'A') D
ON A.ENCOUNTER_ID = D.ENCOUNTER_ID;

```

This kind of time duration calculation can be done between any two events in the Fact table in exactly the same way. The inter-event timings query is algorithmically standardized and can be implemented in many of the popular business intelligence toolsets.

Census Data

As a follow-on to the inter-event timing query, another common data derivation is the census query. A census query looks at the presence of some entity during the time interval between two events. The two events are accessed in the same manner as for the inter-event timing query. To produce a hospital census, retrieve and pair the admission and discharge timestamps for each inpatient encounter. Next, cross-join the resulting list of encounters to the Calendar dimension to create a distinct new fact for each day in the calendar between the admission and discharge dates of each encounter. The aggregate count of those facts is the census in the hospital on that day:

```

SELECT CD.CALENDAR_DATE, COUNT(*) AS CENSUS
    FROM CALENDAR_D CD
INNER JOIN
    (SELECT A.ENCOUNTER_ID, ADMISSION_DATE, DISCHARGE_DATE
        FROM ... ON A.ENCOUNTER_ID = D.ENCOUNTER_ID) LOS
        ON CD.CALENDAR_DATE BETWEEN LOS.ADMISSION_DATE AND LOS.DISCHARGE_DATE
GROUP BY CD.CALENDAR_DATE;

```

I tend to refer to queries like these as being a “cross-join” to the Calendar dimension. Technically, as the example shows, these queries do involve a WHERE clause, so they are actually INNER JOIN queries. My use of the term “cross-join” is meant only to indicate that the join is taking place without using of the normal pathway through the Calendar Bridge table.

Some census queries are more granular with respect to time. If you take the arrival and departure times for each encounter in each ambulatory unit and cross-join to the Clock dimension to create a distinct fact for each hour in the day between arrival and departure, the aggregate count of those facts is the census by hour in each unit. For ambulatory units that are active around the clock, the Calendar dimension also needs to be included in the calculation so encounters spanning midnight are calculated correctly.

Timeline Analysis

Another common form of early data access that goes beyond these fact-counting applications is timeline query (introduced back in Chapter 17). A timeline query results in a longitudinal view of some dimensional element of the warehouse, with facts listed in temporal order. The examples in Chapter 17 are the two most common timelines produced in training: the patient timeline. This kind of timeline presents all relevant facts for a patient, sorted by time. A patient-encounter version of the timeline includes encounter in the sort order between patient and timeframe for cases where overlapping encounters might be undesirable for the user:

```

SELECT FACT SUBJECT_MASTER_ID AS SUBJECT,
       FACT INTERACTION_MASTER_ID AS INTERACTION,
       CALENDAR_D SORT_DATE AS DATE,
       CLOCK_D SORT_TIME AS TIME,
       META SUBJECT,
       META EVENT,
       META ENTITY,
       META ATTRIBUTE,
       CASE
         WHEN META REDACTION_CONTROL_FLAG = 'N'
             THEN WHEN UOM_D REDACTION_CONTROL_FLAG = 'Y'
                  AND LENGTH(FACT VALUE) > 15
                  THEN '~Redacted~'
                  ELSE FACT VALUE
                  ELSE '~Redacted~'
         END AS VALUE,
       UOM_D ABBREVIATION AS UOM,
FROM FACT, METADATA_D META,
      UOM_D, UOM_B,
      CALENDAR_D, CALENDAR_B,
      CLOCK_D, CLOCK_B
WHERE FACT METADATA_ID = META MASTER_ID
  AND FACT CALENDAR_GROUP_ID = CALENDAR_B GROUP_ID
  AND CALENDAR_B MASTER_ID = CALENDAR_D MASTER_ID
  AND CALENDAR_B ROLE = 'Event'
  AND CALENDAR_B RANK = 1
  AND CALENDAR_D STATUS_INDICATOR = 'A'
  AND FACT CLOCK_GROUP_ID = CLOCK_B GROUP_ID
  AND CLOCK_B DIMENSION_ID = CLOCK_D DIMENSION_ID
  AND CLOCK_B ROLE = 'Event'
  AND CLOCK_B RANK = 1
  AND CLOCK_D STATUS_INDICATOR = 'A'
  AND FACT UOM_GROUP_ID = UOM_B GROUP_ID
  AND UOM_B MASTER_ID = UOM_D MASTER_ID
  
```

```

AND UOM_B.ROLE          = 'Value'
AND UOM_B.RANK          = 1
AND UOM_D.STATUS_INDICATOR = 'A'

ORDER BY SUBJECT_B.MASTER_ID, INTERACTION_B.MASTER_ID,
CALENDAR_D.SORT_DATE, CLOCK_D.SORT_TIME,
META.SUBJECT, META.EVENT, META.ENTITY, META.ATTRIBUTE;

```

Note that in this example, the query uses the Master ID from both the Subject and Interaction dimensions as proxy values for the display and ordering of data. I recommend doing this to avoid accidentally displaying protected health information, although the acceptability of my proxy approach should be approved by your HIPAA controller. There will certainly be situations where you need to more completely join your query to those dimensions in order to add filtering to the query or add nonprotected data to the display. Check with your HIPAA governance group (discussed in Chapter 20) regarding an appropriate level of control for these simply introductory queries.

The basic timeline query results in a time-ordered list of facts with proxy patient and encounter identifiers. Each fact includes metadata tags, the fact value (if not protected), and the UOM for that value. If the encounter is included in the sort order of the results, the displayed facts will only be in temporal order within individual encounters. Including it only alters the results if patients have overlapping encounter data. This does happen often, even though many users think it doesn't. Remember that all the data for an encounter doesn't necessarily all fall between the admission and discharge. Scheduling and preliminary lab tests for some encounters occur weeks or months in advance. Some billing and payment facts can continue for years after discharge. As a result, even in organizations that don't experience much clinical overlap of encounters for a patient, there is often administrative overlap that causes these timelines to become confusing if the encounter is not included in the sort order of the results. When in doubt, leave encounter in the ordering.

As we saw in Chapter 17, this query can be supplemented, like our ranked lists, with any variety of additional dimension joins or filters to make it more useful and informative. Many facts in the list lack an appropriate context unless descriptions from one or more dimensions are included. The exact dimensions to be included depend upon the facts of greatest interest. When the facts include lab data, the Procedure dimension is usually included so the lab panels or tests are displayed. When medication data are present, the Material dimension is included to provide the names of the medications ordered or administered. For admission or discharge facts, the Facility dimension is included to provide information on the unit, room, or bed involved in the fact. In theory, all dimensions that might be needed by a fact can be included in a timeline query to maximize the information displayed. In practice, however, including all of the possible dimensions makes the query unmanageable as the result set gets

wider and wider. Experienced users learn to include a few commonly needed dimensions and to rerun the query with additional dimensions depending upon their purpose in conducting the queries.

Nonpatient Timelines

Patient-oriented timelines are a natural query for novices to use to become familiar with the data in the new warehouse. However, they quickly run into restrictions on protected data that make the results less useful. Many of the facts in the timeline display as redacted data because of privacy restrictions on the use of the warehouse for deidentified queries. Data available in a patient timeline are also already readily available in the organization's electronic health record for authorized users. The warehouse is actually far more useful for looking at data in any dimension other than Subject. It is the alternative dimensional timelines that will provide views of the data not available anywhere else.

I have found that new users of the warehouse get very excited over the ranked list queries discussed earlier. A natural progression for those users is into rudimentary timeline analysis. Suppose a user has been querying the organization's most frequently used procedures. A natural follow-on query is to develop a timeline query for each of the highest-ranking procedures. The query looks just like the earlier patient-centric query, except this query is built using the procedure description as its highest ordering value:

```

SELECT PROCEDURE_D.DESCRIPTION,
       PROCEDURE_D.MASTER_ID,
       CALENDAR_D.SORT_DATE           AS DATE,
       CLOCK_D.SORT_TIME             AS TIME,
       META.SUBJECT,
       META.EVENT,
       META.ENTITY,
       META.ATTRIBUTE,
       CASE
           WHEN META.REDACTION_CONTROL_FLAG = 'N'
               THEN WHEN UOM_D.REDACTION_CONTROL_FLAG = 'Y'
                     AND LENGTH(FACT.VALUE) > 15
                         THEN '~Redacted~'
                         ELSE FACT.VALUE
                     ELSE '~Redacted~'
               END
                   AS VALUE,
       UOM_D.ABBREVIATION           AS UOM,
FROM  FACT, METADATA_D META,
      PROCEDURE_D, PROCEDURE_B,
      UOM_D, UOM_B,
      CALENDAR_D, CALENDAR_B,
```

```

CLOCK_D, CLOCK_B

WHERE FACT.METADATA_ID          = META.MASTER_ID
    AND FACT.PROCEDURE_GROUP_ID = PROCEDURE_B.GROUP_ID
    AND PROCEDURE_B.MASTER_ID   = PROCEDURE_D.MASTER_ID
    AND PROCEDURE_D.STATUS_INDICATOR = 'A'
    AND FACT.CALENDAR_GROUP_ID = CALENDAR_B.GROUP_ID
    AND CALENDAR_B.MASTER_ID   = CALENDAR_D.MASTER_ID
    AND CALENDAR_B.ROLE        = 'Event'
    AND CALENDAR_B.RANK        = 1
    AND CALENDAR_D.STATUS_INDICATOR = 'A'
    AND FACT.CLOCK_GROUP_ID   = CLOCK_B.GROUP_ID
    AND CLOCK_B.DIMENSION_ID  = CLOCK_D.DIMENSION_ID
    AND CLOCK_B.ROLE          = 'Event'
    AND CLOCK_B.RANK          = 1
    AND CLOCK_D.STATUS_INDICATOR = 'A'
    AND FACT.UOM_GROUP_ID     = UOM_B.GROUP_ID
    AND UOM_B.MASTER_ID       = UOM_D.MASTER_ID
    AND UOM_B.ROLE            = 'Value'
    AND UOM_B.RANK             = 1
    AND UOM_D.STATUS_INDICATOR = 'A'
    AND PROCEDURE_D.MASTER_ID IN (
        << procedure ranking query here >>
    )

ORDER BY PROCEDURE_D.DESCRIPTION, PROCEDURE_D.MASTER_ID,
         CALENDAR_D.SORT_DATE, CLOCK_D.SORT_TIME,
         META.SUBJECT, META.EVENT, META ENTITY, META.ATTRIBUTE;

```

Note that the timeline query includes both the Master ID and the description of the procedure selected. The use of both precludes two different procedures that happen to have the same description, from being merged in the result set. The results obtained in this query depend upon the logic of the ranking query used and the variety of procedural facts loaded into the warehouse, as well as the metadata used to load those facts.

The hierarchy is very important in ranking queries. The warehouse dimensions are full of hierarchies that many users are unfamiliar with or might not even know it exist. These users have result expectations that are typically best addressed by querying ranking queries through the respective hierarchies. For example, lab tests and reference ranges are typically different entries in the procedure dimension connected in a hierarchy, so all reference ranges used for a given lab test are hierachic children of the actual test. The lab results in the Fact table are typically dimensionalized to the Lab Test Reference Range entry. When users query to rank these procedures, they are thinking of the highest-ranking lab test, not the ranking of each reference range variant of that test. For tests with only a single reference range, there's no difference, but for tests that vary their reference ranges across time,

equipment, or facilities, the difference can be dramatic. This means that the most used lab test could be hidden in the ranking because it is defined using many possible reference ranges.

In addition to hierarchic differences, there are also differences in the operational details behind each fact. A typical lab test result has a procedure under which it was ordered and a procedure under which results were recorded. The order is typically for a panel of individual tests. The hierarchic relationship between the panel and its associated tests might be part of the dimension hierarchy or it might not. Whether the ranking is done on the basis of orders or results could make a significant difference in the query results. This is why the nonpatient timeline can be a valuable query and training tool. Because a variety of data is being loaded into the data warehouse, few users automatically or intuitively grasp what all of those data are or how they are related. Creating timeline queries for areas of interest helps provide an increased awareness of the data and can identify opportunities for further inquiry that some users might not otherwise identify.

Much of the power of the timeline is dependent upon the ranking query that drives it. (Note that in theory you could do a timeline without a ranking subquery, but its descriptive power is lost in the sheer size of the result set.) The more discriminating the ranking subquery, the more informative the resulting timeline can be. For example, when you rank procedures, it's probably a good idea to fully use the hierarchy and also include the bridge rolls in the rankings. In this way, your results will include the procedures most ordered and those most resulted. Remember that, by definition, aggregate ancestors always rank higher than their component descendants so it doesn't take any special logic to filter out the lowest ranking entries.

Producing the basic timeline is only the start of a user's *ad hoc* analysis. The base timeline shows all of the different kinds of facts that are recorded for each procedure (identified by the metadata values), the fact values themselves (unless redacted under privacy protection rules), and the units of measure within which those facts are recorded. If a particular type of fact loads in multiple units of measure, the timeline makes that apparent quite readily.

Beyond the simple base timeline, a user can continue to add dimensionality to the results to gain a better understanding of the facts. Adding a diagnosis description to each fact will help to establish context for the ordered procedures. Adding the clinical specialty of the attending provider who ordered a procedure can illuminate a pattern. Knowing the facility (i.e., unit, room, bed) in which something happened can make facts more meaningful. Each of these additional dimension displays is at the discretion of the user and dependent upon the dimensionality that exists on the facts being displayed. Timeline users must become accustomed to seeing the Not Applicable entry under a given dimensional column for many of the facts, since dimensions added to a timeline query will rarely be involved in every fact displayed.

The variants that can be built on these timelines are limited only by the imagination and creativity of the user. Any dimension can be used to anchor a timeline as long as facts have been dimensionalized to make the dimension meaningful within the context in which it is being queried.

Timeline queries of the Material dimension can be used to learn about the range of facts associated with the most-used medications. The timeline will show order and administration events and, by adding appropriate dimensionality, will show where these medications were used (facility), whether or not they were given and why (quality), and what conditions they were being used to treat (pathology). Adding allergens to the timeline, you can observe the timeliness of allergy assessments relative to medication ordering. This last piece requires something about the patient to be pulled back into the timeline, since allergy data are about the patient rather than the medication, but this inclusion is as just one supporting dimension, rather than as the ordering value for the entire timeline. In these cases, the Master ID of the Subject dimension serves as a useful proxy to minimize the amount of patient data that is accessed.

Statistical Analyses

As early queries make users more and more familiar with the data, they can start to think in terms of increasingly sophisticated queries and analyses. The next level of generic queries that I typically work through with early users involves the notion of calculating basic descriptive and applied statistics for any quantitative value stored in any of the Fact tables. Calculating these values, and making them available, can greatly simplify a host of query applications that would otherwise have to start with defining these statistics.

I should preface this discussion by admitting that, while I am an informatics specialist, I am not a biostatistician. I know enough about basic statistics to build the queries that many of my user analysts have looked for over the years, but not enough to fully explain the underlying statistical principles on which many analyses are based. Ensure your project team has some of these skills before you get too deeply into the materials covered in this section. What I discuss in the following can be viewed from a simple “query of the warehouse” standpoint or from a more advanced statistical analysis standpoint. I build the former comfortably, but rely on people who know far more than I do for the latter.

Descriptive Statistics

Descriptive statistics requires a sample of data. We already have several basic queries in our toolset for building data samples: ranked lists, correlations, and timelines. Entries in a ranked list or correlation can be used to differentiate populations of data for statistical analysis, and entries in a resulting timeline

form the basis for longitudinal sampling of data. If we want an hourly median lab result across the most common procedures that we can segment against a research and control cohort of patients, the ranked list and timeline queries are the building blocks for doing so.

In a typical case, the definition of a sample is based on some combination of warehouse dimensional data, such as lab results (metadata) for blood glucose (procedure) measured in mg/dL (UOM). While holding those three dimensions constant, we might see that results are dimensionalized against a patient (subject) and date (calendar). We can break the data into two cohorts: one including all patients with any diagnosis of diabetes in their clinical picture and the other including patients who do not have diabetes anywhere in their clinical picture. We can then conduct a metrics aggregation across all of these data to create a descriptive Metric Fact entry for each permutation of desired data samples, which is a summary for each patient or cohort for each day, month, quarter, and year. This analysis can include two-sample variance comparisons (F-tests) across the two cohorts to determine whether or not the diabetic cohort is different from the nondiabetic cohort during each temporal period in any statistically significant way. If so, further investigation can be conducted using the identified cohorts. If not, then the choice of cohorts can be reevaluated, as needed.

Statistical Process Control

For quantitative facts, whether sourced from outside the warehouse or generated internally as descriptive statistics, time series data are used to identify statistical anomalies and trends using statistical process control (SPC) in a relatively generic fashion that can create additional new facts that can then be made available in queries.

The type of control chart created for any particular sample of data will vary based on the qualities and volume of data available. The blood glucose example earlier consisted of a continuous random variable (measured in mg/dL) that would be controlled using variable control charts. The specific chart selected needs to be determined at query time by looking at the sample sizes associated with the statistics being included in the charts (e.g., small subgroup samples generating X-bar and R charts and larger subgroup samples generating X-bar and S charts).

If we were to plot each lab result in a control chart, we'd be plotting a control chart for each patient. Because the sample size for each data point would be one (i.e., a single result), we'd produce a set of individual and moving range charts. Since the control limits calculated for a sample size of one are very large (e.g., smaller samples yield larger estimated sigma), outliers identified under this approach are out-of-control conditions that need to be evaluated carefully. While the out-of-control condition might indicate actual spikes, troughs, or trend in blood glucose levels, indicating a clinical concern, it might also indicate that

one or more data values were erroneously out of range, indicating a data quality problem. Statistical control is still valid in these cases. An out-of-control condition in a control chart indicates that something nonsystemic has happened in the data. In some cases, the assignable cause is bad data that can be investigated by the warehouse support team. Warehouse users are seeking assignable causes of a clinical nature, so they'll have to settle for finding them among the cacophony of outliers that might be identified in the messiness of your healthcare data. As the out-of-control conditions raised by data quality problems subside, the use of statistical control to identify clinical issues improves. If the source data are clean, the clinical value of the analysis is immediate.

Another approach to analyzing these data is to continue with patient-specific control charts, but build them using aggregated statistics, typically at the hourly or daily basis depending upon how much data are available. For each set of statistics that was derived from a sample size greater than 1 (and less than 30), we'd use the mean value for blood glucose as a subgroup data point in a set of X-bar and R charts. If any of these charts identifies the reading as out of control, it would indicate that an assignable cause is associated with that patient's blood glucose, most likely an increasing or declining trend, but sometimes a set of outlier readings.

Note that SPC makes no judgments about the clinical meaning of any out-of-control conditions. An out-of-control declining trend could represent either a patient with high glucose being improved or a patient with normal glucose becoming hypoglycemic. The out-of-control condition signals only that something is happening in the data that might be worth investigating, nothing more. The assignable cause might turn out to be low-quality data that need to be discarded before repeating the analysis.

As sample sizes grow, the range of a subgroup becomes far less reliable as a measure of the variability of the data in the subgroup. For subgroups larger than 30, we shift from X-bar and R charts to X-bar and S charts, using an estimator of the subgroup sigma to determine more accurate control limits. These charts are likely to be more effective for determining control at the cohort level, where the sample size, even at the daily level, can be expected to exceed 30. Controlling larger time periods becomes problematic unless longitudinal data are available. Control charts at the calendar quarter level can be useful in identifying long-term shifts, but since establishing statistical control generally requires at least 24 subgroups in the dataset, we'll only be able to produce these charts in situations where we have at least 6 years of data. Annual control charts would require at least 24 years of longitudinal data.

Semantic Annotation

The techniques you've seen so far in this chapter have been about discovering value within the data you've loaded into the warehouse. These queries are limited by the actual data that have been loaded. More importantly, the data

loaded have limitations regarding the ways that different data are connected to each other. There are likely to be many possible *semantic* connections among the elements of data that are unknown to the warehouse because they were unknown to the source applications systems from which the data were sourced.

We want to add data to the warehouse to support and promote better connections among the data we've already loaded. We'll start by identifying and loading a variety of semantic ontologies into the warehouse dimensions. This ontological data will enrich our dimensions and allow our queries to find more meaningful connections among the data. The ontological data will also further enrich our stored facts by annotating them with semantic tags that will make them more powerful when used in queries and searches.

Let's take a look at an example. Semantic ontologies like the Human Disease Ontology or the Foundational Model of Anatomy can provide a lot of added value to the warehouse by introducing pathways and views beyond the dimensionality and reference data provided by the original data sources. By loading ontologies such as these into the warehouse, the integration provided by these semantic datasets becomes available for querying our Fact tables, even though the concepts defined in the ontologies were unavailable, or even unknown, at the time the facts were loaded.

To illustrate, let's assume that diagnostic data for patients have been loaded into the warehouse using fairly standard ICD-9 codes as reference data. A patient diagnosis of pneumonia is dimensionalized to the 482.9 diagnosis code in the diagnosis dimension. The current description of that diagnosis code in the dimension is "bacterial pneumonia, unspecified" (Figure 19.2). Users who don't necessarily know the coding of diagnoses can find these facts by querying some form of text-based WHERE clause that focuses on that description, or descriptions similar to it. This *ad hoc* text-based approach is generally unsatisfying, because it is more difficult, and it tends to miss important entries that would be expected by users to be part of the results, but that don't contain the necessary textual descriptions to match the specified WHERE clauses.

Including semantic ontologies in our warehouse gives us a better alternative to the historical text-based WHERE and LIKE clauses. The ontologies embed certain semantic knowledge regarding the relationships among concepts that

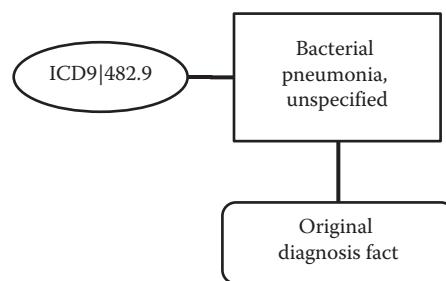


Figure 19.2 Diagnosis fact against single ICD-9 code.

will allow us to query data without the need to create and refine *ad hoc* textual searches. The Human Disease Ontology includes the following entry for bacterial pneumonia:

```

id: DOID:874
name: bacterial pneumonia
alt_id: DOID:13815
def: "A pneumonia involving inflammation of the lungs caused by bacteria."
synonym: "gram-negative pneumonia" EXACT []
synonym: "Pneumonia due to other gram-negative bacteria" EXACT [ICD9CM_2006:482.83]
xref: ICD9CM:482.9
is_a: DOID:552 ! pneumonia

```

There's nothing unique about the loading of ontologies into the warehouse. Ontologies will load concepts into the dimension Definition tables and relationships into dimension Hierarchy tables. To load this ontology into your warehouse, you have to analyze it like any other data source and build a source data intake job to source the data into your ETL process. The aforementioned example sources new data into the Diagnosis dimension, including

- A new construct for “bacterial pneumonia” using the identifier “DOID|874” in the Reference Composite
- An alias Reference entry with “DOID|13815” in the Reference Composite
- A new construct for “gram-negative pneumonia” using the identifier “DOID|874|gram-negative pneumonia” in the Reference Composite (which will be connected as a natural hierarchy child to the main construct)
- Another new construct for “pneumonia due to other gram-negative bacteria” using the identifier “DOID|874|pneumonia due to other gram-negative bacteria” (also a natural hierarchy child of the main construct)
- An *exactMatch* mapping connection between the “pneumonia due to other gram-negative bacteria” construct and the dimensional entry for the ICD-9 482.83 (pneumonia due to other gram-negative bacteria) diagnosis construct
- A *closeMatch* mapping connection between the main “bacterial pneumonia” construct and the dimensional entry for the ICD9 482.9 (bacterial pneumonia, unspecified) diagnosis construct
- An *isA* parent relationship from the main “bacterial pneumonia” construct and the higher-level “pneumonia” construct with a “DOID|552” Reference Composite

Note that the connection between the ICD-9 code and the ontology construct has nothing to do with any textual similarities between the two constructs. (This isn't about text-matching entries in the dimension, something very tedious that we've been doing for years, but will no longer need to do because of the availability of some of these semantic tools.) These constructs are being aligned in the warehouse because the ontology lists an explicit cross-reference between the ontology entry and the 482.9 instance in the ICD-9 codeset.

The ontology also contains cross-references back to codes in MeSH, NCI, UMLS, and SNOMED that are not shown in the example. Those entries would typically only be loaded if the related ontologies were also being loaded, which can be accomplished contemporaneously or at a later date, as they are needed. As a result of loading just this ICD-9 subset of the ontology data, multiple additional pathways become available for querying diagnosis data, even though none of that sourced data were originally dimensionalized against any of the new ontology-based concepts (Figure 19.3).

With semantic relationships loaded, user queries can now find facts more easily, without having to know the particular coding schemes that were originally used in the source application systems to define the sourced facts. For example, qualifying a search on pneumonia from the Human Disease Ontology will return facts about pneumonia that were originally coded in any scheme that has been mapped in the dimension, including ICD-9, ICD-10, SNOMED, or any other coded source. Since the warehouse consolidates data from many clinical applications, each of which might use a different coding scheme, this semantic annotation capability is critical to the maturing of your data warehouse.

The major ontologies you'll likely load first are public datasets defined and maintained by the professional community around the world. None of the datasets is perfect. Any source dataset is going to have data quality problems, and semantic ontologies are no exception. Generally, I find that I trust the consensus reached across the ontology community in defining and curating these ontologies. While I've seen small inconsistencies introduced into many of them, I've never seen an error that I considered significant enough to warrant not using a particular ontology. I've also found that ontology entries can be validated in the warehouse in ways that would not be possible based only on

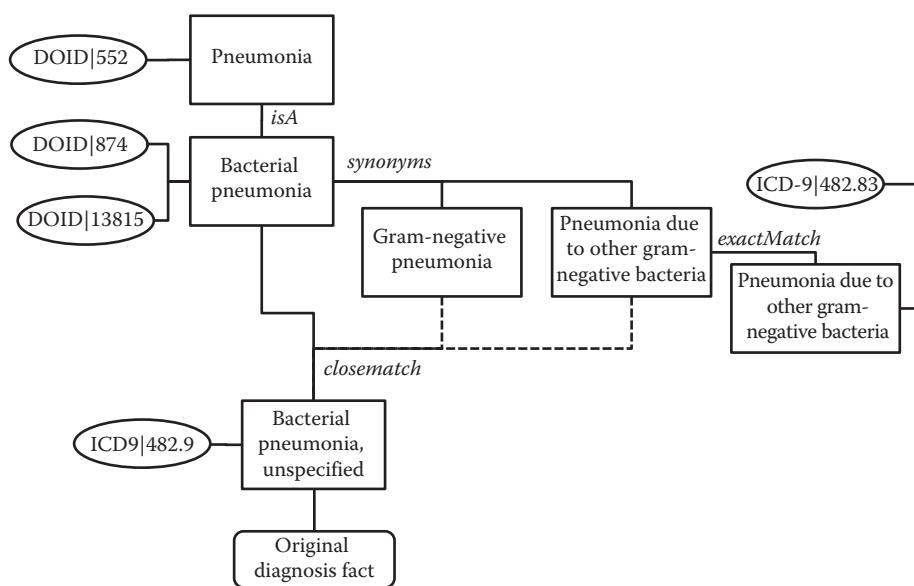


Figure 19.3 Enriched diagnosis fact with human disease semantics.

the individual ontologies. As a result, a semantically annotated warehouse can be used to provide feedback to the ontology communities in order to continuously improve their entries and relationships.

A key tool in defining semantic annotations is the Simple Knowledge Organization System (SKOS). The SKOS is an ontology specifically defined by the community to support the linking of data among multiple vocabularies, taxonomies, and other ontologies. The *mappingRelation* construct in SKOS allows us to classify the form of semantic mapping that is represented by each relationship that we load into a Hierarchy table while loading various ontologies (Figure 19.4).

Each *mappingRelation* entry between two definitions in a warehouse Hierarchy table asserts a relationship between those two definitions: they are *mapped*. The SKOS provides five different levels of semantic association to be defined between those two entries. If you add a Mapping Relation column to the Hierarchy table, you can store the appropriate relation as part of the hierarchy entry and use that value to qualify, or automatically expand, user queries.

The least restrictive of the SKOS mappings is the *relatedMatch* concept. By definition, the entry in your Hierarchy table implies that, at a minimum, the two definitions are related. The SKOS defines this relationship as *symmetric*, so if A-relatedMatch-B, then B-relatedMatch-A. Many ontologies don't include inverse relations among concepts, so they can be generated by your source jobs, if desired. Generating relations can be problematic because the implied relationships are semantic, and your sourcing job might not be able to adequately parse the verbiage in provided relations to discern what their inverse relation names might be. For example, if an ontology asserts that A-causes-B, the SKOS rule implies that B-isCausedBy-A. Actually, generating the inverse verbiage requires some natural language processing capability that might be unavailable to your warehouse jobs, so another loading strategy is to generate an arbitrary

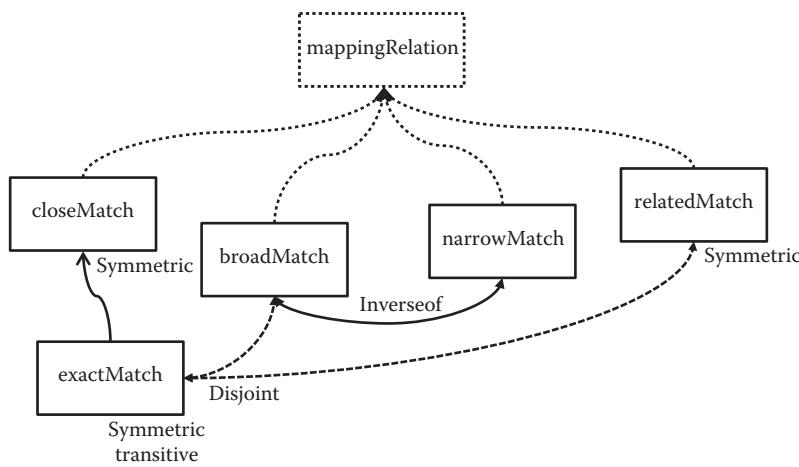


Figure 19.4 Simple Knowledge Organization System *mappingRelation* subtypes and constraints.

inverse relation: *inverseCauses*. The arbitrary generation eliminates the risk of accidentally generating an incorrect relation and is the strategy used in many ontologies throughout the community.

As a loose-fitting relationship mapping, the relatedMatch isn't very useful for expanding or augmenting user queries. Because nothing is being asserted about the semantic relationship content, no automatic semantic processing is enabled; however, a user seeing these relations can choose to include or exclude them from a query.

Automatic processing is enabled by the other four SKOS mapping types. The *broadMatch* and *narrowMatch* mappings are very common and serve as inverses of each other. If a definition is a *broadMatch* to another, the target definition broadly encompasses the source definition (e.g., Florida-*broadMatch*-United States). A *narrowMatch* relation is always defined as the inverse of a broadMatch mapping (e.g., United States-*narrowMatch*-Florida). If one or the other is not present in the ontology being loaded, either can safely be generated at load time.

The *narrowMatch* mappings can be used at query time to expand the range of data included in the results of a query. Since the target is an encompassed subset of the source, data about the target are legitimate data to be returned in a query against the source. For example, when querying all facts for ICD-9 code 482, facts associated with ICD-9 code 482.9 should be returned because ICD-9 482-*narrowMatch*-482.9. A query of the 482.9 code, specifically, would not return data for the broader 482 code because the inverse *broadMatch* mapping doesn't justify such an expansion. While the data for the broader 482 code might include some cases that would warrant inclusion of the narrower 482.9 code, the results would also be expected to include data that aggregates other specific codes. Automatically expanding queries based on broadening relation mappings would expand results incorrectly.

Note that the United States-to-Florida and 482-to-482.9 examples involve definitions in the dimensions that would probably have resulted in natural hierarchies being loaded because of the overlapping structures of the values of their respective Reference Composites (e.g., USA *includes* USA|FL and 382 *includes* 382.9). It is true that all natural hierarchy entries are *narrowMatch* mappings. In all of these cases, the *syntax* of the keys is sufficient to infer the *semantic* relationship. The power of the SKOS mappings is that the relationship between the two definitions can be defined semantically without any regard to the syntax of the associated keys. Natural hierarchies are merely the easiest semantic relationships to recognize. Regrettably, for the vast majority of semantic mappings, you won't have the benefit of any syntactic overlap.

The SKOS provides two more mapping types that are useful for augmenting user queries. A *closeMatch* relationship exists between two definitions if the two definitions are considered functionally equivalent for the semantic purpose of using them in queries. An *exactMatch* relationship exists if the two definitions are considered semantically equivalent. These two mapping types are particularly useful when loading master data into dimensions from

multiple diverse sources (e.g., mapping codes across ICD-9, ICD-10, and SNOMED). Those diverse sources might use completely different coding schemes to represent the same semantic information, so mapping the master values across those source systems as *closeMatch* and *exactMatch* relations allows queries to automatically merge the data without having actually to convert any of the master codings during a query.

Initially, the mappings you load to the warehouse will be sourced from the ontologies you choose to load. Many public ontologies are using these SKOS mapping types to cross-reference concepts within and among different ontologies. With practice, you'll find that you want to load your own curated mapping based on user feedback and experience and coordinated with your warehouse governance function. When you observe users struggling to combine data from multiple sources in their queries, you'll often find that adding a few *narrowMatch*, *closeMatch*, or *exactMatch* relationships among some of the dimension data will solve the problem.

Adding semantic mappings to your Hierarchy tables allows the standard query used to access a set of facts against a dimension to be refined. Counting facts by their connection to particular definitions in a dimension requires a simple JOIN statement:

```
SELECT D.DESCRIPTION, COUNT(*)
  FROM FACT
 INNER JOIN DIAGNOSIS_B B ON (FACT.DIAGNOSIS_GROUP_ID = B.GROUP_ID)
 INNER JOIN DIAGNOSIS_D D ON (B.MASTER_ID = D.MASTER_ID);
```

In Chapter 17, I described this join as unsatisfying because it fails to take advantage of the aggregation capability of the hierarchies within the dimension. By adding a hierarchic element to the standard query, the resulting counts can be made to automatically include those hierarchic roll-ups:

```
SELECT D.DESCRIPTION, COUNT(*)
  FROM FACT
 INNER JOIN DIAGNOSIS_B B ON (FACT.DIAGNOSIS_GROUP_ID = B.GROUP_ID)
 INNER JOIN DIAGNOSIS_H H ON (D.MASTER_ID = H.DESCENDENT_MASTER_ID)
 INNER JOIN DIAGNOSIS_D D ON (H.ANCESTOR_MASTER_ID = D.MASTER_ID);
```

This initial version of the standard query actually produces incorrect results when you load hierarchy entries from ontologies because not all relationships available within those ontologies represent legitimate aggregating relationships. The SKOS mapping type can be used to correct the standard query, so aggregation only occurs across legitimate and warranted relationships:

```
SELECT D.DESCRIPTION, COUNT(*)
  FROM FACT
 INNER JOIN DIAGNOSIS_B B ON (FACT.DIAGNOSIS_GROUP_ID = B.GROUP_ID)
 INNER JOIN DIAGNOSIS_H H ON (D.MASTER_ID = H.DESCENDENT_MASTER_ID
                             AND H.RELATION_MAPPING IS IN ('Exact', 'Close', 'Broad'))
 INNER JOIN DIAGNOSIS_D D ON (H.ANCESTOR_MASTER_ID = D.MASTER_ID);
```

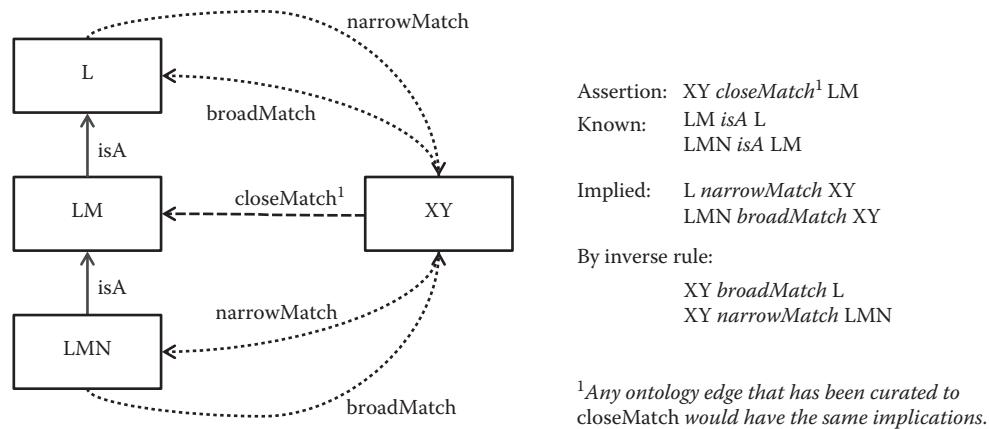


Figure 19.5 Implied mappings based on Simple Knowledge Organization System types.

The query change can be made within your business intelligence layer, so users benefit from it without needing to know it is present or whether these relationships actually exist within your dimensions.

As you incorporate semantic ontologies and their SKOS mappings into your warehouse, you'll be able to use those data to further validate your dimensions and their connections (Figure 19.5). Existing mappings will be used to validate each other and to generate additional implied mappings that you can curate into the data once approved by your governance group.

When errors occur in ontology mappings, I tend to find that the mistakes are more likely to be encountered among the more specific mappings. I believe this is because the teams, who define those more specific relationships fail to recognize sufficient exceptions to what they are asserting. Figure 19.5 illustrates some of the common combinations of relationships found in cross-ontology mappings. If the XY-closeMatch-LM is asserted from one ontology to another, and the target ontology includes the two “isA” relationships among its own concepts, the illustrated narrowMatch and broadMatch mappings are implied. It's not difficult to write queries in your dimension to discern these patterns. If contradictory relationships are generated, you know there are errors in the mappings. You don't know which mappings are incorrect, only that contradictory mappings do exist. For example, suppose that Figure 19.5 also asserted XY-closeMatch-L. Such a relationship would contradict the XY-broadMatch-L relation generated by your query, so the error would be discerned. Logically, you shouldn't have a closeMatch *and* a broadMatch relationship between two definitions because this mismatch can cause a query to produce different results based on exactly how it is written. It could be that both interpretations are “true” under certain circumstances, but there's an element of risk in the apparent contradiction. I provide queries and tools for users to explore these considerations, allowing them to include or exclude data based on their own interpretations of what they are looking for. With respect to any automatic expansion of queries in my business intelligence layer, I always interpret apparent

contradictions conservatively, taking the less restrictive view (e.g., I would treat the combination *broadMatch* and *closeMatch* as a *broadMatch* relationship).

Compared to other work you've done to implement your data warehouse, knowledge synthesis and generation generally, and semantic annotation particularly, are advanced functions that you are unlikely to heavily invest in during your first year of activity. By your third or fourth year, ontology management will become one of your dominant activities as you approach a point where most or all of your available in-house data sources are regularly loaded into the warehouse. The ontology community is large and active, and the scope and complexity of available ontologies changes almost every day. Anything else I could tell you about their implementation will be obsolete long before you get around to implementing them. I urge you to pursue more information in this area and to become an active participant and contributor to that community. Your warehouse support work will likely depend on this fast-changing and growing discipline.

Chapter 20

Data Governance

In order for the new data warehouse to truly enable an organizational shift toward information-based analysis and decision making, the entire organization must embrace the challenges of properly governing and administering data. Data management needs to be an explicit and specific set of functions that must encompass all of the data across the enterprise, not just the data warehouse. The warehouse presents unique opportunities for enabling data governance because it is the one system in the institution that combines data from across literally all other systems. Governance challenges are made visible in the warehouse, yet the solutions to data problems often require going back to the data sources. Properly managed, data governance will drive the organization into an informatics-based warehouse-enabled future.

Organizing for Governance

With a new data warehouse in place, the functions of organizational readiness raised in Chapter 1 now need to be institutionalized in order to maximize the benefits to be achieved through warehouse-based analytics. This entails creating new data governance bodies and assigning new responsibilities, if they don't already exist within the organization. From a high-level data governance group to data owners on the clinical side and data administrators on the technical side, along with the team dedicated to supporting the warehouse, a large number of people and functions in the organization contribute to an effective data governance framework. The roles these people play in using the warehouse were introduced back in Chapter 17. Our interest here is in the organizational aspects of these groups and their interactions.

Data Governance Group

There are typically no central governing bodies spanning large organizations to manage the policies needed for data ownership and definition across the array of applications that are likely to interface with the new data warehousing portfolio. Without some form of governance body, information technology staff will be forced to make choices among data alternatives that should be made within the clinical and research user community. If your organization doesn't have a governance body, I urge you to establish a cross-functional data governance group to set policy and guidance for data definition, ownership, and stewardship of the strategic data assets of the enterprise.

The data governance group should be made up of very senior management from across the enterprise. They will be responsible for defining high-level policies and allocating project resources based on those policies, decisions that can't be effective without the positional and organizational authority necessary to implement and enforce them. Implementing this recommendation will be much more difficult than the implementation of any particular function within information technology because the stakeholders of interest are almost exclusively outside the information technology function. The people I prefer to see in this group are functional and department heads from across the institution, including executive and financial officers, medical and nursing management, and compliance and safety directors. To the extent that information technology is represented within this group, I recommend the CIO be a member in her or his role as a senior manager, and not as a technical representative.

This group's mission to define data policy often involves arguing among themselves on key issues of conflict. The advantage is that it prevents information technology people from having to decide issues that might stay in conflict if not agreed to by the user community. The goals of the data governance group should include the following:

- Enhancing the quality of data throughout the enterprise environment
- Identifying and promoting innovative uses for data and information
- Providing advice and information to data owners, stewards, and users
- Ensuring timely and appropriate addressing of data management issues
- Promoting the voice of the customer in data administration products and services

The objectives for meeting these goals should include the following:

- Developing data management policy recommendations and procedures
- Defining roles and responsibilities for data across the enterprise
- Identifying enterprise, regulatory, and statutory data elements and controls
- Promoting creation and use of standards and principles to support data integration

- Establishing guidelines for data modification, migration, and archiving
- Promoting acquisition and sharing of data to minimize cost and maximize reusability
- Developing a vision and recommendations for anticipated future data requirements
- Developing and sharing consistent technical solutions for priority issues
- Conducting data quality assurance analyses and audits
- Responding to additional data management issues as they arise

The data governance group will play a key role in prioritizing the data sources available to the data warehouse, working to resolve semantic conflicts among those data sources, and (particularly important in the beginning) reconciling the host of reference data codes that will be in conflict during the initial loading of most of the dimensions in the new warehouse. Having some form of data governance function in place will be a critical success factor in implementing the new data warehouse portfolio. Without effective governance, the new warehouse will become a large repository of data that can't be effectively integrated into useful information that can be shared and reused across the enterprise.

Data Owners

Owing to its senior management membership, the data governance group will not have the knowledge and experience with the institution's application systems and data needed to actually research, discuss, and decide data issues that arise in the environment. I recommend creating a pool of data owners from across the enterprise to serve as subject matter experts regarding data and applications. This pool of people can be quite large depending on the scope and scale of the organization and the complexity of its information portfolio.

I recommend identifying a data owner for each dimension or subdimension of the warehouse and one for each data source that is, or will be, loaded. This allocation creates a form of matrix data group, where most issues that arise will require input from one or more source owners coupled with one or more dimensional owners. I also recommend adding functional specialists to this group for any shared or common capabilities, such as HIPAA or IRB. Together, these teams will be able to address and resolve issues that are raised by the data warehouse support team. The responsibilities of the data owners should include the following:

- Providing definitions of data elements and terms within their domain
- Providing recommendations to data governance on proposed actions
- Assisting with the dimensionalization of facts from related sources
- Establishing control settings for information privacy and quality
- Recommending solutions to data quality or data mapping inconsistencies
- Advising the warehouse support team on day-to-day issues and concerns

Ownership of a data source or subdimension cannot always be established in a single individual. In many cases, a small team might be needed to provide proper coverage for a domain. These teams will typically perform their role through some form of consensus process and disseminate alternatives and suggestions to other governance functions accordingly. A bigger problem exists when there are no individuals in the organization who are able to serve the data ownership role for subsets of data. This occurs most often with externally sourced data, particularly as semantic ontologies are sourced during future release efforts, the ontologies having no internal stakeholders or advocates who will serve as data owners. For the data to be viable, ownership needs to be established for every new data type or source. When no ownership can be established, I recommend against adding the data to the warehouse.

Data Administration

Most organizations need to look more carefully at how they establish, manage, and monitor data standards and issues across all of their information technology portfolios. A new warehousing environment will increase this need, and in order for data administration to get the attention and focus it requires, a more formal charter and structure is typically needed. If your organization does not have some form of centralized data administration function within your overall information technology organization, I recommend that you charter and establish one as part of your data warehouse initiative.

The data administration function can help define, and advocate for, standards and practices related to data and metadata management across the enterprise, not just in your data warehouse. Creating a data administration function is initially about standardizing practices and conventions that are at least partially in place, if only informally. The participants should be almost exclusively drawn from the staff of information technology, and their emphasis should be on setting data standards and controls necessary to enable the policies to be set by the (possibly newly established) data governance group and implemented as easily as possible. The basic responsibilities of a data administration function might include the following:

- Acting as a communication conduit among user constituencies and the data governance and owner groups on data management issues
- Representing constituent groups in the broader organizational community on data-related matters
- Promoting best practices for the management and use of enterprise data
- Communicating with constituent projects and reporting progress and implications of new or updated data policies and procedures
- Acting as project advocates and change agents

- Expediting and facilitating data management issues resolution
- Chairing and/or participating on working subgroups as appropriate
- Participating in staff meetings or as designees to project reviews

There is a wealth of information available in the academic and industry literature to assist with implementing a data administration function. The Data Administration Management Association (DAMA) is a good place to start.

Warehouse Support Team

The information technology team that implements the warehouse isn't always the same team that supports the warehouse in operation, although an overlap of staffing is expected and normal. Day-to-day operational support of the warehouse is separate from any continuing development of new releases of the warehouse. Since development of upgrades and new data sources is expected to be continuous, there will always be a level of parallel activity between development and support. It is the support role that I am considering here. The basic areas of concern for the warehouse support team should include the following:

- Functional, such as an inability to define queries over the range of desired results
- Quality, such as data errors, mismatches, or omissions
- Integration, such as an inability to naturally integrate additional data across sources
- Control, such as a lack of confidence in HIPAA or IRB integration
- Scaling, such as an inability to continue adding new data or sources
- Performance, such as excessive ETL job, or query, run times

Warehouse support team members are among the most active users of the warehouse. They use the same tools and operate under the same privacy restrictions, as any other users. Much of their work could be described as routine monitoring of the various controls that have been built into the warehouse and initiating actions at the data administration, data owner, or data governance levels based on trends and concerns in the data. Their work also involves user enablement, including ongoing training of users as well as providing assistance to users when problems are encountered.

Collectively, the data governance group, data administration, data owners, and data warehouse support team constitute the data governance capability for the enterprise, with responsibilities that go beyond the data warehouse to include all of the enterprise data within and across the institution. Different elements of this organizational structure will be active in pursuing opportunities that present themselves, with ultimate policy responsibility residing in the data governance group of senior management.

Governance Opportunities

Your data governance function enables all of the various data governance challenges that have arisen in all of the previous chapters to be consolidated and addressed as opportunities. Some will involve corrective actions against data problems, while others will involve enabling extended capabilities in the new warehouse.

Immediate Decisions

Among the issues that the data governance function must be concerned with, many involve decisions that need to be made immediately, some in order to facilitate the actual implementation of the warehouse. The need for immediate resolution of certain issues highlights the importance of establishing the data governance function in parallel with the development of the warehouse. Actions and decisions that arise during development that must be addressed by data governance include the following:

- Setting policy with respect to variations and revisions that will be permitted between data as sourced versus how it appears when loaded in the warehouse (Chapter 13)
- Defining the required level of data visibility that should be designed into the ETL Control pipe (Chapter 9)
- Ensuring that broken keys that require resolution to the Unknown system row in any dimension are diagnosed and corrected quickly to prevent degradation of the data associated with any erroneous source data (Chapter 10)
- Investigating arriving data that violate the modeled expectations for the data in question (e.g., NKA w/ Hives, allergies to “Apples and Bananas”), so metadata can be corrected or enhanced to resolve problems (Chapter 8)
- Adopting standards and guidelines for the setting and handling of Redacted, Unexpected, and Undesired flags in the dimensions (Chapters 10 and 15)
- Monitoring and resolving dimension orphans that occur during source loads (Chapter 15)
- Establishing the maximum length of textual data that can be displayed while still presuming deidentification of patients and beyond which query results must be redacted (Chapter 17)

Near-Term Control

Beyond immediate decisions, there will be many issues that the data governance function needs to address on a day-to-day or month-to-month basis. It is likely that many of these issues are already present in the initial implementation. It is at this level that the interactions among the various data governance groups

and layers need to be most integrated to assure that issues that arise in the early warehouse get resolved both quickly and consistently. The types of decisions and actions in this category include the following:

- Ensuring that all terms, concepts, variables, and metrics in the warehouse environment are correctly and consistently named and manipulated (Chapter 16)
- Defining expected units of measure aggressively (e.g., “350 mg” as “text” vs. “350” as “mg”), so sourced data are most likely to be defined within functionally specific UOMs rather than broader narrative forms (Chapter 14)
- Defining the balance between Alpha and Beta risk in the processing and resolution of entries, where *Undetermined* dimension entries might resolve to *Ambiguous* (Chapter 14)
- Continuously researching, and responding to, data quality hypotheses derived from the warehouse (Chapter 15)

The resolution of near-term issues and concerns is a key success factor in completing the initial implementation of the data warehouse within 1 year. Questions or issues that are allowed to linger can delay implementation or, worse, result in an incorrect implementation. Getting your data governance group functioning as early as possible in the development life cycle will avoid having the implementation team make many of these decisions in an IT-centric way.

Medium-Term Growth

The data governance function must work to enable medium-term growth of the warehouse. As opportunities present themselves, the governance function must prioritize and allocate resources to the continuing development of warehouse capabilities. Most of these issues can be addressed on a quarter-to-quarter basis in the first few years and a year-to-year basis beyond that point. The types of decisions and actions in this category include the following:

- Ensuring that corrections applied to the data in the warehouse (e.g., annotation of medication administrations that were subsequently reversed) represent well-defined improvements for which organizational consensus has been achieved and that those changes are made in a well-controlled manner (Chapter 8)
- Annotating quantitative measures or facts using information derived from qualitative or narrative facts (e.g., Lab Comments used to updimensionalize Lab Results) based on natural language processing (NLP) analysis or manual curation (Chapter 8)
- Reviewing and approving requests for exported data marts from the warehouse (e.g., to i2b2 datasets), including whether those exports may be allowed to contain identified data (Chapter 17)

- Reviewing and approving new query-layer views of data in the warehouse (e.g., flattened groups, pivoted facts) and ensuring that correct and logical consistency is maintained by any filtering or manipulation of the data (Chapter 17)
- Ensuring that proposed new organization-specific Data State dimension entries are implemented in a consistent and controlled manner (Chapter 7)

Other medium-term areas of concern tend to move beyond the initial implementation of the data warehouse to issues of organizational and environmental integration and expansion, such as management of releases, providing feedback to source applications, and expanding the sharing of data.

Release Management

The new data warehouse portfolio is a long-term commitment of resources that needs to be carefully planned, exploring alternative scenarios that can vary the order in which new data or functionality is implemented. The architecture of a dimensional warehouse requires a significant initial effort to get the infrastructure up and running, but then supports an iterative cycling through new data sources on a regular basis. If such a plan for those iterations is in place early, the pressure to maximize the data loaded into the first production-level release is lessened because of assurances that subsequent data sources will be added routinely.

I recommend that you formalize a planning process for the data warehouse support team, in conjunction with the new data administration and governance bodies, to identify and prioritize data sources to be loaded into the new data warehouse portfolio, emphasizing new release upgrades at least quarterly for the first 3–5 years. Having a plan in place to maximize the data loaded will lessen the inevitable pressure to load a few more data sources into the new warehouse as the implementation of the first release draws near. Pushing for those few extra initial sources can cause the entire warehouse effort to slip into disarray and potentially miss those 1-year target goals. Knowing that a next release will pick up missed lower-priority sources can alleviate the concerns of the data owners associated with those sources.

Without a reasonable and believable plan that shows how data sources will be added to the warehouse over time, there will be tremendous pressure on the data warehouse team to maximize the data included in the production warehouse. The governance team takes political pressure off the data warehouse team by prioritizing the sources. The most favorable and workable scenario is to plan an updated release of the warehouse every 3–4 months, with each upgrade adding several data sources. In this manner, most major data sources will be loaded and available within only 2–3 years.

Source Application Feedback

One of the most important, yet unanticipated, contributions that the warehouse will make to your organization involves providing feedback about your use of

various application systems within the organization. The feedback will include data quality issues and usage patterns that can be improved or enhanced back in the sources from which your warehoused data are drawn. You'll be able to see patterns and errors in the warehouse data better than anywhere else in your environment, because so many of those problems only become evident when more rigorous unit-of-measure controls are added to the data, or when situations arise that are only recognizable through the integration of data from multiple sources.

Most of these issues will trace back to three areas: (a) flaws in the underlying software applications that collected the data, (b) errors or inconsistencies in the master data that have been configured in those applications, and (c) procedural issues regarding how those applications are used by the institution.

Of these three, the first is the hardest to fix because most healthcare applications today are purchased or licensed products, the vendors of which are working to satisfy a large market. These kinds of flaws typically don't cause their software not to work, so the vendors have few incentives to correct them. They'll work on the big ones, but the cycle time on corrections in new releases is sometimes years. There might be work-around procedures that can be implemented in the meantime, but, for the most part, the governance actions will usually have to presume that the underlying flaws won't be corrected in the short term.

Procedural or master data inconsistencies can usually be corrected within the institution, although this will only happen if your governance group is willing to own the problems and use their positional authority to drive or demand the necessary changes. By definition, these issues are beyond the direct scope of the warehouse support team. The main work of the warehouse support team is to apply changes and revisions to the warehouse based on whatever corrective action is taken by the governance function.

It's important to wait until corrective action has been taken to begin adjusting or correcting the data in the warehouse. You want to be sure that those adjustments result in the historical data looking the same as any new data as a result of the change. You don't want governance actions to create temporal seams that would introduce logical breaks in data queries. Changing the source through systems, master data, or procedural change should be viewed as an enhancement to the data sources that are received for that data, and the process of redesigning the metadata for those revised sources should be carried out as rigorously as for any new or revised source.

Expanded Data Sharing

The healthcare sector is seeing dramatic expansion in the types, ranges, and quantities of data that are shared within and across institutions, and all of these data will eventually make its way into your warehouse. Some of this is driven by technology development, particularly in the areas of information exchanges and data federation, and is being enabled by an expanded range of data sharing

messages that promote continuity of care capabilities across the healthcare system. The motivation for much of this sharing comes from policy initiatives and programs such as Meaningful Use and Accountable Care. Within institutions, increased and expanded sharing can be the result of merger and acquisition activity within the healthcare sector.

A result of this expansive movement and consolidation of data is that data brought into the warehouse need to be analyzed very concretely in terms of how they fit the entire healthcare sector rather than just the institution. The governance group should work to ensure that all data brought into the warehouse are interpreted as part of an expansive system-wide view. Overly specific data and metadata, particularly in the earlier implementation releases, can create problems later when more data of similar logical types are brought into the warehouse. If earlier definitions were too broad, the new data can conflict with old data. This problem can always be corrected by adjusting historical metadata, but it can be avoided by aggressively focusing early releases on sector-wide analysis.

Long-Term Evolution

Our warehouses will evolve in ways that are difficult to imagine today, both in technical capacity and functional capability, but much will stay the same. The data warehouses that I built 35 years ago consisted of fact tables surrounded by master dimensions, required bridges and groups to allow individual facts to reference multiple entries in each dimension, provided aggregation of facts across multiple hierarchies stored in the dimensions, and required ways to annotate or correct facts that embedded data quality issues that couldn't necessarily be corrected in the source applications. A large fact table had millions instead of billions of rows, and we didn't have semantic ontologies to provide a rich set of connections among the data, but we certainly had *requirements* for these things.

Many of the changes that have taken place have been within the technical layers of the warehouse. Database management systems can perform functions and provide capabilities that were either impossible in those early years or that had to be provided through application programming. Without those ongoing technical improvements, we wouldn't be able to store and index the scale of data we routinely use today nor could we expect query response times measured in seconds. Those are precursors to a successful warehouse, but ultimately they are in the background.

The big changes today are in the semantics of understanding the meaning of, and relationships among, the data we warehouse and of extracting that meaning from unstructured and semistructured data. It is these emerging semantic capabilities that will prevent our “Big Data” warehouses from becoming landfills. The governance function must continually facilitate the curation of data throughout the warehouse, particularly the use of SKOS relation types, to provide semantic integration across otherwise orthogonal or isolated data (Chapter 19). Evolution is more punctuated than continuous, and right now the

people who are defining and building the semantic ontologies in biomedicine are ahead of our ability to use them in data warehouses. As a result, I expect a leap to take place in warehouse semantics in the next few years, and the enabled capabilities will be very exciting. Other long-term issues of interest for governance include increasing complexity and diversity of data as well as process and staff capabilities for handling these evolutionary factors. The healthcare trends toward personalized and population medicine will require entire new categories of genomic and proteomic data to be loaded into our warehouses, and that increased scale and complexity of data will require improved processes and support skills.

Process Maturity

The organizational process maturity of the information technology groups at most healthcare sector organizations typically rests upon some very *ad hoc* process management, where the skills of the individuals doing various jobs are significant, but the documentation and processes for doing those jobs are largely in the heads of the people doing them. As your data warehouse increases the interaction and complexity of your entire biomedical information portfolio, it will put a strain on individuals as the scope of complexity of their jobs increases. A lot of work on information technology process maturity has gone on in multiple industries over the past two decades, with little penetration of those models into the biomedical sector. As your organization's IT environment becomes more complex with large-scale warehousing, you can reduce its long-term risk by beginning to adopt some of these models.

Depending on your own self-assessed level of process maturity in your organization, I recommend that you begin the process of identifying, developing, and institutionalizing more mature processes and practices throughout the information technology function of the enterprise. As the basis of your process management improvement effort, I recommend the SEI Capability Maturity Model Integration (CMMI) for engineering, acquisition, or development and ISO 20000 for service and operations. This book deals almost exclusively with the development and implementation of your data warehouse. The operation and service models for your warehouse that can be enabled by adopting these two models also need to be in place as well before you complete your implementation.

Translational Medicine

In physics, the conflict between general relativity and quantum mechanics is clear to see: General relativity *requires* a singularity to exist, and quantum mechanics *forbids* one. The conflict is irreconcilable, and experts involved in the fields know it. Biomedical data warehousing has similarly irreconcilable problems, but they aren't as easy to see. The methods we use to store and use

tens of thousands of facts for a patient simply won't extend to storing and using the *millions* or *billions* of facts implied by genomics, proteomics, and the other *-omic* fields. Resolving this conflict will require the continuing evolution of the capabilities of data warehousing technologies along with a commitment to the expansion of standards and semantics throughout the biomedical and healthcare continuum.

Data warehousing has changed during the time it took me to write this book and has probably even changed during the time it has taken you to read this book. The more important changes will be those that address some of the fundamental or constraining challenges we have been discussing. While we can't predict the future absolutely, the terrain within which we're moving is fairly well understood, enabling us to plan for that future today. I'll continue my implementation projects and semantic research, and I'll work to pass along what I learn.

Index

A

Admission–discharge–transfer (ADT) facts, 497–498
Affinity analysis, dimensional data modeling
ad hoc, 34
clusters, 33–34
collection, heuristics, 33–34
concepts, 33
logical groupings, 33
project-relevant data, research, 33
reverse engineering, 37–38
simple affinity principle, 33
supertype entities
characteristics, 35
cohorts and people, 37
combination, 37
database design process, 36
data normalization, 35–36
entities, 34–35
geopolitics, 36
information technology, 37
patient dimension, 36–37
Aggregated metrics
algorithmic calculation, 504
binomial fact, 506
conduct comparable hierachic, 504
creation, nursing unit level, 502
N-dimensional, 504
nonbinomial fact, 506
process of querying, 502
quantitative values, 507
single measure
calendar, 503–504
facility, 502–503
facility-calendar, 505–506
single measures, 502–503
Alpha version
coded, 149–150
completion, 188

groups, bridges and hierarchies process, 162–164
implementation, 149
lab outcomes, dimensions and facts
Alpha coding, 179
bridge groups, 179–180
brute-force fact load, multi-bridge Group, 186–187
complexity, 176
differences, Reference Range columns, 183–184
Group table, 181
multiple-bridge groups, 182–183
natural key, 176
nonNULL Reference Range data, 180–181
NULL Reference Range data, 180
Procedure dimension, 176, 179
Reference Range entries, 177–179
Reference table entries, 177
Result Comment facts, 184–186
source table, 175–176
SQL code, 176–177
surrogate generation entries, 180
surrogate keys, 176
UNION in Bridge entries, 181–182
variations, coding structures, 187
patient address facts, 164–170
patient master data, 159–162
quick fact queries, 170–172
recap of simple ETLs, 172–175
sources selection and preparation
adding simple facts, 156
admission facts, 152
birth facts, patient, 151–152
calendar, 151–153
caregiver dimension, 155–156
cities, 153
countries, 152–153
data access, 150

- data identification, 155
 - death facts, patient, 152
 - departure facts, 152
 - discharge facts, 152
 - encounter data, 156
 - encounters, 152
 - fact enrichment, 155–156
 - generating facts, 150
 - geopolitics hierarchies, 154
 - home address facts, 154
 - master data dimensions, 150
 - patient and encounter data, 150–151
 - postal zones, 153
 - proof-of-concept, 154
 - queries, 154
 - re-creation, tables, 154
 - registration facts, 152
 - states, 153
 - visualization, 155
 - work address facts, 154
 - surrogate keys (*see* Surrogate keys)
 - verifying and validating (V&V)
 - dimension-fact counts, 190–192
 - dimension–subdimension counts, 190
 - metadata-fact counts, 189
 - objectives, 189
 - recreating sources, 191–194
 - Almanac dimensions, biomedical data
 - warehouse
 - calendar, 99–100
 - clock
 - clinical usage, 101–102
 - data, time-period instances, 100
 - grains, 100–101
 - multiple time zones, 102
 - performance, 101
 - recorded events, 101
 - subdimensions, 100
 - time-oriented queries, 100
 - elements, 96
 - environment, 102
 - geopolitics
 - boundaries, 96
 - complexities, 97
 - data clean-up, 98
 - entities, 96
 - fact table, 97–98
 - free-text data facts, 98–99
 - hierarchy, 97–98
 - messiest data, 98
 - storing addresses, 99
 - star design, 77
 - structure, 102–103
 - system, 103
 - UOM (*see* Unit of measure (UOM))
 - Audit trail facts
 - database management system, 407
 - ETL logic, 408
 - implementation, 409
 - slowly-changing data, 409–410
 - Unknown Flag set, 408
 - UOM dimension, 409
 - warehouse, 408
 - Auto-adopt orphan definitions, 374
- B**
- Bacterial pneumonia, 596–597
 - Basic formal ontology (BFO)
 - calendar and clock dimensions, 114
 - continuants
 - aggregates, 44, 46
 - analysis, data sources, 46
 - categories, 44–45
 - dependent, 44, 114
 - fact tables, 46
 - functions and dispositions, 47
 - independent, 44, 114
 - object boundaries and sites, 46
 - perspectives, 47
 - qualities, 46–47
 - spatial regions, 44
 - star schema warehouse, 46
 - definition and classification, 43
 - feedback loops, 43
 - identification and analysis, data
 - elements, 43
 - mapping, 109–110, 113
 - occurrents
 - aggregation, 52
 - categories, 47–48
 - context, 52–53
 - process boundary, 52
 - processual entities, 51–52
 - spatiotemporal regions, 50
 - temporal regions, 48–50
 - Biomedical data; *see also* Warehouse data
 - EHR databases, 1
 - informatics development, 1
 - large-scale database technologies, 1
 - nature of
 - and actual clinical data, 2–3
 - complexity, 2–3
 - differences, 1–2
 - full-fledged textual reports, 3
 - incomplete, 3

IRB, 4
 privacy, 3
 regulation, HIPAA, 3–4
 storage and tracking, 3
 structure, 3

Biomedical warehouse
almanac (see Almanac dimensions, biomedical data warehouse)
 Alpha and Beta versions, 75
 analysis dimension
 annotation dimension, 105–106
 quality, 106
 control, 106–107
 data, star design (*see Star dimension design pattern*)
 facts, 78–79
 Gamma version, 75–76
 implementation, 75
 master dimensions
 caregiver, 82–83
 interactions, 83–84
 object instantiation, 85
 organization, 80–81
 queries, 79
 structure, 79–80
 studies, 81–82
 subject, 84–85
 reference dimensions
 Accounting, 94–95
 Diagnosis, 86–87
 elements, 86
 Facility, 93–94
 Material, 92–93
 -Omics, 95–96
 Pathology, 87–90
 Procedure, 90–91
 Treatment, 91–92
 requirements alignment, 109–110, 113
 BFO (*see Basic formal ontology (BFO)*)
 calendar and clock dimensions, 114
 design sequence, 114
 geopolitics, 114
 interactions and operations, 114–115
 ontology, 109
 processes, 115
 queries, 114
 semantic consistency, 115
 star schema, 75

Break Identifier, 272–273

BRIDGE_COMPOSITE literal, 385

Business requirements
 categories, 7–8, 565–566
 Gamma version, 565–566

implementation, 8
 initiatives, 8
 planning, 8
 prioritization, 7

C

Cohort identification
 actual syntax, 526
 core warehouse problem, 524
 CQT, 524
 definition, 528
 deidentified data, 527–528
 diabetes diagnosis, 525
 Diagnosis dimension, 526
 dimensional filters, 526
 fact metadata, 526
 female patients, 524
 generic query, 527
 ICD-9 (Id 250) data, 526
 ICD-10 (Id E08–E13) data, 526
 intersection of dimensional filtering, 525
 Metformin concept, 526
 open source i2b2 platform, CQT, 528
 permutations, 527
 RxNorm ontology, 525
 selection of patients, 524
 subqueries, 525
 timing aspects, 526–527
 working diagnoses, 525

Cohort Query Tool (CQT), 524–525

Consolidated SDI table, 303–305

Context Key 01, 272

Context tables, 412–415

Control dimensions
 Datafeed
 biomedical warehouse, 109
 design completion, 226
 ETL fact pipe, 401
 data state
 biomedical warehouse, 109
 design completion, 224–225
 ETL fact pipe, 400–401
 fact tables, 458
 master dimensions, 79, 142
 metadata, 107–108
 operation
 biomedical warehouse, 109
 design completion, 225
 processing, 78

Cross-ontology mappings, 602

Current Procedural Terminology (CPT), 59–60, 91, 129, 133, 362

D

- Database administrator (DBA), 14, 340, 451, 453, 457, 569–570
- Data construction
- control values
 - characteristics, LTB metric, 499–500
 - exceptional, 500
 - Length-of-Stay metric, 499–500
 - Metric Fact table, 498–499
 - thresholds, 500
 - displays
 - benchmarked metric, 502
 - dashboards, 500–501
 - metric aggregation, 502–507
 - metric-control fact, 507–510
 - scorecards, 501
 - skill sets, 501
 - targeted metrics, 501–502
 - users query, 501–502
 - generic ETL subsystem, 481
 - minimize data seams
 - additional metadata, 485–486
 - complexity, 482
 - conceptual, 484–485
 - dimension-fact architecture, 486
 - encounter table, 482
 - flexibility and extensibility, 484
 - historical and prospective data sources, 482
 - HL7 messages, 482
 - jagged edge, 484
 - multifact sources, 485
 - orthogonal data, 482
 - outcomes, 482–483
 - performance advantages, 483
 - physical organization, 484
 - queries, 481–482
 - routine data element, severity property, 483
 - SDI job, 482, 486
 - sources, 484
 - temporal/conceptual points, 481, 483
 - vital-sign observations, 484–485
 - warehouse users, 481
 - Data control
 - Data Control Points table, 304–305
 - data monitoring
 - decision-making process, 465
 - orphan tracking, 469–473
 - pipeline counts, 466
 - quality and management, 466
 - system rows, 466–467
 - unexpected and undesired values, 467–469
 - implementation, 476–479
 - redaction
 - additional LEFT JOIN, 464
 - in a CASE statement, 465
 - column-based access, 464
 - definition row, 464
 - deidentified view, 462
 - effective balance, 462
 - EHR system, 462–463
 - fundamental issue, 462
 - HIV-related/psychology procedures, 463
 - noncompliant privacy-related behaviors, 462
 - privacy controls, 463
 - research settings/clinics, 465
 - warehouse design, 463
 - security controls, 475–476
 - surrogate merges, 473–475
 - warehouse
 - design (*see* Warehouse design)
 - support team, 479–480
 - Data Control Points table, 304–305
 - Datafeed Break, 269, 272, 274, 302, 382, 401
 - Data governance
 - and administration, 301, 608–609
 - agreement, 403
 - capabilities, 451
 - controls, 240
 - data owner, 607
 - delivery, 514
 - effective, 247
 - expanded data sharing, 613–614
 - function, 12, 324, 436, 444, 469, 488, 565
 - governance body, 606
 - group, 463, 466, 468, 471, 474, 541, 606
 - immediate decisions, 610
 - long-term evolution, 614–615
 - master data inconsistencies, 613
 - mature, 240
 - maturity process, information technology, 615
 - medium-term growth, 611–612
 - near-term control, 610–611
 - organization, 605
 - ownership, data source, 608
 - process, 467
 - release management, 612
 - source application feedback, 612–613
 - tools, 468

- translational medicine, 615–616
- warehouse support team, 609
- Data sourcing
 - analysis, 233
 - columns selection
 - avoiding redundancy, 262
 - Beta version, 261
 - complexity, 261
 - derived, 261–262
 - ETL loads, 260–261
 - generic ETL architecture, 260
 - identifying patients, 262
 - internal system keys, 262
 - joint tables, 262
 - lab results, 262
 - low and high reference value, 261
 - multiple variations, people
 - names, 261
 - omitting, 260
 - relationship, source's Person ID and MRN, 263
 - SDI extraction, 263
 - specific, 260
 - variations, 261
 - coverage and seamlessness
 - admission and discharge history, 235
 - admission facts, 236–237
 - analysis, retrospective and prospective data, 235
 - biomedical data warehouse project, 234
 - calculation, LOS, 235
 - cutover point, 234
 - discharge disposition, 236
 - EHR, 234–235
 - fact level, 236
 - fact table, 234, 237
 - HL7 messages, 234–235
 - integration, 234
 - loading, 234
 - principle complaint, 237
 - single fact historical loading, 235
 - dimensionalizing facts
 - actual clinical result and remark column, 258
 - foreign key points, 257
 - lab result table, 259
 - level, data normalization, 257
 - outcomes, 257–258
 - row-to-row variation, 259
 - sequence loads, 257
 - single, 257
 - source tables, 258
 - updimensionalizing risk, 258
 - ETL
 - processes, 233
 - workflows, 259
 - fact superseding (*see* Superseded facts)
 - functional normalization
 - allergy assessment, 240–241
 - allergy types, 241
 - analysis, 2 × 2 framework, 239
 - analytical process, 238
 - challenges, 242
 - conservative analysis, 240
 - drug allergies, 241–242
 - EHRs, 240–241
 - elimination, 243
 - Gamma and Beta version, 240
 - lab outcomes, 242–243
 - loading data, 242
 - medication administrations, 238–240
 - querying, 238
 - reduction, 238
 - tracking, 243
 - loading data, 259
 - organization implements, 234
 - qualities of facts, tables, 243
 - risks, 233–234
 - rows selection, 263–264
 - SDI hobs, 260
 - single generic ETL workflows, 233
 - time selection
 - allergy assessment selection, 265
 - calendar dimensionalization, 265–266
 - controlling dates, ETL, 264–265
 - extra and cross-referencing facts, 266
 - Onset Date, 265
 - querying, 266
 - retrospective and prospective processing, 264
 - warehouse development project, 233
 - DBA, *see* Database administrator (DBA)
 - Default Key 01, 269, 272, 318
 - Delivering data
 - advantages, 511
 - business intelligence, 539–541
 - cohort identification (*see* Cohort identification)
 - external data marts
 - concept dimension, 557–560
 - description, 548
 - fact table (*see* Fact table)
 - i2b2 data (*see* i2b2 data)
 - and internal data mart views, 548
 - patient dimension, 553

- provider dimension, 550–552
 - visit dimension (*see* Visit dimension, delivering data)
 - fact count queries, 528–529
 - flattened facts, 547–548
 - flattened groups (*see* Flattened groups)
 - metadata browsing, 520–523
 - privacy-oriented usage profiles, 518–520
 - timeline query
 - ADT, 537
 - Calendar, Clock and Unit of Measure, 531
 - fact type, 529–530
 - fact value, 531–532
 - generic patient, 533
 - Interaction Master ID, 530
 - lab results, 535
 - material and quality descriptions, 536
 - procedure descriptions, 534
 - Procedure, Material and Diagnosis dimensions, 535
 - procedure quality descriptions, 535
 - returned row, 538–539
 - Subject Master ID, 530
 - toolset, 537–538
 - traditional view, 511
 - warehousing program, 511–518
 - Depth From Ancestor (DFA), 138
 - Descriptive statistics, 593–594
 - Design completion
 - Alpha version, 197–198, 208, 212, 217–218, 222, 226, 232
 - Beta version, 197–198, 202, 207–208, 212, 217, 226
 - components, 197
 - control dimensions
 - data feed, 226
 - data sate, 224–225
 - operation, 225
 - functionality, 198
 - metadata
 - loading, 232
 - mappings (*see* Metadata mappings, warehouse design)
 - reinitializing (*see* Reinitializing the warehouse design)
 - UOM dimension (*see* Unit of measure (UOM))
 - Diagnosis-Related Group (DRG), 87, 236, 545–546
 - Dimensional data modeling
 - affinity (*see* Affinity analysis, dimensional data modeling)
 - Alpha version, 17
 - anticipating, 32
 - evolution, data warehouses
 - dimensional design, 19–22
 - dimension tables, 17
 - fact tables, 17
 - implementation, traditional database, 17
 - queries, 17
 - relational data, 17
 - relational normalization, 18–19
 - star schema (*see* Star schema)
 - transposing (*see* Transposing dimensional schema)
 - Directed Acyclic Graph (DAG), 138
 - DRG, *see* Diagnosis-Related Group (DRG)
- E**
- Electronic health record (EHR), 1, 6–7, 41–42, 83–84, 90, 98, 107, 129, 234–235, 239–242, 248, 262, 272, 324–325, 345–346, 462, 469, 482, 484, 498, 522, 563
 - Enterprise Identifier, 272
 - ETL Definition pipe
 - auto-adopt orphan definitions, 374
 - day-to-day operations, 375
 - DELETE, 372
 - individual source load, 372
 - INSERT and UPDATE statements, 372
 - insert new definitions, 374–375
 - INSERT statement, 356–359
 - master loads
 - Alpha version, 354
 - auto-adopt transactions, 353
 - Beta version, 356
 - insertion, Diagnosis dimension, 354
 - metadata transformation join, 353
 - patient, 352
 - quality control characteristics, 356
 - Reference pipe data, 354–355
 - rudimentary orphan adoption
 - update, 355
 - simple diagnosis definition data, 353
 - simple SNOMED metadata mappings, 353
 - SNOMED codes, 353
 - unpivoted SNOMED metadata source data, 354
 - multiple simultaneous transactions, 364–367
 - new dimension IDs, 373
 - new orphans, 359–360
 - orphan auto-adoption, 360–361
 - processing complexities
 - Beta version of warehouse, 346
 - deep and wide staging, 348–352

- dropping transaction values, 345
- execution, 346
- historical encounters, 347
- logical pathways, 345
- metadata transformation, 347–348
- Orphan and Placeholder flags, 345
- Subject dimension, 345
- subsystem, 343
- transaction-oriented design, 345
- SCDs (*see* Slowly-changing dimensions (SCDs))
- sequence diagram, 343–344
- update existing definitions, 375
- ETL PIVOT statements, 386
- ETL Reference pipe
 - Definition ID, 311
 - logic of, 312
 - Master ID, 311
 - metadata transformation (*see* Metadata transformation)
 - process, 309
 - sequence diagram, 309–310
 - transaction types, 309
- ETL transactions
 - add dimension master, 445
 - add fact, 445
 - add/update logic, 446
 - controls
 - Break Identifier, 273
 - Context Key 01, 272
 - Datafeed Break, 274
 - Default Key 01, 272
 - Enterprise Identifier, 272
 - generic ETL process, 274–275
 - highest-order key source column, 272
 - Hospital Code column, 272–273
 - Source Timestamp identifies, 273–274
 - standard intake columns, 272
 - Target State control column, 275
 - deletion, dimension master, 445
 - description, 424
 - generic ETL logic, 446
 - initialize dimension master, 445
 - SDI intake datasets, 445
 - update dimension master, 445
 - update fact, 445
 - warehouse database, 446
- ETL workflows
 - Alpha version, 267
 - alternatively sourced keys
 - Beta version, 425
 - description, 423
 - generic architecture, 425
 - in the reference pipe, 428–431
 - sourcing compound natural keys, 425–427
 - sourcing warehouse surrogates, 427
- Beta version functionality, 284
- cloning approach, 286
- components, 285–286
- control, transaction (*see* ETL transactions)
- data control
 - layers, 300–304
 - pipe entries, 299–300
 - points, 304–305
- dataset controls
 - fact-specific metadata, 272
 - job identifier, 268–269
 - lab results, 270
 - Logical Dataset and Fact Break columns, 270
 - nonresult entries, 271
 - physical–logical–fact source controls, 270
 - redundant entries, 271
 - standardized source data, 268
 - textual column, 270
 - three-tiered hierarchy, 269
 - warehouse's procedure, 271
- external *vs.* internal sourcing, 283–284
- Gamma version, 285, 423
- hard-coded literals, 286
- job ID, 300
- Master IDs to ETL layers, 305–306
- metadata transformation (*see* Metadata transformation)
- pipes
 - Alpha version, 287
 - Beta version, 287
 - checkpoint, restart and bulk loading, 290–291
 - definition, 289
 - reference, 288–289
 - simplified sequence diagram, 288
 - subsystems, 287
 - types of data, 287
- SDI design pattern (*see* Source data intake (SDI))
- single point of failure, 286–287
- single set of, 284
- source data
 - consolidation, 282
 - SDI tables, 267–268
 - values, 275
- sourced metadata
 - ancestor and descendent entries, 431
 - COALESCE/CASE statement, 432

- Definition pipe, 432
 - description, 423–424
 - fact pipe, bridges, 432
 - functional expansion, 431
 - Metadata Definition table, 432
 - metadata dimension, 431
 - reference pipe, 432
 - SDI query, 431
 - standard data editing
 - data type checking, 433–436
 - description, 424
 - generic nature, 432
 - range checking, 436–437
 - timestamp handling, 433
 - value trimming and cleanup, 433
 - standard intake dataset, 268
 - standardized 13 SDI source columns, 268–269
 - superseded facts
 - Beta version, 447
 - description, 424
 - fact table, 447
 - laboratory result, 449
 - Metadata dimension row, 449
 - Metadata hierarchy, 450
 - one-per-Subject and
 - one-per-Interaction, 448
 - superseding logic, 447–448
 - values presume, 449
 - target states, 424, 446–447
 - traditional approach, 284
 - traditional architecture, 425
 - undetermined dimensionality
 - advantages, 442, 444
 - datasets, 441
 - description, 424
 - DISTINCT Subject dimension entries, 443
 - generic capability, 444
 - in healthcare, 441
 - multi-subject facts, 443
 - Person subdimension, 443
 - SDI selection query, 441
 - value-level unit of measure (*see* Unit of measure (UOM))
 - wide *vs.* deep data, 306–307
 - Extensive health record, 1
 - Extract-Transform-Load (ETL)
 - Definition pipe (*see* ETL Definition pipe)
 - execution
 - auto-adopt, 346
 - new definition rows, 346
 - new orphan rows, 346
 - updates, 346
 - PIVOT statements, 386
 - Reference pipe (*see* ETL Reference pipe)
 - subsystem pipes (Beta version), 287
 - transactions (*see* ETL transactions)
 - workflows (*see* ETL workflows)
- F**
- Face, Legs, Activity, Cry, Consolability (FLACC)
 - score, 72
 - Factless facts, 380–382
 - Fact pipe, ETL
 - bridge staging, 384–385
 - build facts
 - Bridge pivot, 392–393
 - Fact table entries, 392
 - Reference pipe, 391
 - value alignment, 394
 - calendar and clock, 398–399
 - Datafeed, 401
 - data state, 400–401
 - deep references to wide facts, 383
 - finalizing columns, 394
 - Group ID, 383
 - group lookups, 387–389
 - group staging, 386–387
 - insert fact values, 401–402
 - insert new bridges, 391
 - insert new groups, 390–391
 - Master IDs, 383
 - new group surrogates
 - Group IDs to bridges, 390
 - surrogates to Group IDs, 389–390
 - optional dimensions, 400
 - organization, 399
 - Placeholder Flag, 384
 - process, 377–378
 - Reference pipe, 383
 - superseding facts, 402–405
 - transformation, metadata (*see* Metadata transformation)
 - UOM (*see* Unit of measure (UOM))
 - Fact superseding metadata columns, 403
 - Fact table
 - anticipated cohort, 561
 - BFO continuants, 46
 - cohort of patients, 560
 - common cross-dimensional Concept set, 563–564
 - Concept key in i2b2, 562
 - control dimensions, 458
 - data mart, 560
 - data sourcing, 234, 237

- dimensional data modeling, 17
 - DISTINCT clause, 562
 - entries, 392
 - extensions, 564
 - extraction, 560
 - Fact ID, 561–562
 - fact types, 561
 - finalizing Beta, 417–418
 - financial charge, 250
 - geopolitics, almanac dimensions, 97–98
 - i2b2 dimensions, 560
 - IRB approval, 564
 - Master ID, 564
 - Metadata mappings, 212
 - metric
 - aggregated parameters, columns, 490–491
 - Alpha and Beta versions, 490
 - control, 491–492
 - definitions, 489
 - measures, 488–489
 - properties, 490
 - size, 491
 - support, 489
 - UOM, 491
 - Metric Fact table, 498–499
 - MRN, 563
 - NKAs, 561
 - star dimension design pattern, 142–144
 - superseded facts, 447
 - transposing dimensional schema, 31
 - values, UOM (*see* Unit of measure (UOM))
 - warehouse design control, 457–462
 - Fanned-out INSERT statement, 391
 - Finalizing Beta
 - audit trail facts
 - database management system, 407
 - ETL logic, 408
 - implementation, 409
 - slowly-changing data, 409–410
 - Unknown Flag set, 408
 - UOM dimension, 409
 - warehouse, 408
 - Datafeed dimension, 410–411
 - data warehouse, 411
 - Gamma version, 419
 - structural verification
 - Bridge tables, 416
 - Context tables, 412–415
 - Definition tables, 415–416
 - ETL execution, 412
 - fact table, 417–418
 - generic ETL pipes, 412
 - Group tables, 416–417
 - metadata, 418–419
 - traceability, 411
 - transparency, 411
 - Flattened groups
 - baseline two-quality fact characterization, 542
 - Definition rows, 543
 - Diagnosis dimension, 544
 - DRG and Primary Diagnosis, 545–546
 - orthogonal queries, 542
 - outrigger pairing abnormalcy with status, 544
 - primary ICD-9 and DRG diagnosis, 546
 - Quality dimension, 541, 543, 545
 - source columns, 542
 - Vernacular Code, 543–544
 - wider data access to users, 547
- ## G
- Gamma finalizing
 - capabilities/controls, 574
 - categories, business requirements, 565–566
 - configuration management
 - Alpha version, 567
 - Beta system, 567
 - direct data transfer, 567–568
 - ETL job, 568
 - information technology function, 566–567
 - production environment, 566–567
 - resourcing, 568
 - turnover activities, 568
 - UOM, 567
 - database tuning, 569–570
 - HIPAA redaction, 572–573
 - implementation, 574
 - IRB integration, 573–574
 - job scheduling, 568–569
 - merge/unmerge, patient
 - administrative transactions, 570
 - complication, 570
 - data states, 571
 - ETL pipe, 572
 - ETL transaction, 570
 - HL7 messages, 570
 - logical dataset, 571
 - Master ID, Reference table, 571
 - previous patient, 570
 - source data loading, 570–571
 - surrogate, 572
 - traceability and transparency, 571
 - transaction process, 571–572

H

Health Insurance Portability and Accountability Act (HIPAA), 3, 97–99, 106, 154, 247, 451
 controller, 170, 512–513, 589
 governance group, 589
 layers, 519
 Privacy Rule, 518
 redaction, 572–573
 Hierarchy tables, star design pattern
 ancestor–descendent relationship, 137
 construction, 137
 DAG and network structure, 138
 internal control and rigor, 138–139
 natural
 context reference labels, 141
 control flags, 140
 crossover, 140–141
 flat hierarchies, 140
 internal, 140–141
 SQL CREATE statement, 141–142
 organization chart, 138
 perspective column, 138
 records, DFA column, 138
 SCD controls, 140
 tree structure, 137–138, 140
 user-/source-defined perspectives, 138
 HIPAA, *see* Health Insurance Portability and Accountability Act (HIPAA)
 Hospital Code column, 272–273
 Hospital Code source column, 271–272
 Human disease ontology, 596

I

i2b2 data
 columns, 550
 in Concept dimension, 549
 design parameters, 549
 design point, 548
 external data mart candidate, 549
 one-time/throw-away exports, 549
 UOM dimension, 549
 Information carrier (IC), 56–57
 Information content entity (ICE), 55–56
 Institutional Review Board (IRB), 515, 519

K

Knowledge synthesis
 aggregates, 580
 ambulatory/emergency encounters, 583

annotation, semantic (*see* Semantic annotation, knowledge synthesis)
 base timeline, 592
 billing and payment facts, 589
 calendar dimension, 587
 calendar filter, 581–582
 census query, 587
 correlation analysis, 584
 derivative data, 585–586
 diagnosis, fact count, 579
 encounter type, 583
 fact counts, 577–578
 filtering, query, 581
 and generation, 577
 hierarchic relationship, 592
 ICD-9 codes, 580
 ICD-9 diagnosis, 581
 inter-event timings, 587
 learning curve, 578
 material dimension, 593
 metadata, 578
 nonpatient timelines, 590–591
 organization dimensions, 579
 power of timeline, 592
 procedure dimension, 589
 production warehouse, 578
 ranking
 lists, 583
 query, 582
 tier diagnostic hierarchy, 580–581
 timeline query, 588–589
 UOM, 578
 warehouse dimensions, 591

L

Length of stay (LOS), 49, 83, 235, 493, 495–500, 586

M

Material Information Bearer (MIB), 54–55
 Medical record number (MRN), 129, 132–133, 159–161, 173, 216–218, 262–263, 324–326, 333, 563–564
 Metadata mappings, warehouse design
 Alpha version, 208
 basic lab outcome data, 222–224
 Beta version, 208
 bridged and unbridged context, 208–209
 definition, 209–210
 ETL processing, 208
 patient address facts

- Alpha version, 218
- bridged and unbridged
 - data, 219–220
- codeset translation parameters, 221
- country, state and city, 220–221
- definition metadata, 220
- load data, countries, states and cities, 218
- logical dataset, 220–221
- processing, 221–222
- reference columns and values, 218
- specifications, 222
- patient master data
 - codeset translation, 218
 - elements, reference table, 216
 - gender code set, 211, 218
 - MRN values, 217–218
 - physical dataset name, 216
 - population, 217
 - processing, 211
 - reference pipe, 218
 - source columns, 216–217
 - subject definition table, 217
 - subject dimension, 216
- properties
 - Alpha version, 212
 - Beta version, 212
 - codeset translation, 214
 - definition columns, 212
 - ETL processing, 212–213
 - fact table, 212
 - flat hierarchy, 215
 - implicit UOM metadata, 213
 - superseding, 213–214
- reference, 209–210
- source data, 208
- targeting, source value, 211–212
- Metadata transformation
 - alias entries, 332–334
 - “~All~” value, 296–297
 - bridges and groups, 334
 - codeset translations, 292–294, 314–316
 - early *vs.* late binding, 297–299
 - ETL fact pipe
 - factless facts, 380–382
 - sourced facts, 378–380
 - source-to-target mapping, 378
 - staged facts, 382
 - flat hierarchies, 335–336
 - general *vs.* functional, 294–296
 - hierarchy entries
 - new self-hierarchies, 335
 - types of, 334
- INNER JOIN, 292
- joining source data, 291
- natural hierarchies
 - candidates, 337
 - flat hierarchy cascade, 340–342
 - Geopolitics Contexts, 336
 - Geopolitics Reference data, 337
 - hypothetical rows, Miami, Florida, 337–338
 - ICD-9 code 384, 338–339
 - Origin Flag, hierarchy entry, 338
 - terminus resolution, 339–340
 - types of, 336
- reference composite
 - beta limitations, 320–321
 - broader query, 317–318
 - Context Key 01 values, 318
 - innermost subquery, 319
 - mappings, 318
 - reference staging, 319–320
- reference entries, 332
- reference pipe process, 312–313
- reference processing, 316–317
- resolve references
 - alias entry collisions, 324–327
 - Context ID, 322
 - Diagnosis dimension, 321–322
 - ICD-9 diagnosis codes, 322
 - LEFT JOINS, 322
 - unknown/broken keys, 323–324
- source data, 292
- source data table and codeset dimensions, 313–314
- unresolved references
 - alias propagation, 331–332
 - Composite key, 327
 - distinct composites, 328–329
 - source Break ID, 327–328
 - surrogate assignments, 329–330
 - transactional redistribution, 330–331
- WHERE clause, 314
- Metrics, data construction
 - aggregation (*see* Aggregated metrics)
 - challenges, 487
 - and control facts
 - complicated queries, 510
 - conceptual model, 508
 - hierarchy, 508–509
 - perspectives, 510
 - relationships, 508
 - results, 509–510
 - sample code, 509

- value column, 508–509
 - values, 507–508
 - environment, 487
 - healthcare, 487
 - information technology projects, 486
 - new dimension and fact tables
(*see* Fact table)
 - process design, 487–488
 - and reports, organization, 486–487
 - subdimension
 - calculate base values, 492
 - control, 495
 - data integration, 492
 - definitions, 494–495
 - display, 496
 - documentation, 495
 - goodness indicator, 494
 - Length-of-Stay, 493–494
 - properties, 494
 - values, 493
 - values
 - ADT facts, 497–498
 - encounter, timestamp, 497
 - ETL processes, 496–497
 - Length-of-Stay, 496–497
 - measures, 496
 - Metadata dimension, 498
 - MRN, *see* Medical record number (MRN)
- N**
- NarrowMatch mappings, 600
 - New dimension IDs, 373
 - No Known Allergies (NKAs), 238, 241, 243
 - Non-CSD value, 365
- O**
- OGMS, *see* Ontology for general medical science (OGMS)
 - Ontology for biomedical investigations (OBI)
 - concepts, 57–58
 - practicing biomedicine, 64
 - processes
 - acetaminophen, 60
 - aggregates, 61–62
 - auto-curate existing data, 60
 - conduct data analysis, 58
 - continuants and occurrents, 59
 - CPT ontology, 60
 - drugs, 60
 - heuristics, 59
 - lab test, 59
 - loading, 59
 - processual contexts, 63–64
 - semantic meaning, 59
- Ontology for general medical science (OGMS)
- aggregate, 66
 - clinical dysfunction, 67
 - clinical practices, 65–66
 - concepts, 65
 - continuant concepts, 66
 - data source, 65–66
 - human organisms, 66–67
 - occurrent concepts, 65–66
 - pathology, 66
 - processual context, 66
- Orphan auto-adoption, 360–361
- Outrigger
 - person and specimen, 456
 - sample, 455
- P**
- PARTITION clause, 385
 - Personal Health Record (PHR) system, 41, 49
 - Physical–logical–fact source controls, 270
 - Primary care provider (PCP), 255
- Q**
- Qualities of facts, data source
 - allergy assessment
 - common data properties, source table, 247–248
 - factless fact, 249
 - free text values, 249
 - Gamma and Beta versions, 249
 - reaction values, 248–249
 - semantic tools and annotation capability, 249
 - severity column, 248
 - analytical cubes, 244
 - exceptions, 243–244
 - financial charge
 - accounting, 251–252
 - analytic mapping strategy, 252
 - Charge Master analysis, 252–253
 - data patterns, 254
 - data points, 250
 - degree of freedom challenge, 252
 - degree of freedom problem, 250–251
 - dimensional connections, units, 252
 - extended charge facts, 252, 254
 - extended charge, 250–251
 - fact table, 250

- Gamma version, 254
- medication, 253
- non-NULL Unit Charge, 254
- performance, 252
- post hoc technique, 254
- properties, 254
- quality, 254
- SQL queries, 252
- Unit Charge, 250–251
- lab outcomes
 - abnormalcy, 245–246
 - analysis, source dataset, 244
 - clinical, 245
 - columns, 244–245
 - comment data, 246–247
 - dimensional values, 244
 - low and high reference value, 246
 - reference range, 246
- nonbiomedicine, 244
- quantitative, 244
- storage and counting, 244
- traditional business data, 244

R

- Reference dimensions, biomedical data
 - Accounting, 94–95
 - Diagnosis, 86–87
 - elements, 86
 - Facility, 93–94
 - Material, 92–93
 - Omics, 95–96
 - Pathology, 87–90
 - Procedure, 90–91
 - Treatment, 91–92
- Reinitializing the warehouse design
 - Alpha and Beta data, 226
 - configuration data
 - context entries, 230
 - default dimensionality, 231–232
 - implementation, design, 229–230
 - values and scale, 230
 - data states, 228–229
 - design pattern, 226
 - empty state, 226
 - process, 226
 - six system rows, 226–228
 - UOM, 220, 229

S

- SCDs, *see* Slowly-changing dimensions (SCDs)
- SDI, *see* Source data intake (SDI)

- Semantic annotation, knowledge synthesis
 - automatic processing, 600
 - bacterial pneumonia, 596
 - broadMatch and closeMatch, 603
 - closeMatch and exactMatch, 601
 - cross-ontology mappings, 602
 - exactMatch relationship, 600–601
 - human disease semantics, 598
 - ICD-9 code and ontology, 597
 - inverse relation, 600
 - loose-fitting relationship mapping, 600
 - mappingRelation, 599
 - narrowMatch mappings, 600
 - relatedMatch, 599
 - SKOS, 599
 - in warehouse, 596
- Semantic layers
 - BFO (*see* Basic formal ontology (BFO))
 - intelligence, 542
 - myriad data, 42
- Simple Knowledge Organization System (SKOS)
 - construction, mappingRelation, 599
 - cross-reference concepts, 601
 - errors, 602
 - generation, 599–600
 - hierarchic element, 601
 - legitimate and warranted relationships, 601–602
 - loose-fitting relationship, 600
 - mappings, 599–600
 - natural hierarchy entries, 600
 - semantic mappings, 601
 - symmetric, 599
 - types, 602
 - user queries, 600–601
 - XY-broadMatch-L, 602–603
- Slowly-changing dimensions (SCDs)
 - advantages, 362
 - complication, 361
 - control, timestamps, 123–124
 - control, City Medical Center
 - changes, Master ID and Definition ID, 126
 - connections, different definition entries, 128
 - deletion, 127–128
 - features, 125–126
 - late-arriving changes, 126–127
 - SQL CREATE statement, 128–129
 - tracking, hospitals, 125
 - Definition ID, 362–363
 - Definition row, 363–364

- gender, 362
- Master ID, 363
- realignment queries, 364
- Source Timestamp, 364
- transaction sets
 - assigning deep row numbers, 368–369
 - assigning relative wide row numbers, 370–372
 - distribute deep non-SCD updates, 369–370
 - Master IDs, 367
 - staging existing definitions, 367–368
- SNOMED codes, 353–354
- Source data; *see also* Data sourcing
 - clinical diagnosis, 71
 - epistemological
 - hypotheses, 70–71
 - observational, 68–70
 - principle, 71–72
 - validity, 73
 - FLACC score, 72
 - implicit *vs.* explicit
 - causes, 41
 - database management systems, 41–42
 - EHR system, 41
 - load jobs, 42
 - manipulation and load, warehouse, 40
 - metadata, 41
 - portal PHR system, 41
 - unit of measure, 42
 - vital signs table, 41
 - information artifacts
 - high-level categorization, 53
 - IC, 56–57
 - ICE, 55–56
 - information loading, 53–54
 - MIB, 54–55
 - information technology, 39
 - interaction, 72–73
 - laboratory result fact, 72
 - layers, semantic (*see* Semantic layers)
 - OBI (*see* Ontology for biomedical investigations (OBI))
 - OGMS (*see* Ontology for general medical science (OGMS))
 - ontological levels, 67–68
 - Source data intake (SDI)
 - design pattern
 - desired source data, 278
 - Fact Break, 280
 - innermost subquery, 279
 - join statements and filters, 278
 - required 13-column format, 280–281
 - single Master ID, 279
 - sourcing subquery, 279–280
 - standard 13-column configuration, 282
 - jobs, 260
 - Sourced facts, 378–380
 - Sourced metadata
 - ancestor and descendant entries, 431
 - COALESCE/CASE statement, 432
 - Definition pipe, 432
 - description, 423–424
 - fact pipe, bridges, 432
 - functional expansion, 431
 - metadata Definition table, 432
 - metadata dimension, 431
 - reference pipe, 432
 - reference pipe, hierarchy, 432
 - SDI query, 431
 - Source Timestamp, 273–274, 372
 - Source-to-target mapping, 378
 - SQL, *see* Structured Query Language (SQL)
 - Staged facts, 382
 - Standardized 13 SDI source columns, 269
 - Star dimension design pattern
 - bridge table, 134–135
 - Cartesian product, 132
 - changes
 - control, SCD (*see* Slowly-changing dimensions (SCDs))
 - effective and expiration timestamp, 121
 - effective-expiration period, definition
 - row, 121–122
 - effective-expiration periods, active
 - definition row, 122–123
 - faster-changing values, 121
 - final configuration, 123
 - generic ETL, 122–123
 - implementation choice, 121
 - queries, 123
 - SCD tracking, 124–125
 - status indicator, 121
 - tracking, 122
 - track versions, 121
 - values, 124
 - version controls, 124
 - volatility, 124
 - characteristics, 76
 - composite values, 136
 - context table, 129–130
 - development, Alpha version, 117
 - dimension
 - almanac, 77
 - analysis, 77
 - control, 77

- master, 76
 - reference, 76
 - fact table, 142–143
 - group table, 134
 - healthcare data, 76
 - hierarchy tables (*see* Hierarchy tables, star design pattern)
 - hypothetical fact, group and bridge organization, 136–137
 - interaction, group and bridge table, 136
 - lab test, 131–132
 - master data, Definition table
 - description, subdimension, 120
 - design pattern, 117–119
 - dimension-specific columns, 119–120
 - hypothetical subject dimension, 120
 - loss of standardization, 120
 - rows, 118
 - MRNs locations, 133
 - navigating
 - active status indicator, 145
 - Alpha version loading, 147
 - definition's effective and expiration timestamps, 146–147
 - fact table, 143–144
 - group table, 147
 - hierarchy table, 145
 - queries, 145–146
 - traversing different aliases, 146
 - queries, 76, 78
 - reference table, 130–131
 - relationship, context and reference table, 131
 - SQL CREATE statement, context and reference table, 133–134
 - structure, 117–118
 - training and awareness, 132
 - Vernacular codes, 132–133
 - Star schema
 - aggregate databases, 22
 - challenges, 24
 - design of, 23–24
 - dimensional warehousing, 22
 - dimensions and subdimensions, 24–26
 - five-dimensional limitation, 23
 - granularity
 - capability, store facts, 28
 - four-dimensional design model, 26
 - grain of degrees, 26–27
 - process, 28–29
 - simple fact, 26
 - timing, 28
 - import/export facts, 23
 - requirements, 23
 - simple healthcare model, 23–24
 - size and scale, 22
 - subdimensions, 24
 - Star-schema-architected data warehouse, 452
 - Structured Query Language (SQL), 18, 128, 133, 143, 176, 182, 252, 257, 306, 432, 452, 518
 - Superseded facts
 - Beta version, 447
 - data sourcing
 - active and superseded data, 256
 - changing data, 254–255
 - final active fact, 256
 - metadata, 255
 - outcomes, 256–257
 - patient address facts, 256
 - PCP and mailing address, 255
 - queries, 256
 - relationship, 255
 - source system, 255
 - description, 424
 - fact table, 447
 - laboratory result, 449
 - Metadata dimension row, 449
 - Metadata hierarchy, 450
 - one-per-Subject and
 - one-per-Interaction, 448
 - superseding logic, 447–448
 - values presume, 449
 - Surrogate keys
 - creation, tables, 157
 - deletion, 157–158
 - ETL system, 157
 - geopolitics dimension, 157
 - identifiers, 158
 - issues, 158
 - Master IDs and Definition IDs, 157–158
 - process column, 157
 - values, 156–157
- T**
- Transaction controls, ETL workflows
 - Break Identifier, 273
 - Context Key 01, 272
 - Datafeed Break, 274
 - Default Key 01, 272
 - Enterprise Identifier, 272
 - generic ETL process, 274–275
 - highest-order key source column, 272
 - Hospital Code column, 272–273

- Source Timestamp identifies, 273–274
- standard intake columns, 272
- Target State control column, 275
- Transposing dimensional schema
 - annotation, 29
 - clinical warehouse design, 32
 - data storage, 29
 - fact table, 31
 - generalized, 30–31
 - i2b2 data warehouse, 31–32
 - logical subdimensions, 29, 35
 - maturity pathway, 29–30
 - performance, 29
- fact table, 202
- feature, 207–208
- flexibility, 206–207
- Gamma version, 208
- identification, 202
- invalid, 206
- PRODSUM, desired values, 205
- queries, fact table, 203–205
- simple standard query, 206
- SUM query, 205–206
- table scan, 205
- undesirable fact table scans, 207
- volatility, 207

U

- Unit of measure (UOM)
 - almanac dimensions
 - Beta version, 104
 - datasets, 104
 - examples, 104–105
 - internal controls, 103
 - natural hierarchy, 104
 - semantic, 104
 - sources, Alpha version, 103–104
 - syntactic and semantic elements, 103–104
 - assigning the implicit, 397–398
 - class, 199–200
 - dimension
 - Alpha version, 198
 - conceptual hierarchy, 198
 - internal control mechanisms, 198
 - syntactic elements, 198
 - implicit, 396–397
 - interpretation of value, 395
 - interpretation, users, 201
 - language, 201–202
 - Master ID and Group ID, 395
 - Reference Composite, 395
 - scale, 199
 - semantic meaning, 201
 - sourced Value, 395
 - specific units, 200
 - Unexpected Text unit, 200–201
 - values
 - aggregate total, group of facts, 205
 - cardinality, 207
 - changes, query, 207
 - data volume, 206
 - entire, 203
 - ETL control, 206
 - fact pipe, 207

V

- Visit dimension, delivering data
 - ADT facts, 557
 - i2b2 by Master ID, 553
 - i2b2 data, 557
 - Interaction dimension, 554
 - Organization dimension, 555
 - Provider dimension, 554
 - split pathways, 556
 - Subdimension/Category entries, 554–555
 - subset of Visits, 557

W

- Warehouse data
 - administrations, 6–7
 - challenges, 6
 - design and constructions, 7
 - digital, 4
 - dimensionality, 4, 6
 - functional requirements
 - biomedical data, 8
 - data queries, 9
 - good outcomes, 8–9
 - issues and hypotheses, 10
 - loading data, 8–9
 - marts, 9–10
 - primary, 9
 - reports, 9
 - technical implementation, 8
 - hierarchy, database technologies, 5
 - incomplete projects, 11–12
 - implementation approach, 13–14
 - nature of data pathways, 4–5
 - organization, 12–13
 - performance and control, 11
 - products/materials, 4
 - sales organization, 5–6

Warehouse design, data controls
application layer issues, 452–453
Beta version, 451
database statistics, 451–452
DBA, 451
fact tables, 457–462
indexing and partitioning, 453–454
outrigger tables, 454–457

Warehouse support team, 586,
595, 609
Warehousing program
data owners, 513–515
data sources, 517–518
requirements, 511
support teams, 515–517
user analysts, 512–513

