# Part II. Data Flow

Chapter 3. Extracting

Once your data warehouse project is launched, you soon realize that the integration of all of the disparate systems across the enterprise is the real challenge to getting the data warehouse to a state where it is usable. Without data, the data warehouse is useless. The first step of integration is successfully extracting data from the primary source systems.

NOTE

PROCESS CHECK Planning & Design:

*Requirements/Realities* → Architecture → *Implementation* → Test/Release

Data Flow: *Extract* → Clean → Conform → Deliver

While other chapters in this book focus on transforming and loading data into the data warehouse, the focal point of this chapter is how to interface to the required source systems for your project. Each data source has its distinct set of characteristics that need to be managed in order to effectively extract data for the ETL process.

As enterprises evolve, they acquire or inherit various computer systems to help the company run their businesses: point-of-sale, inventory management, production control, and general ledger systems—the list can go on and on. Even worse, not only are the systems separated and acquired at different times, but frequently they are logically and physically incompatible. The ETL process needs to effectively integrate systems that have different:

* Database management systems
* Operating systems
* Hardware
* Communications protocols

Before you begin building your extract systems, you need a logical data map that documents the relationship between original source fields and final destination fields in the tables you deliver to the front room. This document ties the very beginning of the ETL system to the very end. We show you how to build your logical data map in [Part 1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/pt01.html) of this chapter.

[Part 2](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/pt02.html) of this chapter is a tour of the many flavors of source systems you are likely to encounter. We probe moderately deeply into each one to get you started choosing the right extraction approach.

At the end of this chapter, we introduce the subject of change data capture and deleted record capture. Fifteen years ago we thought that the data warehouse was immutable: a huge write-once library of data. With the benefit of lots of experience in the intervening years, we now know that data warehouses constantly need to be updated, corrected, and altered. The change data capture extraction techniques in this chapter are only the first step in this intricate dance. We need to revisit this subject in the data-cleaning chapter, the delivery chapters, and the operations chapter!

Let's dive into the logical data map.

Part 1: The Logical Data Map

The physical implementation can be a catastrophe if it is not carefully architected before it is implemented. Just as with any other form of construction, you must have a blueprint before you hit the first nail. Before you begin developing a single ETL process, make sure you have the appropriate documentation so the process complies logically and physically with your established ETL policies, procedures, and standards.

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release

Data Flow: *Extract* → Clean → Conform → *Deliver*

The logical data map describes the relationship between the extreme starting points and the extreme ending points of your ETL system.

Designing Logical Before Physical

Diving right into physical data mapping wastes precious time and excludes documentation. This section describes how to develop the logical ETL process and use it to map out your physical ETL implementation. Ensure the following steps are achieved before you start any physical ETL development:

1. **Have a plan**. The ETL process must be figured out logically and documented. The logical data map is provided by the data warehouse architect and is the specification for the ETL team to create the physical ETL jobs. This document is sometimes referred to as the data lineage report. The logical data map is the foundation of the metadata that is eventually presented to quality-assurance testers and ultimately to end users to describe exactly what is done between the ultimate source system and the data warehouse.
2. **Identify data source candidates**. Starting with the highest-level business objectives, identify the likely candidate data sources you believe will support the decisions needed by the business community. Identify within these sources specific data elements you believe are central to the end user data. These data elements are then the inputs to the data profiling step.
3. **Analyze source systems with a data-profiling tool**. Data in the source systems must be scrutinized for data quality, completeness, and fitness for the purpose. Depending on your organization, data quality might or might not fall under the responsibility of the ETL team, but this data-profiling step must be done by someone with an eye for the needs of the decision makers who will use the data warehouse. Data in each and every source system must be analyzed. Any detected data anomaly must be documented, and best efforts must be made to apply appropriate business rules to rectify data before it is loaded into the data warehouse. You must hold open the possibility that the project STOPs with this step! If the data cannot support the business objectives, this is the time to find that out. More on data profiling in [Chapter 4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html).
4. **Receive walk-though of data lineage and business rules**. Once the data sources have been qualified by the data-profiling step and the final target data model is understood, the data warehouse architect and business analyst must walk the ETL architect and developers through the data lineage and business rules for extracting, transforming, and loading the subject areas of the data warehouse, as best they understand these rules. Full understanding of the data lineage and business rules will not be achieved until the ETL team has encountered all the data realities, but this step aims to transfer as much knowledge as possible to the ETL team. The data-profiling step should have created two subcategories of ETL-specific business rules:

**4a**. Required alterations to the data during the data-cleaning steps

**4b**. Coercions to dimensional attributes and measured numerical facts to achieve standard conformance across separate data sources

1. **Receive walk-through of data warehouse data model**. The ETL team must completely understand the physical data model of the data warehouse. This understanding includes dimensional modeling concepts. Understanding the mappings on a table-by-table basis is not good enough. The development team must have a thorough understanding of how dimensions, facts, and other special tables in the dimensional model work together to implement successful ETL solutions. Remember that a principal goal of the ETL system is to deliver data in the most effective way to end user tools.
2. **Validate calculations and formulas**. Verify with end users any calculations specified in the data linage. This rule comes from the *measure twice, cut once* aphorism used in the construction business in New York City. Just as you don't want to be caught up on a skyscraper with the wrong-size material, you similarly don't want to be caught deploying the wrong measures in the data warehouse. It is helpful to make sure the calculations are correct before you spend time coding the wrong algorithms in your ETL process.

Inside the Logical Data Map

Before descending into the details of the various sources you will encounter, we need to explore the actual design of the logical data mapping document. The document contains the data definition for the data warehouse source systems throughout the enterprise, the target data warehouse data model, and the exact manipulation of the data required to transform it from its original format to that of its final destination.

Components of the Logical Data Map

The logical data map (see [Figure 3.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#the_logical_data_map_dot)) is usually presented in a table or spreadsheet format and includes the following specific components:

* **Target table name**. The physical name of the table as it appears in the data warehouse
* **Target column name**. The name of the column in the data warehouse table
* **Table type**. Indicates if the table is a fact, dimension, or subdimension (outrigger)
* **SCD (slowly changing dimension) type**. For dimensions, this component indicates a Type-1, −2, or −3 slowly changing dimension approach. This indicator can vary for each column in the dimension.

For example, within the customer dimension, the last name may require Type 2 behavior (retain history), while the first name may require Type 1 (overwrite). These SCD types are developed in detail in [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html).

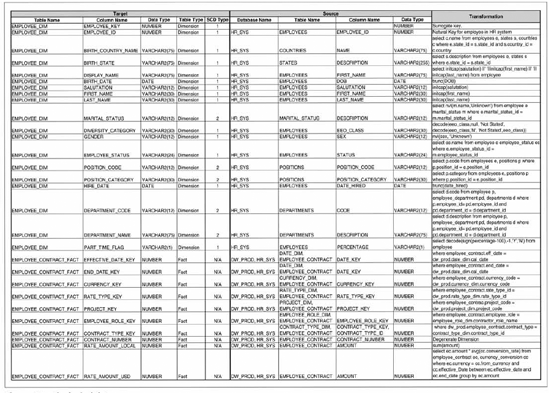
* **Source database**. The name of the instance of the database where the source data resides. This component is usually the connect string required to connect to the database. It can also be the name of a file as it appears in the file system. In this case, the path of the file would also be included.
* **Source table name**. The name of the table where the source data originates. There will be many cases where more than one table is required. In those cases, simply list all tables required to populate the relative table in the target data warehouse.
* **Source column name**. The column or columns necessary to populate the target. Simply list all of the columns required to load the target column. The associations of the source columns are documented in the transformation section.
* **Transformation**. The exact manipulation required of the source data so it corresponds to the expected format of the target. This component is usually notated in SQL or pseudo-code.

Columns in the logical data mapping document are sometimes combined. For example, the source database, table name, and column name could be combined into a single *source* column. The information within the concatenated column would be delimited with a period, for example, ORDERS.STATUS.STATUS CODE. Regardless of the format, the content of the logical data mapping document has been proven to be the critical element required to efficiently plan ETL processes.

The individual components in the logical data mapping appear to be simple and straight-forward. However, when studied more closely, the document reveals many hidden requirements for the ETL team that might otherwise have been overlooked. The primary purpose of this document is to provide the ETL developer with a clear-cut blueprint of exactly what is expected from the ETL process. This table must depict, without question, the course of action involved in the transformation process.

Take a look at [Figure 3.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#the_logical_data_map_dot).

Scrutinizing this figure, you may notice a few revelations that, if they were to go unnoticed, would cause a lot of time troubleshooting and debugging and ultimately delaying the project. For example, you might notice that the data types between the source and target for STATE get converted from 255 characters to 75 characters. Even though the data-scale reduction might be supported by the data-analysis documentation, should any future values with more than 75 characters be created, you would potentially lose the data. Moreover, some ETL tools would actually abort or fail the entire process with this kind of data overflow error. Notice the transformation notation for the STATE does not explicitly define this data conversion—the conversion is implied. By definition, no one explicitly accounts for *implied conversions*. Implied conversions are common and notorious for sneaking up and destroying your processes. To avoid calamity, the ETL team must assume responsibility for explicitly handling these types of implied data conversions.



**Figure 3.1. The logical data map.**

NOTE

ETL tool suites typically keep track of these implied data conversions and can deliver reports that identify any such conversions.

The table type gives us our queue for the ordinal position of our data load processes—first dimensions, then facts.

Working with the table type, the SCD type is crucial while loading dimensions. As we explain earlier in this chapter, the structure of the table itself does not reveal what the slowly changing dimension strategy is. Misinterpreting the SCD strategies could cause weeks of development time gone to waste. Know exactly which columns have historic relevance and the strategy required for capturing the history before you begin the development of the load process. The value in this column may change over time. Usually during unit testing, when your selected users observe the data in the data warehouse for the first time, they see unexpected results. As hard as the data modeler may try, the SCD concepts are very hard to convey to users, and once they are exposed to the loaded dimension, they quite often want to *tweak* the SCD strategies. This request is common and should be handled through the data warehouse project manager and the change management process.

The transformation within the mapping is the *guts* of the process, the place where developers with strong technical abilities look first. But you must constrain yourself from being completely code focused and review the entire mapping before you drill into the transformation. The transformation can contain anything from the absolute solution to nothing at all. Most often, the transformation can be expressed in SQL. The SQL may or may not be the complete statement. Quite often, it is the segment of the code that cannot otherwise be implied from the other elements in the mapping, such as the SQL WHERE clause. In other cases, the transformation might be a method that is not SQL specific and is explained in plain English, like instructions to preload from a flat file or to base the load transformation on criteria outside of the database or to reject known data anomalies into a reject file. If the transformation is blank, this means the mapping is a straight load, from source-to-target, with no transformation required.

NOTE

Upon the completion of the logical data map, do a comprehensive walkthrough of the document with the ETL developer before any actual coding begins.

Using Tools for the Logical Data Map

Some ETL and data-modeling tools directly capture logical data mapping information. There is a natural tendency to want to indicate the data mapping directly in these tools. Entering this information into a tool that enables us to share this metadata is a good practice. But, at the time of this writing, there is no standard for the appropriate data elements related to logical data mapping. The exact elements available in the various tools differ quite a bit. As the metadata standards in the data warehouse environment mature, a standard should be established for the elements defined in the logical data map. Established metadata standards will enable the tools to become more consistent and usable for this purpose. You should investigate the usability of your current toolset for storing the logical data map and take advantage of any features you have available. However, if your tools do not capture all of the elements you need, you will wind up having the logical data map in several locations, making maintenance a horrific chore. Be on the lookout for vast product improvements in this area.

Building the Logical Data Map

The success of data warehousing stems in large part from the fact that all data is in one logical place for users to perform cross-functional analysis. Behind the scenes, the ETL team integrates and transforms disparate, un-organized data seamlessly and presents it as if it has lived together since the beginning of time. A key criterion for the success of the data warehouse is the cleanliness and cohesiveness of the data within it. A unified data store requires a thorough insight of each of its source data systems. The importance of understanding the data in the data sources, and the systems of the sources themselves, is often overlooked and underestimated during the project-planning phase of the ETL. The complete logical data mapping cannot exist until the source systems have been identified and analyzed. The analysis of the source system is usually broken into two major phases:

* The data discovery phase
* The anomaly detection phase

Data Discovery Phase

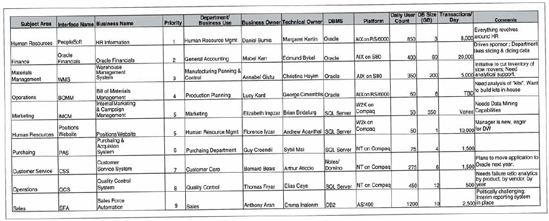
Once you understand what the target needs to look like, you need to identify and examine the data sources. Some or all of the source systems may have been accumulated during the data-modeling sessions, but this cannot be taken for granted. Usually, only the major source systems are identified during the data-modeling sessions. It's important to note that the data modeler's main objective is to create a data model. Any logical data mapping derived from the data-modeling sessions is merely a byproduct—a starting point. Moreover, the data modeler spends most of his or her time with end users, so the source systems defined in the logical data mapping may not be the true originating or optimal source—the *system-of-record*. It is up to the ETL team to drill down further into the data requirements to determine each and every source system, table, and attribute required to load the data warehouse. Determining the proper source, or system-of-record, for each element is a challenge that must be reckoned with. Thorough analysis can alleviate weeks of delays caused by developing the ETL process using the wrong source.

Collecting and Documenting Source Systems

The source systems are usually established in various pieces of documentation, including interview notes, reports, and the data modeler's logical data mapping. More investigation is usually necessary by the ETL team. Work with the team's system and business analysts to track down appropriate source systems. In large organizations, you must ask the question "Who else uses this data?" and find the data source of each user group. Typical organizations have countless distinct systems. It is the ETL team's responsibility to keep track of the systems discovered and investigate their usefulness as a data warehouse source.

Keeping Track of the Source Systems

Once the source systems are identified, it makes sense to document these systems along with who is responsible for them. [Figure 3.2](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#source_system_trackingreport_dot) is a chart created for this purpose. This chart, the source system tracking report, has saved us many times from having to hunt down system administrators or business owners. If you are lucky, the data modeler will have started this list. Regardless of the originator, the maintenance of the list should be a collaborative effort between the ETL team and the data modeling team. If during your analysis systems are deemed inappropriate as a source system to the data warehouse, leave them on the list with the reason for their omission; they may be used in future phases.



**Figure 3.2. Source system trackingreport.**

The source system tracking report also serves as an outline for future phases of the data warehouse. If there are 20 source systems identified in the list, and phase 1 includes two or three systems, plan to be on the project for a long, long time.

* **Subject area**. Typically the name of the data mart that this system feeds
* **Interface name**. The name of the transaction application that the source system supports
* **Business name**. The name the system is commonly referred to by the business users.
* **Priority**. A ranking or ordinal position used to determine future phases. The priority is usually set after the data warehouse bus matrix has been completed.
* **Department/Business use**. The primary department using the database, for example, Accounting, Human Resources, and so on. If the application is used by many departments, indicate the business use, for example, Inventory Control, Client tracking, and so on.
* **Business owner**. The person or group to contact for issues or questions related to the use of the application or database. This person or group is typically the data steward for the subject area.
* **Technical Owner**. Typically the DBA or IT project manager responsible for maintaining the database
* **DBMS**. The source database management system name. In most cases, it will be a relational database such as Oracle, DB2, or Sybase. It can also be nonrelational data stores like Lotus Notes or VSAM.
* **Production server/OS**. When known, this column includes the physical name of the server where the database lives. It also includes the operating system. You need this column when designing OS level scripts for your ETL. For example, you cannot use UNIX shell scripts when the server is operating on NT.
* **# Daily users**. Gives you an idea of how many operational people in the organization the data is exposed to. This number is not the same as the potential end user data warehouse users.
* **DB size**. The DBA should be able to provide this information. Knowing the raw size of the source data can help you determine the ETL priorities and efforts. Generally speaking, the larger databases tend to be higher on the priority lists because performance is usually lacking when large tables or several joined tables are queried in the transaction system.
* **DB complexity**. The number of tables and view objects in the system
* **# Transactions per day**. Estimate that gives you an indication of the capacity requirements for the incremental load process
* **Comments**. Usually used for general observations found while researching the database. It may include notes about future version releases of the database or reasons why it is or isn't a system-of-record for certain entities.

Determining the System-of-Record

Like a lot of the terminology in the data warehouse world, the system-of-record has many definitions—the variations depend on who you ask. Our definition of the system-of-record is quite simple: It is the originating source of data. This definition of the system-of-record is important because in most enterprises data is stored redundantly across many different systems. Enterprises do this to make nonintegrated systems share data. It is very common that the same piece of data is copied, moved, manipulated, transformed, altered, cleansed, or made corrupt throughout the enterprise, resulting in varying versions of the *same* data. In nearly all cases, data at the end of the lineage will not resemble the originating source of data—the system-of-record. We were once on a project where the originally identified source data was four times removed from the system-of-record. In the process between systems 3 and 4, the data was transferred via an algorithm in an attempt to *clean* the data. The algorithm had an undetected bug, and it corrupted the data by inserting data from other fields into it. The bug was discovered by the ETL team during the data-discovery phase of the project.

NOTE

Dealing with Derived Data.

You may run into some confusion surrounding derived data. Should the ETL process accept calculated columns in the source system as the system-of-record, or are the base elements, the foundation of the derived data, desired? The answer to this depends partly on whether the calculated columns are addditive. Nonadditive measures cannot be combined in queries by end users, whereas additive measures can. So you may be forced to use the base elements and calculate the nonadditve measure yourself. But be thoughtful. Should you try to recreate the calculations in the ETL process, you will be responsible for keeping the calculations synchronized and for understanding the business rules that define these calculations. If the calculation logic changes in the source system, the ETL process will have to be modified and redeployed. It is necessary, therefore, to capture the calculation as metadata so users understand how it was derived.

Unless there is substantial evidence that the originating data is not reliable, we recommend you don't sway from our definition of system-of-record. Keep in mind that a goal of the data warehouse is to be able to share conformed dimensions across all subject areas. Should you chose not to use the system-of-record to load your data warehouse, conforming dimensions will be nearly impossible. Should you need to augment your dimensions with different versions of data for specific needs, those should be stored as additional attributes in your conformed dimension.

However, to each rule there is an exception. Identifying the database name or file name may not be as easy as you might think—especially if you are dealing with legacy systems. During a project, we once spent weeks tracking down the *orders* database. Everyone we spoke to referred to the database by a different name. We then discovered that each location had a local version of the database. Since the goal of the data warehouse was to report across the organization, we began documenting each location database name with the intent to migrate the data with the ETL process. During our research of the database names, a programmer finally came forward and said, "You can get the list of the databases you need by reading this replication program." To our surprise, there was already a process in place to replicate the local databases to a central repository. Rather than recreate the wheel, we chose to use this program to get the consolidated database name and loaded the data warehouse from the central repository. Even though the true originating source of data was in each location's database, using the central repository was the most efficient and reliable solution.

NOTE

The further downstream you go from the originating data source, the more you increase the risk of extracting corrupt data. Barring rare exceptions, maintain the practice of sourcing data only from the system-of-record.

Analyzing the Source System: Using Findings from Data Profiling

Once you've determined the system-of-record, your next step is to analyze the source systems to get a better understanding of their content. This understanding is normally accomplished by acquiring the entity relation (ER) diagrams for the systems you've selected to be the system-of-record, if they are based on relational technology. Should the ER diagrams not exist (and don't be surprised if there are none to be found) you may be able to create them. ER diagrams can be generated by *reverse engineering* the database. Reverse engineering is a technique where you develop an ER diagram by reading the existing database metadata. Data-profiling tools are available that make this quite easy. Just about all of the standard data-modeling tools provide this feature, as do some of the major ETL tools.

NOTE

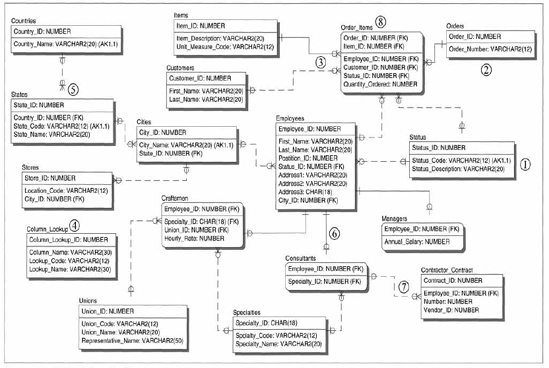
Reverse engineering a system of record to get a proper ER model of the data is obviously useful. But it is not the same as forward engineering a complex ER model to build simple dimensional schemas. Data-modeling tools, in fact, fail miserably at this kind of forward engineering when you are trying to see the forest for the trees in a normalized environment with hundreds of tables.

Before diving into the ER diagram, look for a high-level description of the tables and fields in the database. If it exists, it may take the form of unstructured text descriptions, and it may be out of date, but it's far better to start with an overview than to try to discover the overview by looking at reams of mind-numbing detail. Also, don't forget to debrief the source-system guru who understands all the arcane logic and incremental changes that have occurred in the source system!

Having the ability to navigate an ER diagram is essential to performing data analysis. All members of the ETL team need to be able to read an ER diagram and instantly recognize entity relationships. [Figure 3.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#entity_relationship_diagram_dot) illustrates a simple ER diagram.

In the numbered list that follows, we explain the significant characteristics that you want to discover during this phase, including unique identifiers, nullability, and data types. These are primary outputs of a data-profiling effort. But more important, we explain how to identify when tables are related to each other; and which columns have dependencies across tables. Specific characteristics in the ER diagram outlined in [Figure 3.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#entity_relationship_diagram_dot) are:

1. **Unique identifiers and natural keys**. Unique identifiers depict the columns that uniquely represent a row in a table. This definition can be misleading, so we want to investigate it a bit further. From a referential integrity standpoint, a unique identifier is the primary key for a table. Most of the time, the primary key is artificial, and although it is unique from an ETL standpoint, it is not enough information to determine if the row is unique. In every properly designed transaction table, in addition to the primary key, there is at least one natural key. The natural key is what the business uses to uniquely describe the row. For example, a status table can have a status id, status code, and status description. The status id is clearly the primary key, but depending on the business rules, for purposes of the ETL, the status code could be the unique identifier natural key. Special care must be taken when selecting the correct natural keys, especially when loading slowly changing dimensions.
2. **Data types**. Remember, as ETL analysts, you take nothing for granted. Column names do not infer data types. Just because a column is named Purchase Order Number, are we certain that only numbers, not letters, are stored in the column? Additionally, if there are only numbers, are there leading zeros? Are these zeros important to end users? On a particular project, while building a human resource data mart, we had a source column named SCORE; it was a CHAR(2) data type. In the target, it was a NUMBER data type. As it turns out, the data was approximately 80-percent numbers and the rest were letters (A-D and F). The ETL process needed to convert any letter grades to their numerical equivalent. DATE and TIME elements are notorious for being stored as text. It is up to the ETL process to convert these dates while loading them into the data warehouse.



**Figure 3.3. Entity relationship diagram.**

1. **Relationships between tables**. Understanding how tables are related is vital to ensuring accuracy in joins while retrieving data. If you are lucky, the ER diagram has lines connecting the related tables. Evaluation of table relationships includes analyzing the connecting lines. Unfortunately, data-processing people are not lucky and it is most likely that you'll need to look at the diagram a bit closer to determine table relationships. While loading a target table from heterogeneous sources, it is good practice to bring all of the sources into a data-modeling tool and map out the relationships. This integrated ER diagram lends itself to making the logical data map easier to create.
2. **Discrete relationships**. It is not uncommon for the design of the source system to include a single look-up table that stores all of the static reference data for all of the tables throughout the database. The look-up table contains a column that identifies which table and column the associated group of rows support. This takes time for the unknowing to discover. Carefully document the name of each group of rows and the associated tables and columns. This information will be needed while mapping many of the dimensions.
3. **Cardinality of relationships and columns**. Knowing the cardinality of relationships is necessary to predict the result of your queries. Using crow's feet notation, a single line means the cardinality is 1, and only 1 of the same value is allowed. A line and a circle indicate zero or 1 is allowed. The side with 3 lines in the form of a crow's foot indicates the same value can be repeated many times. In relational databases, all associated tables will be joined in one of the following ways:
   * **One-to-one**. You see one-to-one relationships during super-type/sub-type scenarios and the practice of vertical table partitioning. One-to-one relationships can be identified by observing that the relationship is on the primary key of each table.
   * **One-to-many**. This is the most commonly found relationship for foreign key references. It is easily identified by noting that a nonkey attribute in a table refers to the primary key of another table. We call this nonkey attribute a foreign key, and we insist that all the foreign keys are *good*, that is, they are instances of the primary key they point to.
   * **Many-to-many**. This relationship usually involves three tables with two one-to-many relationships between them. More specifically, there are two tables with an *associative* table between them. The center or associative table has a compound primary key and two foreign keys, one to the primary key of one table and another to the primary key of the other table.

NOTE

Frequently, source systems do not have foreign keys or referential integrity consistently defined in the database dictionary. These issues may also be discovered through simple column-name matching and more comprehensive data profiling.

Be sure you carefully study all data types in your sources, in your intermediate staging tables, and in the final tables to be delivered. It's quite common for the data modeling team to create data elements that don't exactly match their source. In some cases, you'll find data types are purposely mismatched. For example, some designers deliberately make all code fields allow alphanumeric characters, even if the current system uses only numbers and is a number data type. Also, be sure to evaluate the length of each field. In some cases, the target data warehouse can have smaller data lengths or numeric precision than the source database. Smaller data lengths in the target causes data truncation (lost data!). When you see disparities, check with the data modeling team to confirm their intentions. Either await database corrections or get business rules for conversion and truncation routines.

Data Content Analysis

Understanding the content of the data is crucial for determining the best approach for retrieval. Usually, it's not until you start working with the data that you come to realize the anomalies that exist within it. Common anomalies that you should be aware of include:

* **NULL values**. An unhandled NULL value can destroy any ETL process. NULL values pose the biggest risk when they are in foreign key columns. Joining two or more tables based on a column that contains NULL values will cause data loss! Remember, in a relational database NULL is not equal to NULL. That is why those joins fail. Check for NULL values in every foreign key in the source database. When NULL values are present, you must *outer* join the tables. An outer join returns all of the rows regardless of whether there is a matching value in the joined table. If the NULL data is not in a foreign key column but is in a column required by the business for the data warehouse, you must get a business rule on how the NULL data should be handled. We don't like to store NULL values in the data warehouse unless it is indeed an unknown measure. Whenever possible, create default values to replace NULL values while loading the data warehouse.
* **Dates in nondate fields**. Dates are very peculiar elements because they are the only logical elements that can come in various formats, literally containing different values and having the exact same meaning. Fortunately, most database systems support most of the various formats for display purposes but store them in a single standard format (for that specific database). But there are many situations where dates are stored in text fields, especially in legacy applications. The possible variations of date formats in nondate fields are boundless. Following is a sample of the possible variations of the same date that can be found when the date is stored in a text field:

13-JAN-02

January 13, 2002

01-13-2002

13/01/2002

01/13/2002 2:24 PM

01/13/2002 14:24:49

20020113

200201

012002

As you can imagine, it's possible to fill pages with variations of a single date. The problem is that when the source database system doesn't control or regulate the data entry of dates, you have to pay extra-close attention to ensure you are actually getting what you expect.

NOTE

In spite of the most detailed analysis, we recommend using outer join logic when extracting from relational source systems, simply because referential integrity often cannot be trusted on remote systems.

Collecting Business Rules in the ETL Process

You might think at this stage in the process that all of the business rules must have been collected. How could the data modelers create the data model without knowing all of the business rules, right? Wrong. The business rules required for the data modeling team are quite different from those required by the ETL team. For example, the data modeling definition of the status dimension might be something like:

**Status Code**—The status is a four-digit code that uniquely identifies the status of the product. The code has a short description, usually one word, and a long description, usually one sentence.

Conversely, the ETL definition of status might be expressed like this:

**Status Code**—The status is a four-digit code. However, there are legacy codes that were only three digits that are still being used in some cases. All three-digit codes must be converted to their four-digit equivalent code. The name of the code may have the word *OBSOLETE* embedded in the name. *OBSOLETE* needs to be removed from the name and these obsolete codes must have an obsolete flag set to 'Y'. The description should always be in sentence case, regardless of the case used when entered into the source system.

The business rules for the ETL process are much more technical than any other collection of business rules in the data warehouse project. Regardless of their technical appearance, these rules still stem from the business—the ETL team cannot be in the business of making up rules. It is up to the ETL architect to translate user requirements into usable ETL definitions and to articulate these technical definitions to the business people in a way they can understand. The ETL data definitions go through an evolution process. As you discover undocumented data anomalies, document and discuss them with the business—only they can dictate how the anomalies should be handled. Any transformations that come from these meetings have to be documented, properly approved, and signed off.

Integrating Heterogeneous Data Sources

The preceding sections of this chapter expose many of the common data systems that you might come across while sourcing your data warehouse. This section discusses the challenges you may face *integrating* the different data sources. But before you can integrate data, you need to know what *data integration* means. Integrating data means much more than simply collecting disparate data sources and storing that data in a single repository. To better understand what integration really means, consider a corporate merger. During corporate mergers, one or more companies are joined with other similar (or dissimilar) companies. When mergers occur, the business must decide which company is the surviving company and which gets consumed by the new parent. Sometimes, negotiations are made when the parent company recognizes value in certain practices and techniques of its subsidiaries and incorporates those practices in its modified organization. The result of a successful corporate merger is a cohesive organization that has a single business interest. This requires a heroic commitment to aligning terminology (dimension attributes) and aligning key performance indicators (facts in fact tables). If you think of integrating your data in the same fashion as a corporate merger, the result of your data warehouse is a single source of information organized to support the business interest.

But what about those half-finished mergers that allow their subsidiary companies to *do their own thing*? This situation causes a problem because the companies are not integrated—they are merely associated. When you build a data warehouse, integration can occur in several places. The most direct form of data integration is the implementation of conformed dimensions. In the data warehouse, conformed dimensions are the cohesive design that unifies disparate data systems scattered throughout the enterprise.

NOTE

When a dimension is populated by several distinct systems, it is important to include the unique identifier from each of those systems in the target dimension in the data warehouse. Those identifiers should be viewable by end users to ensure peace of mind that the dimension reflects their data that they can tie back to in their transaction system.

What happens when those idiosyncratic dimensions cannot completely conform? Unfortunately, this question is as much of a political issue as a technical one. Conformed dimensions and facts are crucial to the success of the data warehouse project. If the result of your project offers disparate dimensions that are not cohesive across business subject areas, you have not accomplished your goal. [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html)discusses loading dimensions in painstaking detail, but we want to mention specific techniques for loading conformed dimensions in a disparate source system environment here.

1. **Identify the source systems**. During the data-profiling phase of the construction of the logical data mapping, the data warehouse team must work together to detect the various sources of your target dimensions and facts. The data warehouse architect should uncover most of the potential sources of each element in the data warehouse and attempt to appoint a system-of-record to each element. The system-of-record is considered the *ultimate* source for the data being loaded.
2. **Understand the source systems (data profiling)**. Once the source systems are identified, you must perform a thorough analysis of each system. This is also part of data profiling. Data analysis of the source systems uncovers unexpected data anomalies and data-quality issues. This phase declares the reliability of the source system for the elements under scrutiny. During this phase of the project, the assignment of the system-of-record can actually be reassigned if data-quality issues persist or if reliability of the data is problematic for any reason.
3. **Create record matching logic**. Once you understand all of the attributes of all of the entities of all of the systems under consideration (quite a tall order), your next objective is to design the matching algorithm to enable entities across the disparate systems to be joined. Sometimes, the matching algorithm is as simple as identifying the primary key of the various customer tables. But in many cases, disparate systems do not share primary keys. Therefore, you must join the tables based on *fuzzy* logic. Perhaps, there is a social security number that can be used to uniquely identify your customer or maybe you need to combine the last name, e-mail address, and telephone number. Our intention here is not to offer a matching solution but to get you thinking about how your customers can be linked. The various business areas must be involved with and approve of your final matching logic. Don't forget to make sure that this record-matching logic is consistent with the various legislated privacy rules, such as HIPAA in the health care arena.
4. **Establish survivorship rules**. Once your system-of-record has been identified and the matching logic has been approved, you can establish the *surviving record* when data collisions occur in your ETL process. This means that if you have a customer table in your accounts receivable, production control, and sales systems, the business must decide which system has overriding power when attributes overlap.
5. **Establish nonkey attribute business rules**. Remember, dimensions (and facts) are typically sourced from various tables and columns within a system. Moreover, many source systems can, and usually do, contain different attributes that ultimately feed into a target final dimension. For instance, a list of departments probably originated in your HR department; however, the accounting code for that department probably comes from your financial system. Even though the HR system may be the system-of-record, certain attributes may be deemed more reliable from other systems. Assigning business rules for nonkey attributes is especially important when attributes exist in several systems but not in the system-of-record. In those cases, documentation and publication of the data lineage metadata is crucial to prevent doubt in the integrity of the data warehouse when users don't see what they expect to see.
6. **Load conformed dimension**. The final task of the data-integration process is to physically load the conformed dimension. This step is where you consider the slowly changing dimension (SCD) type and update late-arriving data as necessary. Consult [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) for details on loading your conformed dimensions.

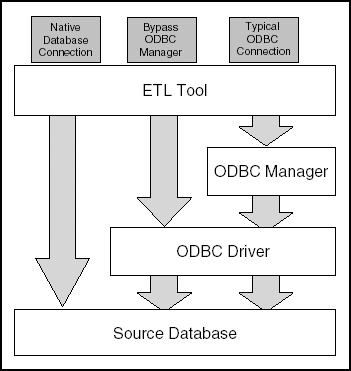
The beauty of your data warehouse is that it has the ability to truly integrate data, yet also to enable users to see dimensions from their perspective. Conformed dimensions and facts are the backbone of the enterprise data warehouse.

Part 2: The Challenge of Extracting from Disparate Platforms

Each data source can be in a different DBMS and also a different platform. Databases and operating systems, especially legacy and proprietary ones, may require different procedure languages to communicate with their data. On enterprise-wide data warehouse projects, be prepared to have communication with source systems limited to specific languages. Even if there is no technical limitation, departments or subsystems can, and usually do, have a standard language that is *allowed* to interact with their data. Standards we've been asked to use include COBOL, FOCUS, EasyTrieve, PL/SQL, Transact-SQL, and RPG. When a specific language beyond the realm of your ETL toolset or experience becomes mandatory, request that the owner of the source system extract the data into a flat file format.

Connecting to Diverse Sources through ODBC

Open Database Connectivity (ODBC) was created to enable users to access databases from their Windows applications. The original intention for ODBC was to make applications *portable*, meaning that if an application's underlying database changed—say from DB2 to Oracle—the application layer did not need to be recoded and compiled to accommodate the change. Instead, you simply change the ODBC driver, which is transparent to the application. You can obtain ODBC drivers for practically every DBMS in existence on virtually any platform. You can also use ODBC to access flat files.



**Figure 3.4. The topology of ODBC in the ETL process**

The drawback to ODBC's flexibility is that it comes at a performance cost. ODBC adds several layers of processing and passing of data to the data-manipulation process. For the ETL process to utilize data via ODBC, two layers are added between the ETL system and the underlying database. [Figure 3.4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#the_topology_of_odbc_in_the_etl_process)illustrates the layers involved in an ODBC environment.

* **ODBC manager**. The ODBC manager is a program that accepts SQL from the ETL application and routes it to the appropriate ODBC driver. It also maintains the connection between the application and the ODBC driver.
* **ODBC driver**. The ODBC driver is the real workhorse in the ODBC environment. The ODBC driver translates ODBC SQL to the native SQL of the underlying database.

As you might suspect, once you use ODBC you might lose much DBMS-specific functionality. Particular non-ANSI standard SQL commands are not accepted by the ODBC manager because it needs to maintain an *open* solution. ODBC, particularly Microsoft's OLE DB and .Net providers, have improved significantly in recent years, but for highest performance and native DBMS functionality you should look first to a native database driver. Just don't throw the baby out with the bath water. ODBC can provide a common format gateway to certain troublesome data sources that are otherwise not easily extracted.

Mainframe Sources

The mainframe computer, created in the mid 1960s, is widely used by most large enterprises around the world. The unique differentiator between mainframes and other computers is the hardware architecture. Nonmain-frame, including minicomputers and microcomputers, use their central processing units (CPUs) for virtually all of their processing, including getting data to and from disk and other peripherals. By contrast, mainframes have a special architecture emphasizing peripheral channels that process all input/output, leaving the CPU dedicated to processing only data, such as calculating formulas and balances.

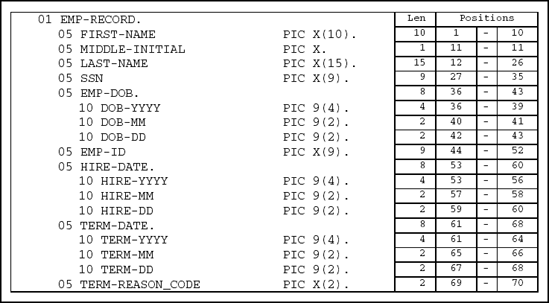
In many large companies, much of the day-to-day business data is processed and stored on mainframe systems (and certain minicomputer systems, such as the IBM AS/400) and integrating data from these systems into the data warehouse involves some unique challenges. There are several characteristics of mainframe systems that the ETL team must be familiar with and develop techniques to handle:

* COBOL copybooks
* EBCDIC character sets
* Numeric data
* Redefines fields
* Packed decimal fields
* Multiple OCCURS fields
* Multiple record types
* Variable record lengths

The rest of this section discusses these mainframe characteristics and offers techniques for managing them when they are encountered.

Working with COBOL Copybooks

COBOL remains the dominant programming language used on mainframe computers, and the file layout for data is described in COBOL copybooks. A copybook defines the field names and associated data types for a mainframe data file. As with other flat files you encounter in your ETL process, only two data types exist in mainframe flat files: text and numeric. However, numeric values are stored in a variety of ways that you need to understand to accurately process. Likewise, dates are stored simply as strings of numbers (or text) and typically require transformation to be stored in date columns in the data warehouse. [Figure 3.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#a_simple_copybook_that_describes_an_empl) illustrates a 70-byte, fixed length record that describes a simple employee record. Notice that the field names are preceded by level numbers. Nesting of level numbers is used to group related fields. COBOL programs can refer to field names at any defined level. For example, a program can refer to *HIRE-DATE* to capture the full date of hire or *HIRE-YYYY* if only the year portion is needed for processing.



**Figure 3.5. A simple copybook that describes an employee record**

Text and numeric data types are denoted using the PIC clauses. PIC X denotes text fields, while PIC 9 means the field is numeric. Field lengths are specified with numbers following the type. For example, the clause PIC 9(4) indicates a four-byte numeric field, whereas PIC X(15) indicates a 15-byte text field. PIC clauses can be coded alternatively by repeating the *X* or *9* data type indicator, such as *PIC 9999* for a four-byte numeric field.

The data file represented in [Figure 3.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#a_simple_copybook_that_describes_an_empl) can easily be transmitted via FTP and loaded into the data warehouse because all of the data is contained in *display* format. But before you try to transfer this file from the mainframe to the data warehouse platform, you need to take a short lesson on the difference between the familiar ASCII character set used on UNIX and Windows platforms and the EBCDIC character set used on the mainframe.

EBCDIC Character Set

Both the legacy mainframe systems and the UNIX- and Windows-based systems, where most data warehouses reside, are stored as *bits and bytes*. Each byte is made of eight bits, and each bit represents a binary (base-2) digit. The maximum number that can be represented by a byte made of binary bits is 255 (that is, 28−1). Thus, the number of unique characters (for example, A–Z, a–z, 0–9, punctuation, and special characters) that can be portrayed in a system made up of such bytes is 256 (including character 0).

Converting EBCDIC to ASCII

You might think that since both systems use bits and bytes, data from your mainframe system is readily usable on your UNIX or Windows system. But UNIX and Windows systems use the American Standard Code for Information Interchange (ASCII) character set, whereas mainframes use a different set, known as Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC uses more or less the same characters as ASCII but uses different 8-bit combinations to represent them.

For example, take the lowercase letter *a*. In ASCII, the letter *a* is character number 97 (01100001), but in EBCDIC, character number 97 is */* (forward slash). In EBCDIC *a* is character 129 (10000001). In fact, none of the common characters are represented by the same character numbers in ASCII and EBCDIC. To use mainframe data on your UNIX or Windows system, you must first translate it from EBCDIC to ASCII.

Transferring Data between Platforms

Luckily, translating data from EBCDIC to ASCII is quite simple. In fact it's virtually automatic, assuming you use File Transfer Protocol (FTP) to transfer the data from the mainframe to your data warehouse platform. An FTP connection requires two nodes—a host and a client. When an FTP connection is made between systems, the FTP client identifies its operating system environment to the FTP host, and the host determines whether any translation is required when transferring data between the two systems. So when an FTP connection is made between a mainframe and a UNIX or Windows system, the FTP host translates mainframe data from EBCDIC to ASCII as it transfers the data. In addition, FTP adds the special line feed and carriage return characters used to designate the end of a line (or record) of data on UNIX and Windows. FTP also translates from ASCII to EBCDIC if the data movement is from UNIX or Windows to the mainframe.

If you receive mainframe data on magnetic tape cartridge or CD-ROM rather than via FTP, you need to explicitly translate the data from EBCDIC to ASCII on the data warehouse system. This translation can be performed using the UNIX dd command with the conv=ascii switch. For Windows, you can obtain a port of the dd—and many other useful UNIX commands—on the Internet. In addition, commercial products that handle character-translation duties are available. ETL tool suites all handle this conversion. Most robust tools designed specifically for ETL can convert EBCDIC to ASCII on the fly.

NOTE

If your source data resides on a mainframe system, it is crucial that your ETL tool have the ability to implicitly convert EBCDIC data to ASCII. If at all possible, you want this to occur on the mainframe to avoid any corruption of low values and packed decimals. If data is received via tape or other media, the translation must occur by the ETL tool in the nonmainframe environment. At a minimum, the ETL tool must automatically execute FTP and process files *in stream*, passing the data directly from the mainframe through the ETL process to the target data warehouse.

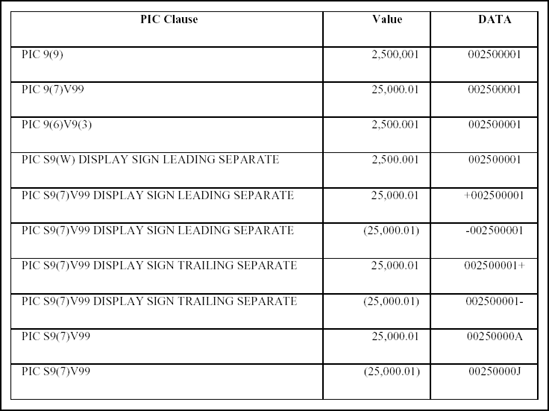
As a final point, although mainframes and UNIX or Windows systems use different character sets, translating data from one system to another is a rather simple task—simple, that is, unless your mainframe data has some other traits that are typical of the mainframe world. The next few sections discuss specific characteristics that mainframe data may possess and recommend strategies for managing them during the ETL process.

Handling Mainframe Numeric Data

When you begin to work with quantitative data elements, such as dollar amounts, counts, and balances, you can see that there's more to these numbers than meets the eye. For one thing, you won't typically find decimal points in decimal data, because the decimal points are *implied*. For example, the value 25,000.01 is stored as 002500001. Worse, the value 2,500,001 is stored the same way. So how does the mainframe COBOL program know that 25,000.01 is meant rather than 2,500,001? It's in the PIC clause. The next section discusses the importance and power of the PIC clause in COBOL copybooks.

Using PICtures

You can see in [Figure 3.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#the_pic_clause_in_a_cobol_copybook_indic) that the PIC clause can give the same data value different meaning. To accurately process a numeric value that comes from a legacy mainframe system, you must first transform it to its display format before transmitting it to the data warehouse system; otherwise, your ETL tool has to handle interpreting these mainframe values on the UNIX or Windows platform. To resolve decimals in the numeric values, you might think that you can simply divide the numeric value by the power of ten equal to the number of implied decimal places. And if all numeric values were stored with only the decimal point implied, you'd be right. However, it's not quite that simple. You also have to consider signed numeric values.



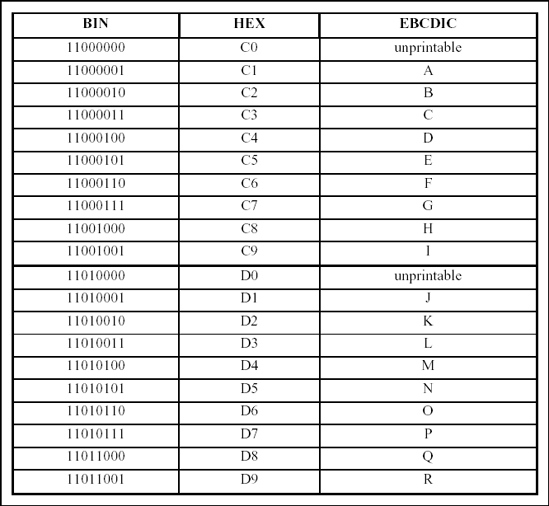
**Figure 3.6. The PIC clause in a COBOL copybook indicates the decimal places of a numeric value**

In mainframe data, the signs may come before or after the numeric value. What's more, the sign may be embedded within the numeric value.

The most common format, *zoned numeric*, embeds the sign within the last numeric digit as shown in the last two rows of [Figure 3.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#the_pic_clause_in_a_cobol_copybook_indic). So, how does *A* in the last position connote both the digit *1* and the sign +, and likewise, how does *J* represent both *1* and -? The trick is that the last byte is treated as two separate half-bytes (each containing four bits) and each half-byte is interpreted separately—in hexadecimal, of course!

For positive numbers, the first half-byte is set to *C*, the hexadecimal value of 1100, and negative numbers are set to *D*, the hexadecimal value of 1101. The second half-byte is set to the hexadecimal value that corresponds to the desired numeric digit. When you combine the first half-byte—1100 for positive or 1101 for negative—to the second half-byte, you get resulting EBCDIC characters, as seen in [Figure 3.7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#hexadecimal_to_ebcdic).

By now, you are probably scratching your head trying to figure out how to deal with numeric data from your mainframe system. Well, before you try to solve the problem, there's still one more twist that you are likely to encounter in most legacy mainframe systems.



**Figure 3.7. Hexadecimal to EBCDIC**

Unpacking Packed Decimals

Though at present computer hard disk storage is relatively inexpensive, in the past disk storage was among the most expensive components of the computer system. To save disk space, software engineers devised creative formats to store numeric data using fewer bytes than the digits in the number. The most pervasive of these formats is COMP-3, also known as *packed numeric*.

In many mainframe systems, most if not all numeric data is stored in COMP-3 format. COMP-3 format is a simple space-saving technique that uses half-bytes—or *nibbles*—rather than full bytes to store numeric digits. Each numeric digit can be stored in binary format within the four bits of a nibble. The last nibble of a COMP-3 numeric field stores the sign (positive/negative) of the numeric value. Using half-bytes to store numeric digits saves nearly half the space used by the display format. But this simple space-saving technique throws a wrench into the EBCDIC to ASCII character-set translation.

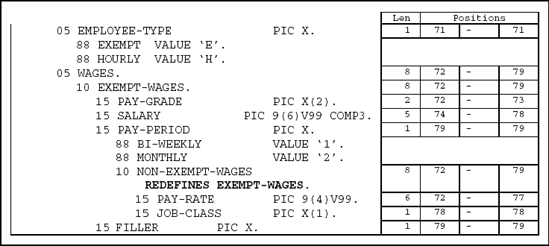
As a result of this translation conundrum, mainframe data that contains numeric values stored using numeric storage formats such as Zoned Numeric or COMP-3 (not to mention COMP, COMP-1, and COMP-2) cannot simply be translated from EBCDIC to ASCII and then processed on the UNIX or Windows warehouse system.

One of the following techniques must be used to maintain the integrity of mainframe numeric data:

* Reformat data on the mainframe into its display format before transmitting it to the data warehouse system using a simple program written in COBOL, Assembler, or a fourth-generation language such as SAS, Easytrieve, or FOCUS. Once data is reformatted in this way, it can then be translated to ASCII via FTP as described earlier in this chapter.
* Transfer data to the warehouse system in its native EBCDIC format. This option is viable only if your ETL tools or process can process EBCDIC data. Several types of tools can perform this task.
  + Use robust ETL tools that can process native EBCDIC, including accurately handling numeric data store in any mainframe-type numeric formats.
  + Use a utility program that can reformat data into *display* format on the warehouse platform. If you receive EBCDIC data and are writing the ETL process without the aid of a specialized ETL tool, we strongly recommend purchasing a utility program that can perform the numeric format conversion and EBCDIC-to-ASCII translation duties. Some relatively inexpensive, commercially available programs handle this task quite well.

Working with Redefined Fields

Rather than wasting space—remember it used to be expensive—mainframe engineers devised REDEFINES, which allow mutually exclusive data elements to occupy the same physical space. [Figure 3.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#redefines_clause_in_a_cobol_copybook) contains an excerpt from a COBOL Copybook that helps illustrate the concept of REDEFINES in mainframe data files. The excerpt describes the data fields that represent an employee's wage information. Notice EMPLOYEE-TYPE, which is a one-byte code that indicates whether the employee is exempt or hourly. Also, notice that two separate series of fields carry the wage information for the employee. The field set used depends on whether the employee is exempt or hourly. Exempt employees' wages are represented in three fields (PAY-GRADE, SALARY, and PAY-PERIOD), which take up a total of eight bytes. Hourly employees use a different set of fields that take up seven bytes (PAY-RATE and JOB-CLASS).



**Figure 3.8. REDEFINES clause in a COBOL copybook**

Since an employee is exempt or hourly, never both, only one of the two field sets is ever used at a time. The exempt wage fields occupy positions 72 through 79 in the file, and the hourly wage fields occupy positions 72 though 78. Furthermore, notice that the fields for exempt and hourly wages use different data types even though they occupy the same positions. When reading the employee record, the program must determine how to interpret these positions based on the value of EMPLOYEE-TYPE in position 71.

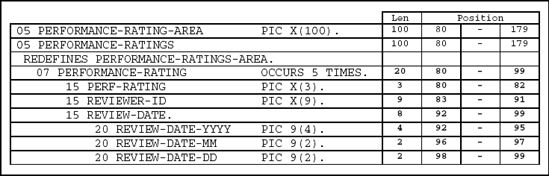
The same positions can have more than one REDEFINES associated with them, so rather than just two possible uses, the same positions can have two, three, or more possible uses. REDEFINES introduce one further complication that renders mere EBCDIC-to-ASCII character-set translation insufficient.

NOTE

When you encounter multiple REDEFINES in your sources, you should consider making each definition a separate pass of the extract logic over the source data if the subsequent processing is quite different (using Exempt versus Hourly as an example). This would allow you to build separate code lines for each extract rather than one complex job with numerous tests for the two conditions.

Multiple OCCURS

Mainframe and COBOL precede relational databases and Edward Codd's normalization rules. Prior to utilizing relational theory to design databases, repeating groups were handled with mainframe COBOL programs that use an OCCURS clause to define data fields that repeat within a data file. For example, in [Figure 3.9](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#cobol_copybook_with_occurs_clause_to_def) you can see an area of an employee record that stores information about performance ratings. The record is designed to keep track of up to five performance ratings. But rather than creating the needed fields five times—remember, this precedes relational theory so there won't be a separate performance rating table with a foreign key that points back to the employee—they are named only once within a special OCCURS field. The OCCURS clause indicates the number of times the fields within it repeat. Essentially, the OCCURS clause defines an array contained within the file. Thus, in the employee record, data for the first performance rating occupies positions 80 to 99, the second rating from 100 to 119, the third from 120 to 139, the fourth from 140 to 159, and the fifth—and last—from 160 to 179.



**Figure 3.9. COBOL copybook with OCCURS clause to define repeating groups within a data record.**

In most cases, the ETL process needs to *normalize* any data contained in a OCCURS section of a mainframe file. Even though it is possible to manually program the ETL process to manage the repeating data, it is strongly recommended that you use a robust ETL tool that allows you to use the COBOL copybooks to define inputs or at least allows you to manually define input file arrays in some other way. If your tools do not support input arrays, you are stuck with the toil of writing code to deal with repeating groups within records sourced from your legacy mainframe systems.

NOTE

Sometimes programmers use OCCURS to store different facts in an array, rather than storing the same fact N times. For example, suppose O-DATE occurs four times. The first date is CREATE, the second is SHIP, the third is ACKNOWLEDGE, and the fourth is PAYMENT. So in this case you don't normalize this OCCURS data but rather create discrete fields for each position in the array.

To ensure data integrity, model data that results from a COBOL OCCURS clause in a normalized fashion—a master table and child table—in the staging area of the data warehouse. It's good practice to stage this data in separate tables because the result of the process most likely loads data into a fact table and a dimension, two separate dimensions, or two separate fact tables. We find that in these situations it makes sense to set data down to *settle* before integrating it with the data warehouse.

Managing Multiple Mainframe Record Type Files

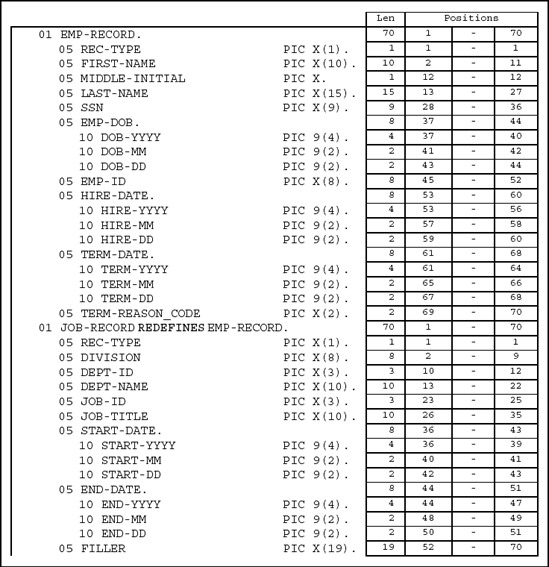
The concept of multiple record type files is touched upon in the section that discusses REDEFINES. The main difference between REDEFINES as discussed earlier and what we're introducing now is that instead of having just a small portion of a record contain multiple definitions, the entire record has multiple definitions. Multiple record types are often used to span a single logical record across two or more physical records. [Figure 3.10](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#recognizing_multiple_record_types_within) contains an extract of a COBOL copybook that illustrates the concept of redefining an entire record.

In [Figure 3.10](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#recognizing_multiple_record_types_within), the REDEFINES clause applies to the entire record. So now, instead of the file carrying only an employee's basic information, it also carries an employee's job history with the company as well. In this file, every employee has at least two records: one EMP-RECORD and one JOB-RECORD. When an employee transfers to a new job, a new JOB-RECORD is added to the file. So an employee's total job history is contained in two or more records on the file: one EMP-RECORD and one or more JOBRECORD(s).

In this file, the physical order of the records is critically important because the JOB-RECORDs do not have any information to link them to their corresponding EMP-RECORDs. The JOB-RECORDs for an employee follow immediately after his or her EMP-RECORD. So to accurately process the job history of an employee, you must treat two or more physically adjacent records as one *logical* record.

The benefit of using multiple record types is—once again—to save space. The alternative, without using relational theory, is to have extremely wide, space-wasting records to carry all data in a single record. If, for example, you want to track job history for up to five prior positions, you have to add 255 bytes to each employee record (the base EMP-RECORD, plus five occur-rences of JOB-RECORD fields (5 × 51 bytes). But the number of job history field segments is *situational*—it depends on how many jobs an employee has held.

By using multiple record types, the mainframe system can store job history records only as needed, so employees with only one job require only one JOB-RECORD (70 bytes including FILLER), saving 185 bytes on the file. Furthermore, you are no longer limited to a fixed number of jobs in file. An unlimited number of 70-byte JOB-RECORDs can be added for each employee.



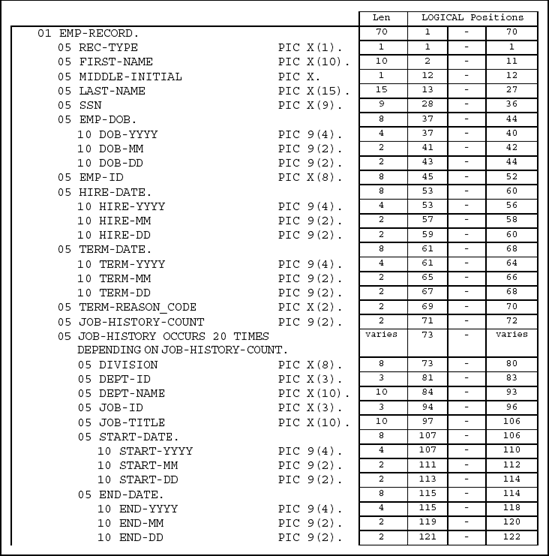
**Figure 3.10. Recognizing multiple record types within the same file**

Our employee example has only two record types, but multiple REDEFINES can be used to create any number of record types that can combine into a single logical record. If we expand the employee example, you might imagine a third record type to carry information about the employee's fringe benefits and a fourth type to carry information about the employee's family dependents.

The ETL process must manage multiple record types by retaining the values from the first physical record in a set—which is only the first part of the logical record—in memory variables so they can be joined to the rest of the data that follows in subsequent records.

Handling Mainframe Variable Record Lengths

In the previous section, we discuss how information related to a single entity is spanned across two or more records using multiple record types. A variable record length is another approach used in mainframe files to store situational information. Rather than storing the job history of an employee in separate JOB-RECORDs, each job is stored in an OCCURS job history segment. Furthermore, as illustrated in [Figure 3.11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#variable_record_lengths_in_a_cobol_copyb), instead of the record having a fixed number of such segments, the number of segments varies between 0 and 20, based on the numeric value in the DEPENDING ON JOB-HISTORY-COUNT clause. Each additional employee job, up to a maximum of 20, adds 50 bytes to the length of the record.



**Figure 3.11. Variable record lengths in a COBOL copybook using the DEPENDING ON clause.**

Variable-length records that use DEPENDING ON clauses in the copy-book make straightforward EBCDIC-to-ASCII character-set translation ineffective. The following is a list of ways to mitigate the risk of creating corrupt data from variable-length records during the ETL process.

* Convert all data to *display* format on the mainframe and convert to fixed-length records, adding space at the end of each record for all unused variable segment occurrences.
* Transfer the file in BINARY format to the data warehouse platform. This technique requires having tools that can interpret all of the nuances of mainframe data discussed throughout this chapter. Robust dedicated ETL tools handle most or all of these situations. If your data warehouse project does not include such a tool, third-party utility programs are available in various price ranges that can interpret and convert mainframe data on a UNIX or Windows platform.
* The last option is to develop your own code to handle all of the known nuances that can occur when dealing with legacy data. However, this option is not for the faint-hearted. The cost in time and effort to handle all of the possible data scenarios would most likely exceed the cost of either developing the reformat programs on the mainframe or purchasing one of the utilities to assist in handling the mainframe data.

Extracting from IMS, IDMS, Adabase, and Model 204

If you use any of these systems, you will need special extractors. For starters, there are ODBC gateways to each of these. It is beyond the scope of this book to discuss detailed extraction techniques for these legacy database systems, but we realize they may be important to some of you.

Flat Files

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → Implementation → Test/Release

Data Flow: Extract → Clean → Conform → Deliver

Flat files are the mainstay of any data-staging application. In most data warehouse environments you cannot avoid flat files. Flat files are utilized by the ETL process for at least three reasons:

* **Delivery of source data**. When data is sourced by mainframes or external vendors, it is quite often FTP'd to the data-staging area in flat files. Data that comes from *personal* databases or spreadsheets is also usually delivered via flat files.
* **Working/staging tables**. Working tables are created by the ETL process for its own exclusive use. Most often, flat files are used because I/O straight reads and writes to the file system are much faster than inserting into and querying from a DBMS.
* **Preparation for bulk load**. If your ETL tool does not support *in-stream* bulk loading, or you want a load file for safekeeping or archiving, you need to create a flat file on the file system after all of the data transformations have occurred. Once a flat file is created, your bulk load processor can read the file and load it into your data warehouse.

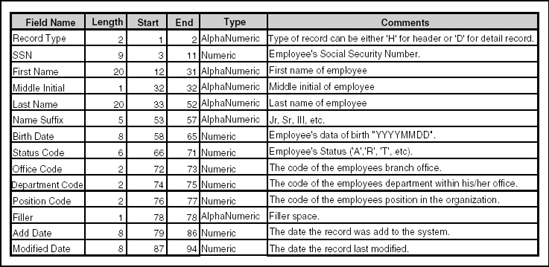
Not all flat files are created equally. Flat files essentially come in two flavors:

* Fixed length
* Delimited

Processing Fixed Length Flat Files

At times, you cannot access the physical data required to populate the data warehouse from its originating system. In those cases, you need to have a flat file created for you by the programmers who support the source system. Quite often, those files will be fixed length—also known as *positional* flat files. One of us was once on a data warehouse project that required very complex calculations from the legacy source system. The names of the calculations were given to the data warehouse architect, but the sources and calculations were a mystery. After some time-consuming investigation and detective work, the fields were discovered on a report that was written in COBOL in 1977. Naturally, the programmers that designed the report were no longer with the company. Moreover, the source code was nowhere to be found. As it turned out, the calculations were not so much complex as they were nonexistent.

Unfortunately, telling business users that data could not be derived and would not be available in the data warehouse was not an option. The team's solution was to redirect the report containing the required data to output to a flat file and use the prederived data from the report as a data source. Because of the nature of the report—fixed columns—it was simply treated as a positional flat file and processed with the rest of the ETL processes.



**Figure 3.12. Fixed length flat file layout.**

Processing a fixed length flat file requires a file layout to illustrate the exact fields within the file, as illustrated in [Figure 3.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html#fixed_length_flat_file_layout_dot). A fixed length file layout should include the file name, where the field begins; its length; and its data type (usually *text* or *number*). Sometimes, the end position is supplied. If it is not, you have to calculate the end position of each field based on its beginning position and length if it is required by your ETL tool.

In most ETL tools, you most likely have to manually input the file layout of the flat file once. After the layout is entered, the tool remembers the layout and expects that same layout each time it interacts with the actual flat file. If the file layout changes or the data shifts off of its assigned positions, the ETL process must be programmed to fail. Unfortunately, unlike XML, no implicit validation of the file layout occurs when you process fixed length flat files—an explicit preprocess test must be successful before the data is processed.

NOTE

When processing fixed length flat files, try to validate that the positions of the data in the file are accurate. A quick check to validate the positions is to test any date (or time) field to make sure it is a valid date. If the positions are shifted, the date field most likely contains alpha characters or illogical numbers. Other fields with very specific domains can be tested in the same way. XML offers more concrete validation abilities. If data validation or consistency is an issue, try to convince the data provider to deliver the data in XML format.

Positional flat files are often indicated on the file system by a .TXT extension. However, positional flat files can have virtually any file extension—or none at all—and be processed just the same.

Processing Delimited Flat Files

Flat files often come with a set of delimiters that separate the data fields within the file. Delimiters are used as an alternative to using positions to describe where fields begin and end. Delimited files can use any symbol or group of symbols to separate the fields in the flat file. The most common delimiter is the comma. Comma-delimited files can usually be identified by the .CSV extension on the file name. Obviously, however, other application-specific delimited flat files may simply have a .TXT extension or no extension.

Most ETL tools have a delimited file wizard that, once the developer indicates the actual delimiter characters, scans the flat file, or a sample of it, to detect the delimiters within the file and specify the file layout. Most often, the first row of delimited files contains its column names. The ETL tool should be intelligent enough to recognize the column names supplied in the first row to assign logical column names in the metadata layer and then ignore the row during all subsequent data processing.

Just as with positional flat files, no implicit validation on delimited files exists. They must have explicit validation tests written by the ETL team and embedded in the data-processing routines.

XML Sources

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release

Data Flow: *Extract* → Clean → Conform → Deliver

Extensible Markup Language (XML) is slowly but surely becoming the standard for sharing data. Much excitement has been generated by this new paradigm that stores data in a *well-formed* document. After all of its hype, we thought by now virtually all data warehouse sources would involve XML. But so far, we've observed that methods for sharing internal data have not changed all that much. On the other hand, methods for sharing external data have radically evolved in the past year or so to become almost completely XML.

XML has emerged to become a universal language for exchanging data between enterprises. If your data warehouse includes data that comes from external sources—those from outside of your enterprise—odds are that those sources will be provided in XML.

To process XML, you must first understand how it works. XML has two important elements: its metadata and the data itself. XML metadata can be provided in various ways. The next section illustrates different forms of XML metadata and what each means to the ETL.

Character Sets

Character sets are groups of unique symbols used for displaying and printing computer output. The default character set for most relational database management systems is ISO8859-15 (Latin 9). The character set supersedes ISO8859-1 (Latin 1) by enabling the euro sign: € The Latin character sets are intended to be used in the Western world to support languages based on the English alphabet. However, since XML is primarily used as a language for the Internet, it must support languages and alphabets from all over the world, not just the Western world. Therefore, XML supports the UTF-8 character set. UTF-8 is a character set that preserves the basic ASCII encoding method and also supports Unicode (ISO10646), the Universal Character Set (UCS). UTF-8 supports most of the languages and alphabets from around the world.

Many problems can arise if the source XML document and the target data warehouse are not based on the same character set. Of course, this flawed synchronization is always a risk when you integrate disparate systems (not just XML data sets). But in most cases, the risk is minimal because with few exceptions database systems use the Latin character sets. Organizations that don't use Latin-based character sets, but Unicode to support specific alphabets or characters, should adopt an enterprise-wide standard character set to avoid integration difficulties. Whenever you have a requirement to integrate data, especially using XML, from external sources, you must be ready to deal with dissimilar character sets. The good thing is that in XML, you can at least tag the document with the appropriate metadata to indicate the character set being used. For instance, the tag <?xml version="1.0" encoding="UTF-8" ?> indicates that the XML document is encoded using the UTF-8 character set.

XML Meta Data

We hear quite often that XML is nothing more than a flat file that contains data. In our opinion, that could not be farther from the truth. The only thing that makes XML remotely similar to a flat file is that it is stored on the files system as opposed to in the database. And in fact, many database systems are adding the capability to read, create, and store XML natively, calling themselves XML *enabled*.

XML is an odd entity because it stores data but is considered a language. It's not an application; therefore, it is dependent on other applications to make it work. Yet it is not merely data because of its embedded tags. The tags in XML documents are what make XML so powerful. But the self-describing data comes at a cost. The tags, which contain its metadata, can consume 90 percent of the XML file size, leaving about ten percent of the XML file for actual data. If your current flat files were XML, they could potentially be ten times their original size—containing exactly the same amount of raw data.

We find it ironic that a primary objective of data warehousing is to keep data as lean as possible to process it as quickly as possible and that even though XML tags cause a deviation from that goal by adding a considerable amount of overhead to the data processes, many still insist on making it a standard for data exchange. Not only do the tags increase the size of the data files; they also add substantial complexity to them. Because of such inherent complexity, never plan on writing your own XML processing interface to parse XML documents. The structure of an XML document is quite involved, and the construction of an XML parser is a project in itself—not to be attempted by the data warehouse team. There are many XML parsers (or processors) on the market, and most ETL vendors now include them in their product offerings.

NOTE

Do not try to parse XML files manually. XML documents need to be processed by an XML processor engine. Many of the major ETL tools now include XML processors in their suite. Make sure XML processing capabilities are in your ETL toolset proof-of-concept criteria.

To process an XML document, you must first know the structure of the document. The structure of an XML document is usually provided in a separate file. The next few sections discuss each of the possible metadata files that might accompany your XML and provide the structure of the XML document.

DTD (Document Type Definition)

As someone who views XML as a *data* source as opposed to a programming language, we equate the DTD to the COBOL file layout. It is a file that describes the structure of data in the XML document or file. Definitions can be embedded within an XML document, but to enable validation, keep the metadata and the actual data files separate. The DTD can be quite complex, incorporating such allowable XML data structures as the following:

* **Base Data**. If an element must contain only data, it is tagged with the #PCDATA declaration.
* **Element structures**. The structure of an element in a DTD is specified by listing element names within an element. For example, <!ELEMENT OrderLineItem (ProductID, QuantityOrdered, Price)> indicates that an order line item is composed of the Product ID, the quantity ordered, and the price of the item at the time of the order.
* **Mixed Content**. When either data or elements are allowed, PCDATA is declared to indicate that the base data is allowed and element names are indicated to enable the nested elements.
* **Nillable**. That's not a typo! In XML, you indicate if a field can be NULL with the 'nill=' or 'nillable=' tags. In the DTD, you'll see a question mark (?) to indicate that a subelement is optional. For example, the code <!ELEMENT Customer (FirstName, LastName, ZipCode?, Status)>indicates that the first and last name and status are required but the zip code is optional.
* **Cardinality**. One-to-many is indicated by the plus sign (+). For example, <!ELEMENT Customer (FirstName, LastName, ZipCode+, Status)> means that the customer can have more than one zip code.
* **Allowed Values**. Similar to a check constraint, XML enforces allowed values by listing the acceptable values separated by vertical bars. For example, <!ELEMENT State (Alabama|Louisiana|Mississippi)> indicates that the state must contain *Alabama, Louisiana*, or *Mississippi*.

At the time of this writing, the XML paradigm is still evolving and changing. Today, the DTD is viewed by many as *dated* technology. Because XML is evolving into more data-centric roles, similar to relational databases, most likely the DTD will be replaced by XML Schemas as the standard metadata. The next section discusses XML Schemas and how they are different from the DTD.

XML Schema

The XML Schema is the successor of the DTD. XML Schemas are richer and more useful than the DTD because they were created to extend the DTD. An XML Schema allows an SQL CREATE TABLE statement to be defined directly. This is not possible with simple DTDs, because the detailed data types and field lengths are not specified in DTDs. Some features of the XML Schema include the following:

* Elements that appear in an XML document
* Attributes that appear in an XML document
* The number and order of child elements
* Data types of elements and attributes
* Default and fixed values for elements and attributes
* Extensible to future additions
* Support of namespaces

Namespaces

XML is growing in popularity because it forces disparate data sources to send *consistent and expected*data files. But the reality is that different systems always have slightly different meanings and usage for the same elements within an entity. For example, if you receive a customer file from both human resources and operations, they might have different definitions of a customer. One department may deal with organizations, while the other transacts with individuals. Even though both are customers, organizations and individuals have very different attributes. To alleviate this situation, an XML document can refer to a *namespace*. A namespace indicates where to get the definition of an element or attribute. The same entity can take on a different meaning based on its declared namespace. The same customer entity with the namespace tag <Customer xmlns=http://www.website.com/xml/HRns> can have different meaning than the same entity referring to <Customer xmlns=http://www.website.com/xml/OPSns>.

NOTE

Because XML is emerging as the data source for Web-based applications, you will most likely see that increasingly reflected in your data warehouse data sources. When you pick your ETL tool, make sure it can natively process XML and XML Schemas.

Web Log Sources

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release

Data Flow: *Extract* → Clean → Conform → Deliver

Virtually every company in the world has a Web site. Beneath each Web site are logs—Web logs—that record every object either posted to or served from the Web server. Web logs are important because they reveal the user traffic on the Web site.

NOTE

A Web log in this section is not a *weblog* or *blog*! Our Web log is a control document automatically produced by every Web server. A blog is a kind of diary maintained and published by individuals, principally teenagers, for anyone to read.

Understanding the behavior of users on your Web site is as valuable as following a customer around a store and recording his or her every move. Imagine how much more organized your store can be and how many opportunities you can have to sell more merchandise if you know every move your customers make while navigating your store. Web logs provide that information. The activity of parsing Web logs and storing the results in a data mart to analyze customer activity is known as clickstream data warehousing.

NOTE

CROSS-REFERENCE An excellent source for more information on clickstream data warehousing is the book *Clickstream Data Warehousing* by Mark Sweiger, Mark R. Madsen, Jimmy Langston, and Howard Lombard (Wiley 2002).

From the data-modeling perspective, a clickstream data mart may be no more challenging than any other subject in the data warehouse. The ETL process, however, is significantly different from any other source you're likely to encounter. The difference is that the source to the clickstream is a text-based log that must be integrated with other source systems. Fortunately, the format of the text-based log is standardized. The standard is maintained by the World Wide Web Consortium (W3C).

W3C Common and Extended Formats

Even though the format of Web logs is standardized, its format and the content can vary. The operating system (OS) of the Web server and the parameter settings that control the log contents affect exactly what is written to the logs. Regardless of the OS, Web logs have a common set of columns that usually include the following:

* **Date**. This field is in a common date format—usually dd/mm/yyyy. If the time zone is adjusted during the ETL process, you must concatenate the date and time and adjust them together; otherwise, you may be a day off. You can split them up again upon loading.
* **Time**. This is the time of the Web hit. The format is HH:MM:SS and is usually set to Greenwich Mean Time (GMT). However, the time zone can be changed. Be sure you know what time zone your Web servers are set to. This usually involves conversion to the local time for the data mart.
* **c-ip**. This is the IP address of the user's Internet service provider (ISP). It is a standard IP address that can be used for domain name system (DNS) look-up to estimate where the user came from and *sessionizing*. Using the c-ip can be less than reliable because of known anomalies, such as AOL (the most popular ISP), which gives millions of users the same IP address and indicates that all of their users are from the same state in the United States, even though they might really be from all over the world.
* **Service Name**. This refers to the Internet service that was running on client computer, for example, w3svc1, w3svc2, w3svc3, and so on. This field identifies the site the log came from in environments that host many different Web sites or domains. This field is typically turned off for single-site environments.
* **s-ip**. This is the server IP address. It is standard IP address format. This is useful to identify individual Web servers in a Web farm environment. It also enables analysis of load balancing.
* **cs-method**. There are only two values possible in this field: POST or GET. Only GET records are usually stored in a Clickstream data mart.
* **cs-uri-stem**. This is the resource accessed (that is, the HTML or ASP page requested).
* **cs-uri-query**. This is the query the client passed. This field contains highly customizable, very valuable data. We call this and the cookie (discussed later in this list) the golden nuggets of the Web log. This field typically uses an ampersand (&) as a delimiter between label=value pairs, but any symbol is possible. More on parsing the cs-uri-query is discussed later in the "Name Value Pairs" section of this chapter.
* **sc-status**. This is the HTTP status, for example, 302 (redirect), 402 (error), and 200 (ok). A complete list of HTTP status codes can be found on the Web with a search for *HTTP status codes*. We recommend you preload the HTTP status dimension with all of the possible codes and their descriptions.
* **sc-bytes**. This is the number of bytes sent by the server. This is usually captured as a fact of the Hit.
* **cs(User-Agent)**. This is the browser type and version used by the client. The user-agent, along with the date and time, can be used to determine unique visitors.

NOTE

You can refer to *The Data Webhouse Toolkit: Building the Web-Enabled Data Warehouse* by Ralph Kimball and Richard Merz (Wiley 2000) for more information on alternative methods on identifying unique users on your Web site.

* **cs(Cookie)**. This is the content of the cookie sent or received, if any. This field is the other half, along with the cs-uri-query, of the gold found in the Web log. The cookie is highly customizable and very valuable. It can explicitly identify the user and many other characteristics about the user's session.
* **cs(Referrer)**. This is the URL describing the site that directed the user to the current site.

The following fields are available in the W3C extended format, but it is not an all-inclusive list. For an exhaustive list of the fields available in your environment, refer to your Web server documentation or the W3C Web site (www.w3c.org).

* **Server Name**. This is the name of the server on which the log was generated. This should be one-to-one with the s-ip, which is the IP address of the Web server.
* **cs-username**. This is the username and contains values only when the method is a POST.
* **Server Port**. This is the port number the client was connected to.
* **Bytes Received**. This is the number of bytes received by the user.
* **Time Taken**. This is the length of time the action took.
* **Protocol Version**. This is the protocol version used by the client, for example, HTTP 1.0, HTTP 1.1.

Name Value Pairs in Web Logs

Web logs consist of standard fields that are all distinct in content. For the most part, the content of the Web log can be extracted without too much transformation logic. This straightforwardness is especially true for the date, time, c-ip, service name, s-ip, cs-method, cs-uri-stem, sc-status, and sc-bytes. However, fields such as cs-uri-query and cs(Cookie) are not standard at all. In fact, it would be an extremely rare event to find two unrelated Web sites that have the same content in these fields. The cs-uri-query and cs(Cookie) contain customized name value pairs to capture specific attributes of a transaction that are important to the business.

The cs-uri-query typically contains detailed information about the transaction such as the product being served on the page. Consider the following cs-uri-query as an example:

/product/product.asp?p=27717&c=163&s=dress+shirt

Take a look at the query string and notice the following segments:

* **/product/**—The initial portion of the query string represents the directory that the executing program resides. The directory is always preceded and followed by a slash '/'. In this example, the product.asp program is in the product directory. If the program were in the *root* directory, a single slash would precede the program name.
* **product.asp**—This is the executing program file that generates the Web page. Common executing programs have the following extensions: .asp (active server pages) and .jsp (java server pages). The program file can be found immediately after directory.
* ?—The question mark indicates that parameters were sent to the program file. In this example, there are three parameters: p, c, and s.

Before the question mark, the query string is pretty standard. After the question mark is where the custom parameters for the program are stored. A different set of parameters can be defined by the Web site developer for each program file. The parameters are captured in the Web log in name-value pairs. In this example, you can see three parameters, each separated by an ampersand (&).

* **p** indicates the product number
* **c** indicates the product category number
* **s** indicates the search string entered by the user to find the product

NOTE

The ampersand (&) is the most common delimiter for separating parameters in the Web log, but it is not guaranteed. Make sure you visually scan the logs during your analysis phase to ensure that the parameter delimiters are identified.

Notice in the s= parameter that the search string is written as *dress* + *shirt*. Actually, the user entered *dress shirt*. The + was automatically inserted by the Web browser because HTTP cannot handle spaces. Extra care must be taken when you are processing textual descriptions in the query string. The ETL process must substitute any + in textual descriptions with a <space> before they are used for look-ups or stored in the data warehouse.

The intent of this section is to expose you to Web logs to give you a head start on your clickstream data mart project. Again, if you are deep into a clickstream project or plan to be in the near future, you should invest in either of the two books referenced earlier in this section.

ERP System Sources

NOTE

PROCESS CHECK Planning & Design:

*Requirements/Realities* → *Architecture* → Implementation → Test/Release

The existence of an ERP system has an immense effect on ETL system planning and design, as described in this section. This can range from treating the ERP system as a simple arms-length source of data, all the way to having the ERP system be the data warehouse and subsuming all the components, including the ERP system. We make recommendations in this section.

Data Flow: Extract → Clean → Conform (maybe) → Deliver

Enterprise resource planning (ERP) systems were created to solve one of the issues that data warehouses face today—integration of heterogeneous data. ERP systems are designed to be an integrated enterprise solution that enables every major entity of the enterprise, such as sales, accounting, human resources, inventory, and production control, to be on the same platform, database, and application framework.

As you can imagine, ERP systems are extremely complex and not easily implemented. They take months or years to customize so they contain the exact functionality to meet all of the requirements to run a particular business. As noble as the effort to be an all-inclusive solution is, it's very rare to see an entire enterprise use only an ERP system to run a company.

ERP systems are notoriously large, and because they are really a framework and not an application, their data models are comprehensive, often containing thousands of tables. Moreover, because of their flexibility, the data models that support ERP processing are incredibly difficult to navigate. The more popular ERP systems are SAP, PeopleSoft, Oracle, Baan, and J.D. Edwards.

Because of the sheer number of tables, attributes, and complexity of a typical ERP implementation, it is a mistake to attack these systems like any other transaction source system. Performing system and data analysis from scratch is cost and time prohibitive and leaves much room for error. If your data warehouse sources are from an existing ERP system, it is best to acquire someone with vast experience of the underlying database structure of your specific ERP system as well as the business objectives of the application.

To help you along, many of the major ETL vendors now offer ERP *adapters* to communicate with the popular ERP systems. If you are sourcing from an ERP system, take advantage of the available adapters. They can help you navigate the metadata in these systems and make sense of the application.

NOTE

Special Considerations for SAP

Because of the marketplace dominance of SAP as an ERP system, we get asked about the role of SAP in the data warehouse. Here are our unvarnished recommendations.

You have probably heard the cliché that, from a decision-support standpoint, SAP ERP is like a black hole: Rivers of data flow in, but there is no way to get the information back out. Why?

A contemporary SAP ERP implementation is likely to have a data foundation that consists of tens of thousands of physical tables, exhibiting few DBMS-defined table relationships, with entity and attribute names rendered in abbreviated German! Thus, the SAP RDBMS, for all practical purposes, is incomprehensible and proprietary. SAP ERP comes with an extensive library of operational reports, but these typically fall short of fully addressing the decision-support needs of most business communities. This is not a design flaw. SAP's OLTP data architecture simply lacks support for fundamental business-reporting needs, such as historical retention of transactions and master data images, comprehensible and easily navigated data structures, and robust query performance characteristics.

Some early SAP adopters tried to free their operational data trapped inside the ERP labyrinth by creating ERP subject areas in their data warehouses, populated via hand-crafted ETL. Predictably, with few specialized tools to assist them in this heroic undertaking, many of these efforts achieved unremarkable degrees of success.

Recognizing this unmet and blossoming need, SAP created a decision-support extension to their ERP application called the Business Information Warehouse (SAP BW). Early generations of SAP BW were rather primitive and consisted mainly of SAP-specialized ETL feeding proprietary OLAP repositories, thus lacking many of the foundational architectural elements of the contemporary data warehouse. Newer releases of SAP BW have evolved considerably and now embrace many of the core tenets and structures of contemporary data warehousing: better support for non-SAP data sources and persistent mainstream data repositories (Staging Areas, ODS, Data Warehouse, Dimensional Data Marts, and OLAP cubes). Some of these repositories support open access by third-party reporting tools.

SAP BW value proposition, at face value, now offers a compelling price and timeframe *sales*story that is likely to attract the attention of CIOs. And so, the contemporary DW architect will likely be asked to define and defend a role for SAP BW within the corporation's overall data warehousing vision.

In the following table, we present pros, cons, and recommendations for several SAP BW role scenarios within an overall enterprise DW strategy. We humbly recognize that this is a rapidly evolving area, with many variables, in which few fully satisfactory solutions exist. BW and ETL tool capabilities will change, thereby modifying the decision balances that follow. But we hope nonetheless that our evaluation process will be useful to you and extensible to your unique situation and challenges.

|  |  |
| --- | --- |
| BW as Enterprise DW | Utilize SAP BW as the foundational core of the enterprise data warehousing strategy. |
|  | We cannot recommend using BW in this role. Making BW the centerpiece of an enterprise BI architecture seems, at present, to be architecturally indefensible. As of this writing, BW offers no unique capabilities for cleansing / integration of non-SAP data or for the delivery of non-SAP analytics. Also, packaged BI tends to offer reduced opportunity for competitive differentiation through analytic capabilities.  Nonetheless, organizations that do not view BI as an area of strategic competitive differentiation and whose reporting requirements are SAP-centric and well addressed by SAP BW might consider using BW in this role. |
| BW on the Dimensional DW Bus | Utilize SAP BW as a set of satellite data marts that interoperate within a broader distributed dimensional data warehouse bus architecture. |
|  | Although this may someday become an appropriate way to utilize BW, we find (as of this writing) that BW is not yet well suited to play the role of an ERP-centric data mart within a broader dimensional data warehouse bus architecture. As we write this book, SAP has announced a facility called Master Data Management that claims to handle cross-functional descriptions of products and customers. It is too early to tell if this offers unique capabilities for either creating enterprise-wide conformed dimensions or for incorporating externally conformed dimensions into its ETL processing stream and presenting its facts accordingly. Utilizing conformed dimensions and facts across subject areas is a core tenet of the dimensional data warehouse bus architecture.  Nonetheless, IT organizations that already have dimensional data warehouses and *inherit* a BW may choose to take on the task of extending BW to utilize externally conformed dimensions, thereby allowing it to plug and play in the bus architecture. Be warned though: Maintenance of these extensions through SAP upgrades may not be a trivial undertaking. |
| ETL and Staging | Utilize SAP BW as a gateway and staging area for feeding ERP data to a downstream dimensional data warehouse. |
|  | This option appears at first to be quite compelling, because it purports to simplify and shorten the work effort to add ERP data into a dimensional data warehouse, while also providing, as a happy benefit, standard BW reporting capabilities for SAP-only reporting. But when comparing this option to the alternative of utilizing a specialized third-party ETL tool with good SAP connectivity (see "Forgo BW" as follows), it may no longer be the optimal solution. Using SAP as a data source through the ETL tool extracyors results in a single set of tables for the DW, typically offering greater control, flexibility, and metadata consistency.  We recommend this option for implementing your ETL system within SAP only to organizations with immature ETL teams under tight timeframes for ERP reporting, who find great value in the canned SAP BW reports. Others should look seriously at a mature ETL tool with good SAP connectors. See "Forgo BW" that follows. |
| Forgo BW | Utilize the SAP connectors offered by most good ETL tools to populate fully integrated ERP subject areas within a separate enterprise dimensional data warehouse bus architecture. |
|  | We're big believers in *buy versus build*, where appropriate. But based on BW's lack of a track record for either creating or publishing conformed dimensions or utilizing externally conformed dimensions, this is our recommended default BW architectural posture. |

Part 3: Extracting Changed Data

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release

Data Flow: *Extract* → Clean → Conform → Deliver

During the initial load, capturing changes to data content in the source data is unimportant because you are most likely extracting the entire data source or a potion of it from a predetermined point in time. But once that load is complete, the ability to capture data changes in the source system instantly becomes priority number one. If you wait until the initial load is complete to start planning for change data-capture techniques, you are headed for a heap of trouble. Capturing data changes is far from a trivial task. You must plan your strategy to capture incremental changes to the source data at the onset of your project.

The ETL team is responsible for capturing data-content changes during the incremental load. Desires and hopes are dictated by the users, and the realities of the feasibility of those hopes are revealed by the source system DBA team—if you're lucky. More often than not, a bit of research will be required to determine the best possible incremental load strategy for your specific situation. In this section, we offer several options and discuss the benefits and weaknesses of each. Naturally, you won't need all of these techniques for every situation. Choose the practice that best meets each ETL challenge throughout your project.

NOTE

Determining the appropriate strategy for identifying changed data in the source system may take some detective work. When analyzing source systems, never assume that what you see is what you get. In many cases, there will be unused or disabled audit columns or, even worse, columns used inconsistently. Be sure to allocate enough research time to investigate and determine the best approach to capture data-content changes for your incremental load process.

Detecting Changes

When managers talk about the maintenance of a data warehouse, most often they are talking about keeping the data current so it is a true reflection of the company's operational position. Capturing changes to the source system content is crucial to a successful data warehouse. The maintenance of the data content is dependent on the incremental load process. There are several ways to capture changes to the source data, and all are effective in their appropriate environments.

Using Audit Columns

In most cases, the source system contains audit columns. Audit columns are appended to the end of each table to store the date and time a record was added or modified. Audit columns are usually populated via database triggers fired off automatically as records are inserted or updated. Sometimes, for performance reasons, the columns are populated by the front-end application instead of database triggers. When these fields are loaded by any means other than database triggers, you must pay special attention to their integrity. You must analyze and test each of the columns to ensure that it is a reliable source to indicate changed data. If you find any NULL values, you must to find an alternative approach for detecting change.

The most common environment situation that prevents the ETL process from using audit columns is when the fields are populated by the front-end application and the DBA team allows *back-end* scripts to modify data. If this is the situation in your environment, you face a high risk that you will eventually miss changed data during your incremental loads. A preventative measure to minimize your risk is to stipulate that all back-end scripts be validated by a quality-assurance team that insists and tests that the audit fields are populated by the script before it is approved.

Once you are confident that audit columns are dependable, you need a strategy for utilizing them. There are various methods to implement the utilization of audit columns to capture changes to data. All of the methods have the same logical objective: to compare the last modified date and time of each record to the maximum date and time that existed during the previous load and take all those that are greater.

One approach we've found effective is to utilize the audit columns in the source system. Essentially, the process selects the maximum date and time from the create date and last modified date columns. Some last modified columns are updated upon insertion and with each change to the record. Others are left NULL upon insertion and updated only with changes after the record has already been inserted. When the last modified date is not populated, you must default it with an arbitrary *old* date in order to not lose new records. The following code can help resolve NULL modified dates:

select max(greatest(nvl(create\_date,'01-JAN-0001'),

nvl(last\_mod\_date,'01-JAN-0001')))

In cases where the rows in the fact table are inserted but never updated, you can simply select records from the source system where the create date and time is greater than the maximum date and time of the previous load and ignore the last modified date column.

Since fact tables and dimension tables can be sourced from many different tables and systems, and since fact tables consist only of foreign keys and measures, you do not store the audit dates in the fact table directly. You need to create an ETL last-change table that captures each source table and the maximum date time found in the source system audit columns at the time of each extract. If your fact table requires audit statistics for its rows, consider implementing an audit dimension as described in [Chapter 4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html).

Database Log Scraping or Sniffing

Log scraping effectively takes a snapshot of the database redo log at a scheduled point in time (usually midnight) and scours it for transactions that affect the tables you care about for your ETL load. Sniffing involves a polling of the redo log, capturing transactions on the fly. Scraping the log for transactions is probably the messiest of all techniques. It's not rare for transaction logs to blow-out, meaning they get full and prevent new transactions from occurring. When this happens in a production-transaction environment, the knee-jerk reaction for the DBA responsible is to empty the contents of the log so the business operations can resume. But when a log is emptied, all transactions within them are lost. If you've exhausted all other techniques and find log scraping is your last resort for finding new or changed records, persuade the DBA to create a special log to meet your specific needs. You presumably need transactions only for a few specific tables out of the hundreds in the source database. Those tables can populate your dedicated log via insert and update triggers.

If you want to pursue log sniffing, we recommend that you survey the available ETL tools in the market to find a proven solution rather than attempt to write the process from scratch. Many real-time ETL solution providers utilize log-sniffing techniques.

Timed Extracts

Select all of the rows where the date in the Create or Modified date fields equal SYSDATE-1, meaning you've got all of yesterday's records. Sounds perfect, right? Wrong. Loading records based purely on time is a common mistake made by most beginning ETL developers. This process is horribly unreliable.

Time-based data selection loads duplicate rows when it is restarted from midprocess failures. This means that manual intervention and data cleanup is required if the process fails for any reason. Meanwhile, if the nightly load process fails to run and misses a day, a risk exists that the missed data will never make it into the data warehouse. Unless your ETL process is extremely straightforward and the data volume is exceptionally small, avoid loading data based purely on time.

Process of Elimination

Process of elimination preserves exactly one copy of each previous extraction in the staging area for future use. During the next run, the process takes the entire source table(s) into the staging area and makes a comparison against the retained data from the last process. Only differences (deltas) are sent to the data warehouse. Albeit not the most efficient technique, the process of elimination is the most reliable of all incremental load techniques for capturing changed data. Because the process makes a row-by-row comparison, looking for changes, it's virtually impossible to miss any data. This technique also has the advantage that rows deleted from the source can be detected. These deleted rows are sometimes missed by other techniques.

This technique can be accomplished inside or out of a database management system. If you prefer using your DBMS, you must bulk load the data into the staging database for efficiency.

Initial and Incremental Loads

Create two tables: previous load and current load.

The initial process bulk loads into the current load table. Since change detection is irrelevant during the initial load, the data continues on to be transformed and loaded into the ultimate target fact table.

When the process is complete, it drops the previous load table, renames the current load table to previous load, and creates an empty current load table. Since none of these tasks involve database logging, they are very fast!

The next time the load process is run, the current load table is populated. Select the current load table MINUS the previous load table. Transform and load the result set into the data warehouse.

Upon completion, drop the previous load table and rename the current load table to previous load. Finally, create an empty current load table.

Since MINUS is a notoriously slow technique when inside the database management system, you'll want to use the ETL tool or third-party application to perform the process-of-elimination routine.

Extraction Tips

Consider the following points as you approach the extract process:

* **Constrain on indexed columns**. Work with the DBA to ensure all of the columns in your WHERE clause are indexed in the source system; otherwise you will probably provoke a relation scan of the entire production database.
* **Retrieve the data you need**. The optimal query returns exactly what you need. You shouldn't retrieve an entire table and filter out unwanted data later in the ETL tool. One situation that might break this rule is if the transaction system DBA refuses to index columns needed to constrain the rows returned in your query. Another exception is when you are forced to download the entire source database to search for the deltas.
* **Use DISTINCT sparingly**. The DISTINCT clause is notoriously slow. Finding the balance between performing a DISTINCT during the extract query versus aggregating or grouping the results in your ETL tool is challenging and usually varies depending on the percentage of duplicates in the source. Because there are many other factors that can affect this decision, all we can recommend is to take care to test each strategy for the most efficient results.
* **Use SET operators sparingly**. UNION, MINUS, and INTERSECT are SET operators. These, like DISTINCT, are notoriously slow. It's understood that sometimes these operators cannot be avoided. A tip is to use UNION ALL instead of UNION. UNION performs the equivalent of a DISTINCT, slowing the process. The hitch is that UNION ALL returns duplicates, so handle with care.
* **Use HINT as necessary**. Most databases support the HINT keyword. You can use a HINT for all kinds of things, but most importantly to force your query to use a particular index. This capability is especially important when you are using an IN or OR operator, which usually opts for scanning a full table scan rather than using indexes, even when usable indexes exist.
* **Avoid NOT**. If at all possible, avoid non-equi constraints and joins. Whether you use the keyword NOT or the operators '<>', your database will most likely opt to scan a full table rather than utilize indexes.
* **Avoid functions in your where clause**. This is a difficult one to avoid, especially when constraining on dates and such. Experiment with different techniques before committing to use of a function in your WHERE clause. Try using comparison keywords instead of functions whenever possible. For example:
* LIKE 'J%' instead of SUBSTR('LAST\_NAME',1,1)='J'
* EFF\_DATE BETWEEN '01-JAN-2002' AND '31-JAN-2002' instead of

TO\_CHAR(EFF\_DATE, 'YYY-MON') = '2002-JAN'

The goal of the extract query is to get all of the relevant natural keys and measures. It can be as simple as selecting multiple columns from one table or as complex as actually creating nonexistent data and can range from joining a few tables to joining many tables across heterogeneous data sources. On a specific project, we had to create a periodic snapshot fact table that needed to present sales for every product in inventory even if there were no sales for the product during the specified period. We had to *generate* a product list, get all of the sales by product, and perform an outer join between the product list and the sales by product list, defaulting the nonselling product sales amounts with zero.

Detecting Deleted or Overwritten Fact Records at the Source

Measurement (fact) records deleted or overwritten from source systems can pose a very difficult challenge for the data warehouse if no notification of the deletion or overwrite occurs. Since it is usually infeasible to repeatedly re-extract old transaction records, looking for these omissions and alterations, the best we can offer are the following procedures:

* Negotiate with the source system owners, if possible, explicit notification of all deleted or overwritten measurement records.
* Periodically check historical totals of measurements from the source system to alert the ETL staff that something has changed. When a change is detected, drill down as far as possible to isolate the change.

When a deleted or modified measurement record is identified, the late-arriving data techniques of the previous section can be used. In cases of deleted or modified fact records, rather than just performing a deletion or update in the data warehouse, we prefer that a new record be inserted that implements the change in the fact by canceling or negating the originally posted value. In many applications, this will sum the reported fact to the correct quantity (if it is additive) as well as provide a kind of audit trail that the correction occurred. In these cases, it may also be convenient to carry an extra administrative time stamp that identifies when the database actions took place.

Summary

In this chapter, we have isolated the extract step of the ETL data flow. We recommended that you step back at the very start and make sure the proposed extracts are even worth it! You can make this go/no-go decision with a data-profiling tool that will tell you if data is of sufficient quality to meet your business objectives.

The next big step is preparing the logical data map that connects the original source data to the ultimate final data. Perhaps the most important part of the logical data map is the description of the transformation rules applied between these inputs and outputs. Since a certain amount of discovery and refinement will take place as you actually implement the ETL system, you should expect to go back and periodically update the logical data map. If it is well maintained, it will be perhaps the most valuable description of your ETL system. At some point, a new person will have to decipher what you did, and he or she should start with the logical data map.

The central focus of this chapter was a tour of the various source systems you are likely to encounter. We gave you more than a teaspoon of some of the extract complexities, but of course, nothing is as valuable as real experience.

The last part of the chapter described the challenge of extracting just the new data, the changed data, and even the deleted data. In subsequent chapters, we point out the special processing needed in these situations.

Chapter 4. Cleaning and Conforming

Cleaning and conforming are the main steps where the ETL system adds value. The other steps of extracting and delivering are obviously necessary, but they only move and reformat data. Cleaning and conforming actually changes data and provides guidance whether data can be used for its intended purposes.

In this chapter, we urge you to build three deliverables: the data-profiling report, the error event fact table, and the audit dimension. You can build a powerful cleaning and conforming system around these three tangible deliverables.

The cleaning and conforming steps generate potent metadata. Looking backward toward the original sources, this metadata is a diagnosis of what's wrong in the source systems. Ultimately, dirty data can be fixed only by changing the way these source systems collect data. Did we say *business process re-engineering*?

Metadata generated in the cleaning and conforming steps accompanies real data all the way to the user's desktop. Or at least it should. The ETL team must make the cleaning and conforming metadata available, and that is where the audit dimension comes in.

Please stay with us in this chapter. It is enormously important. This chapter makes a serious effort to provide specific techniques and structure for an often amorphous topic. The chapter is long, and you should probably read it twice, but we think it will reward you with useful guidance for building the data cleaning and conforming steps of your ETL system.

If you are new to ETL system design, you may well ask "What should I focus on as a bare minimum?" Perhaps our best answer is: Start by performing the best data-profiling analysis you are capable of. You will then be much more usefully calibrated about the risks of proceeding with your potentially dirty or unreliable data. Armed with these understandings from the data-profiling step, you will have decomposed the problem and you will be more confident in designing a simple error event fact table and a simple audit dimension.

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → *Architecture* → *Implementation* → Test/Release Data Flow: Extract → *Clean* → *Conform* → Deliver

This chapter is organized into four data-quality topics:

* Part 1: Design Objectives
* Part 2: Cleaning Deliverables
* Part 3: Screens and Their Measurements
* Part 4: Conforming Deliverables

This is a top-down explanation of data quality. In the objectives section, we urge you to be *thorough, fast, corrective* and *transparent*. The perspectives of the cleaning and conforming steps are less about the upside potential of the data and more about containment and control. In some ways, this is unfamiliar territory for the data warehouse team. In the deliverables section, we introduce the mainstay structures of the cleaning subsystem: the error event table and the audit dimension. We also urge you to study Appendix B of Jack Olson's book in order to design a systematic structure for the results of your up-front data-profiling pass.

Descending a level further, the screens section defines a set of checkpoints and filters that you set up in many places to measure data quality. With screens, we build a unified approach to capturing data-quality events and responding to these events with appropriate actions.

NOTE

Definition: data-quality screen.

Throughout this chapter, we refer to data-quality screens. We use the word *screen* both to mean report and filter. Thus, a data-quality screen is physically viewed by the ETL team as a status report on data quality, but it's also a kind of gate that doesn't let bad data through.

In the fourth part, we describe the big deliverables of the conforming step: the conformed dimensions and facts and how they are handed off. We also suggest some metadata approaches to keep track of the decisions the organization has made to standardize your dimensions and facts.

Finally, the measurements section is a kind of implementer's guide to specific data-quality issues. Much like the details of the different types of extraction issues we describe in [Chapter 3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html), these data-quality measurements are a reasonable set for you to build upon.

This chapter draws liberally from the work of the leading data-quality authors Jack Olsen and Larry English. Their simple and most direct techniques for measuring data quality have been used in this chapter and placed within a kind of a template for building sensible data-quality ETL processes.

Defining Data Quality

Let's agree on some basic vocabulary, focused on *accuracy*. Accurate data means that the data is:

* **Correct**. The values and descriptions in data describe their associated objects truthfully and faithfully. For example, the name of the city in which one of the authors currently live is called New Hope. Therefore, accurate data about that home address needs to contain New Hope as the city name to be correct.
* **Unambiguous**. The values and descriptions in data can be taken to have only one meaning. For example, there are at least ten cities in the U.S. called New Hope, but there is only one city in Pennsylvania called New Hope. Therefore, accurate data about an address in this city needs to contain New Hope as the city name and Pennsylvania as the state name to be unambiguous.
* **Consistent**. The values and descriptions in data use one constant notational convention to convey their meaning. For example, the U.S. state Pennsylvania might be expressed in data as PA, Penn., or Pennsylvania. To be consistent, accurate data about current home addresses should utilize just one convention (such as the full name Pennsylvania) for state names and stick to it.
* **Complete**. There are two aspects of completeness.
  + The first is ensuring that the *individual* values and descriptions in data are defined (not null) for each instance, for example, by ensuring that all records that should have current addresses actually do.
  + The second aspect makes sure that the *aggregate* number of records is complete or makes sure that you didn't somehow lose records altogether somewhere in your information flow.

A related completeness issue surrounds the alternative meanings of missing values in data. A missing value represented as a null might mean that the true value is unknown or that it does not apply. Missing values may be represented as blanks, or strings of blanks, or as creative descriptions (Don't Know or Refused to Say).

Assumptions

The chapter makes a simplifying number of assumptions about the environment in which cleaning and conforming takes place. The first is that there are distinct points in the ETL job stream into which data-quality processing can be *injected*. Two such points are obvious from our model of the overall ETL data flow.

* The first processing milestone is when data has been *extracted*—which means that data has been extracted from some number of sources and placed into a physical or logical structure that is subject aligned. For example, customer information from various sources initially can be staged in a table, data file, or in-memory structure whose configuration is the same regardless of the data source. Such a structure could be used for incoming data from various data sources deposited and queued for ETL work. Little or no data cleansing or integration has yet been applied; data from one or several sources has simply been restructured and now sits waiting for further processing. This should not imply that data from multiple sources must be processed simultaneously, just that source-independent staging structures are utilized. The ETL data-quality techniques described still work even if source-specific staging structures and processing streams are used, but the metadata examples shown at the end of the chapter might need to be adjusted. We propose running lots of data-quality processes at this stage to get an accurate picture of the state of the organization's data quality while it is still *unvarnished* and to weed out hopelessly flawed data before it fouls up your data-cleansing and integration processes.
* The second milestone is when data has been *cleaned and conformed*—which means data has successfully passed through all of the data-preparation and integration components of the ETL stream and is ready for final packaging in the delivery step. We propose you run more data-quality processes at this stage, as a safety net for your data-cleansing and integration software. In essence, you want to run your newly manufactured information products through some quality-assurance checks before you turn them loose in the world.

For simplicity's sake, this chapter also assumes the use of batch ETL processing—rather than real-time or near real-time processing. We assume that batch processing aligns the techniques presented to the reality of most ETL environments and allows this chapter to direct its focus on data-quality issues and techniques rather than on the complexities of real-time ETL. We turn our attention to streaming ETL in [Chapter 11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch11.html).

Part 1: Design Objectives

NOTE

PROCESS CHECK Planning & Design:

*Requirements/Realities* → Architecture → Implementation → Test/Release Data Flow: Extract → *Clean* → Conform → Deliver

This part discusses the interrelated pressures that shape the objectives of data-quality initiatives and the sometimes conflicting priorities that the ETL team must aspire to balance. We propose some approaches to achieving this balance and in formulating a data-quality policy that meets the needs of important user constituencies.

Understand Your Key Constituencies

The data-quality subsystem must support the roles of data warehouse manager, the information steward, and the information-quality leader. Although these roles may be distributed in different ways across actual personnel, it's useful to characterize these roles.

Data Warehouse Manager

The data warehouse manager owns responsibility for the day-to-day decisions that need to be made in running the data warehouse, ensuring that it is an accurate reflection of the internal and external data sources and that data is processed according to the business rules and policies in place.

The cleaning and conforming subsystems should support the data warehouse manager and the surrounding business community by providing a history of the transformations applied to data as it is loaded into the warehouse, including a detailed audit of all exceptional conditions.

Information Steward

The information steward is accountable for defining the information strategy. This person formalizes the definition of analytic goals, selects appropriate data sources, sets information generation policies, organizes and publishes metadata, and documents limitations of appropriate use.

The cleaning and conforming subsystems should support the information steward by providing metrics on the operational data warehouse's day-today adherence to established business policy, issues with the source data that might be testing the boundaries of these policies, and data-quality issues that might call into question the appropriateness of the source data for certain applications.

Information-Quality Leader

The information-quality leader detects, corrects, and analyzes data-quality issues. This person works with the information steward to define policies for dealing with dirty data, setting publication quality thresholds, and balancing the completeness versus speed and corrective versus transparent tradeoffs described in the next section.

The data-quality subsystem should support the information-quality leader by providing data-quality measurements that describe the frequency and severity of all data-quality issues detected during the data warehouse ETL processes. This record should be a complete historical audit, allowing the information-quality leader to assess the success of data-quality improvement efforts over time.

Dimension Manager

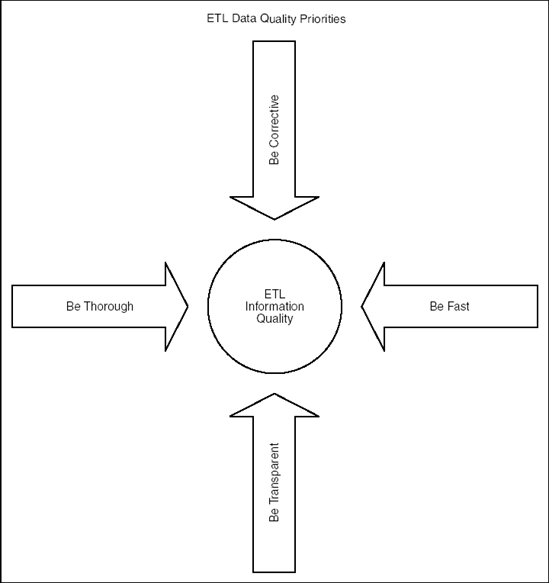
The dimension manager creates and publishes one or more of the conformed dimensions used by the overall organization. There may be multiple dimension managers, each responsible for different dimensions. The dimension manager implements the agreements on common descriptive labels reached by various stakeholders in the overall data warehouse. The dimension manager creates and assigns surrogate keys and assigns version numbers to each release of a dimension to the target fact table environments. When a dimension is released to the data warehouse community, it is replicated simultaneously to all the destinations so that they may install the new version of the dimension simultaneously. The job of the dimension manager is centralized: A conformed dimension must have a single, consistent source. We provide more on the role of the dimension manager later in this chapter.

Fact Table Provider

The fact table provider is the local DBA who *owns* the single instance of a given fact table. The fact table provider is responsible for receiving dimensions from various dimension managers, converting local natural keys to the surrogate keys in the conformed dimensions, and making updated fact tables available to the user community. The fact table provider may have to make complex changes in existing fact tables if postdated (late) dimension records are received. Finally, the fact table provider is responsible for creating and administering aggregates, which are physically stored summary records used to accelerate performance of certain queries. We provide more on the role of the fact table provider later in this chapter.

Competing Factors

Four interrelated pressures or priorities shape the objectives of your data-quality system as depicted in [Figure 4.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#data-quality_priorities).



**Figure 4.1. Data-quality priorities**

Be Thorough

The data-cleaning subsystem is under tremendous pressure to be thorough in its detection, correction, and documentation of the quality of the information it publishes to the business community. End users want to look to the data warehouse as a source of trusted information—a rock upon which to build their management metrics, strategies, and policies.

Be Fast

The whole ETL pipeline is under tremendous pressure to process ever-growing volumes of data in ever-shrinking windows of time. Some of the newest and most interesting customer touch points are very detailed and intimate—like Web clickstream—and drive huge data volumes into the data warehouse.

Be Corrective

Correcting data-quality problems at or as close to the source as possible is, of course, the only strategically defensible way to improve the information assets of the organization—and thereby reduce the high costs and lost opportunity of poor data quality. However, the reality is that many organizations have not yet established formal data-quality environments or information-quality leaders. In such cases, the data warehouse team might be the first to discover quality issues that have been festering for years. This team is expected to do all that can be done to fix these problems.

Be Transparent

The data warehouse must expose defects and draw attention to systems and business practices that hurt the data quality of the organization. These revelations ultimately drive business process re-engineering, where the source systems and data entry procedures are improved. Undertaking heroic measures to mask data-quality defects at the source might be one of those situations where the remedy can be worse than the disease.

Balancing Conflicting Priorities

Clearly, it is impossible for the cleaning subsystem to address in absolute terms all of these factors simultaneously. They must be properly balanced—reflecting the priorities of each situation.

Completeness versus Speed

The data-quality ETL cannot be optimized for both speed and completeness. Instead, we aspire to find an appropriate point on the exponential relationship curve (see [Figure 4.2](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#completeness_versus_speed)) that strikes the balance we seek.

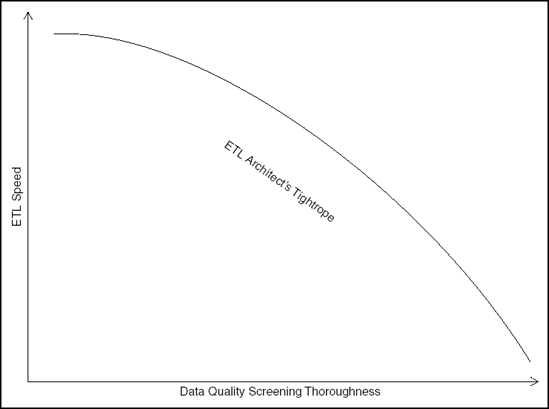
A potentially revealing way to best strike this balance is by asking some tough questions about the latency and quality of the data in your to-be-built data warehouse, such as:

* At what point does data staleness set in?

versus

* How important is getting the data verifiably correct?

If your data warehouse sponsors had to choose, for example, between a higher degree of confidence in data quality and a one-day delay in publication, which would they choose? A data warehouse that publishes daily might, for example, choose to trade one full day of latency for additional data-quality confidence, perhaps through expanded statistical variance testing or data standardization and matching or even selective manual review/auditing. If Monday's operational data were published on Wednesday rather than Tuesday, would this be an acceptable trade-off? There are no easy answers to questions like these.



**Figure 4.2. Completeness versus speed**

Corrective versus Transparent

The data-cleaning process is often expected to fix dirty data, yet at the same time provide an unvarnished view into the workings of the organization warts and all. Striking a proper balance here is essential: A transparency-atall-costs system can yield a feeble business-intelligence system that dilutes potential for insight, and a too-corrective system hides/obscures operational deficiencies and slows organizational progress.

The solution is to establish a sensible policy boundary between the types of defects that are *corrected verses highlighted* by the cleaning and to produce an easy-to-use audit facility (the audit dimension) that dutifully documents the modifications, standardizations, and underlying rules and assumptions of the error- detection and data-reengineering components.

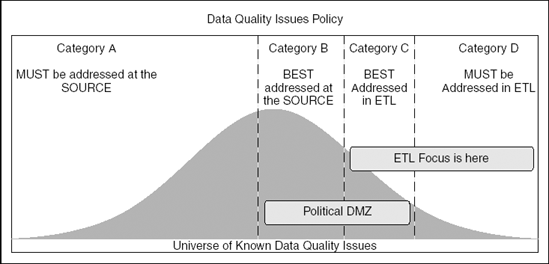
Data Quality Can Learn From Manufacturing Quality

The manufacturing quality revolution is now at least 30 years old. The seminal work on quality is W. Edwards Deming's total quality management (TQM) structure. His 14 points of managing TQM are worth reading while thinking about data quality, although outside the immediate scope of this book. But perhaps Deming's main point is that manufacturing quality requires a total commitment across every part of an organization: It is not a single inspector at the end of the assembly line!

Data quality can learn a great deal from manufacturing quality. One big step in this direction is the emergence of centralized data-quality groups in IT organizations. The data warehousing staff concerned with data quality must not operate independently from the data-quality group. The screens we define in this chapter should supplement other screens and assessment capabilities used by the data-quality team. These should feed a comprehensive database that incorporates results from all manner of data-quality measurements, not just the data warehouse. Most of the issues that come from ETL screens will result in demands to improve source systems, not in demands for more cleansing. All of the demands for improving data quality at the source need to be coordinated through the data-quality team.

Formulate a Policy

Shown in [Figure 4.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#data_quality_issues_policy) is one method for categorizing the set of data-quality challenges faced in data warehouse projects and isolating those that should be addressed by the ETL data-quality subsystems:



**Figure 4.3. Data Quality Issues Policy**

* **Category A** issues, for whatever reason, simply must be addressed at the data source. Examples might include missing information about the subject of a customer complaint or bogus information entered into a field that subjectively captures customer receptivity to a sales call. There is simply no technological way to derive or recreate this information. It must be captured correctly at the source, or it is lost. When addressing Category A data-quality issues, the cleaning subsystems should recognize them as deficiencies at the source, remove any clearly bogus information from the primary reporting and analysis dimensions and facts, and clearly label the information as missing or bogus thereby drawing management focus directly on the source system defect. In most data warehouse projects, the majority of data-quality issues discovered fall into this category—data-quality issues that must be detected and clearly communicated to the end user community.
* **Category D** (we know we are skipping) data-quality issues can only be pragmatically resolved in the ETL system. Examples might include missing or incomplete information from independent third-party data suppliers that can be reliably corrected through integration or the correction of bad data from an inflexible operational source system. Category D issues tend to be relatively rare in most data warehouse projects. In dealing with Category D issues, the ETL system is granted license to undertake creative/heroic measures to correct the data defect, but it must ensure that its polices and actions are visible to users through descriptive and complete metadata.
* **Category B** issues should be addressed at the data source even if there might be creative ways of deducing or recreating the derelict information. The boundary between Categories A and B is therefore technical rather than political. If a given data issue can be addressed with acceptable confidence through technology, it clearly belongs somewhere to the right of Category A in this bell curve.
* **Category C** issues, for a host of reasons, are best addressed in the data-quality ETL rather than at the source. Again, the boundary between Categories C and D is technical rather than political. If a given data-quality issue can be addressed reasonably at the source, it clearly belongs somewhere to the left of Category D in this bell curve.

By dividing and conquering our data-quality issues, we find that the only really tough boundary to define is that between Categories B and C: issues that, from a technology standpoint, can be addressed either at the source or in the ETL system. This is the *Political DMZ (demilitarized zone)*.

Part 2: Cleaning Deliverables

A serious undertaking to improve data quality must be based on rigorous measurement. This should include keeping accurate records of the types of data-quality problems you look for, when you look, what you look at, and the results. Further, you need to be able to answer questions from the data warehouse manager, information steward, and information-quality leader about your processing and the data-quality insights discovered, such as:

* Is data quality getting better or worse?
* Which source systems generate the most/least data-quality issues?
* Are there interesting patterns or trends revealed in scrutinizing the data-quality issues over time?
* Is there any correlation observable between data-quality levels and the performance of the organization as a whole?

Perhaps the data warehouse manager also asks:

* Which of my data-quality screens consume the most/least time in my ETL window?
* Are there data-quality screens that can be retired because the types of issues that they uncover no longer appear in our data?

The data-cleaning subsystem follows the extract step in the overall ETL processing stream. The primary deliverables, discussed in the next three sections, are:

* Data-profiling results
* An error event table
* An audit dimension

Data Profiling Deliverable

Data cleaning must actually start before the first step of building the ETL system. We have strongly urged that you perform a comprehensive data-profiling analysis of your data sources during the up-front planning and design phase. Good data-profiling analysis takes the form of a specific meta-data repository describing:

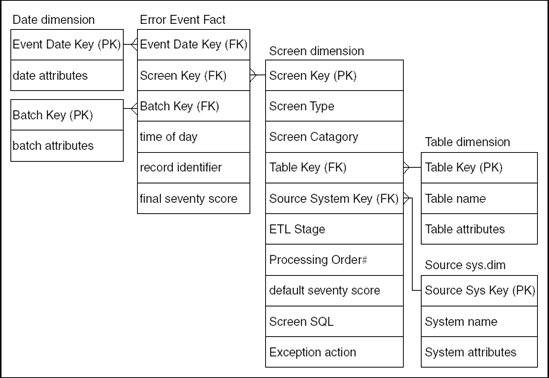
* Schema definitions
* Business objects
* Domains
* Data sources
* Table definitions
* Synonyms
* Data rules
* Value rules
* Issues that need to be addressed

Not only is data profiling a good quantitative assessment of your original data sources; this output should strongly influence the content of the two operations deliverables described as follows. Appendix B of Jack Olson's book, *Data Quality: The Accuracy Dimension*, has a comprehensive list of subcategories expanding the preceding list that should be created through data-profiling analysis to form the basis of the metadata repository.

Cleaning Deliverable #1: Error Event Table

The first major data-cleaning deliverable is a fact table called the error event table and a set of dimensions. This deliverable is structured as a dimensional data model, that is, as a dimensional *star schema*. (See [Figure 4.4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#error_event_table_schema))

Each data-quality error or issue surfaced by the data-cleaning subsystem is captured as a row in the error event fact table. In other words, the grain of this fact table is each error instance of each data-quality check. Remember that a quality check is a screen. So, if you were to run ten separate screens against some set of data and each screen uncovered ten defective records, a total of 100 records would be written to the error event fact table.



**Figure 4.4. Error event table schema**

The event date is a standard dimension representing the calendar date. The time of day is represented in the fact table as the number of seconds since midnight, expressed as an integer.

The batch dimension contains a record for each invocation of the overall batch process—and typically contains interesting timestamps, and numbers of records processed.

The screen dimension table contains constant descriptive information about each data-quality check, or screen, applied. It is not a description of a specific run (that is what the fact table records) but rather is a description of what the screen does and where it is applied. One of its attributes, the default severity score, defines a severity value for each of the various types of errors it may encounter. These error-severity scores are used as the basis of the final severity score error event fact table. For example, the final severity score could be higher than the individual default scores if a large number had accumulated.

The attributes of the screen dimension are as follows:

* The **ETL Stage** describes the stage in the overall ETL process in which the data-quality screen is applied.
* The **Processing Order Number** is a primitive scheduling/dependency device, informing the overall ETL master process of the order in which to run the screens. Data-quality screens with the same processing-order number in the same ETL stage can be run in parallel.
* The **Default Severity Score** is used to define the error-severity score to be applied to each exception identified by the screen in advance of an overarching processing rule that could increase or decrease the final severity score as measured in the fact table.
* The **Exception Action** attribute tells the overall ETL process whether it should pass the record, reject the record, or stop the overall ETL process upon discovery of error of this type.
* The **Screen Type** and **Screen Category Name** are used to group data-quality screens related by theme, such as Completeness or Validation or Out-of-Bounds.
* And finally, the **SQL Statement** captures the actual snippet of SQL or procedural SQL used to execute the data-quality check. If applicable, this SQL should return the set of unique identifiers for the rows that violate the data-quality screen so that this can be used to insert new records into the error event fact table.

For reporting purposes, it is useful to associate each screen to the table or set of columns that it scrutinizes, so that the information-quality leader can run reports that identify areas of data-quality problems and track these over time. This is the purpose of the table foreign key in the screen dimension.

The source system dimension identifies the source of the defective data. Because data-quality screens are run against both staged data that belongs to a single data source and data that may have been distilled from several sources, error events can be associated with a special (dummy) *integrated* source system.

The unique identifier of the defective record that allows the error event to be traced directly to the offending record is represented in the fact table as a *degenerate dimension* consisting of the ROWID or other direct pointer to the record in question. Note that with this design there is an implied responsibility to maintain referential integrity between this identifier in the screen dimension table and the real record. If you delete the real record, the screen record will be left as an orphan. The screen category field is simply used to categorize the types of errors detected by the screen. Possible values might include: Incorrect, Ambiguous, Inconsistent, and Incomplete, allowing the analyst to aggregate error events into interesting classifications.

The error event fact table is the central table for capturing, analyzing, and controlling data quality in the ETL system. All error events from all ETL processes should be written to this table. The screen dimension, of course, is the main driver for this table. This schema is the basis of the master control panel for the ETL system.

Cleaning Deliverable #2: Audit Dimension

The error event fact table described in the previous section captures data-cleaning events at the grain of the individual record in any and all tables in the ETL system. Obviously, these events may not occur at the grain of an individual record in a final delivered table being sent across to the front room. To associate data-quality indicators with the final end user fact tables, we need to build a dimension that is single valued at the grain of these tables. We will call this the audit dimension. The audit dimension describes the complete data-quality context of a fact table record being handed to the front room.

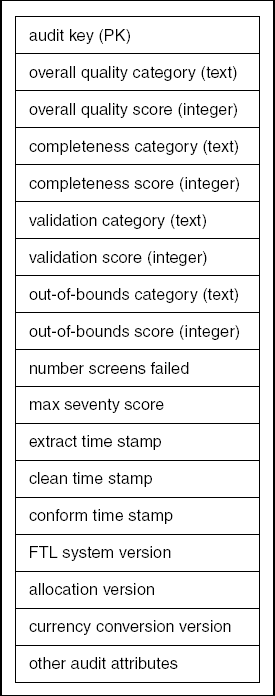
The audit dimension is literally attached to each fact record in the data warehouse and captures important ETL-processing milestone timestamps and outcomes, significant errors and their frequency or occurrence for the that record, and an overall data-quality score. Audit dimension records are created as the final step of the processing for cleaned and conformed fact table records and must contain a description of the fixes and changes that have been applied to the record.

NOTE

The audit dimension captures the specific data-quality context of an individual fact table record. This does not usually produce an enormous proliferation of audit dimension records, because the purpose of the audit dimension is to describe each type of data quality encountered. For instance, in the ideal case of a completely clean run of new data to be loaded into a fact table, only one audit record would be generated. Alternatively, if the run was clean except for a few input records that triggered out-of-bounds checks because of abnormally high values, two audit records would be generated: one for normal data records and one for out-of-bounds records. The vast majority of fact records would use the surrogate key for the normal audit record, and the few anomalous fact records would use the surrogate key for the out-of-bounds audit record.

A representative audit dimension design is shown in [Figure 4.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#audit_dimension).

The data-quality attributes and overall score are calculated by examining all error event facts for the integrated record and its associated source system records. The audit dimension contains a number of attributes calculated from the error event fact table by summing the error scores of the fact record, the scores of the conformed dimension instances that it is associated with, and each of the source records from which the integrated dimensions and facts were created. If you classify each screen, the aggregated data-quality score for each of these classifications can be carried in the audit dimension as descriptive attributes, both in textual and numeric form. The textual forms are useful for labeling reports with qualitative descriptions of error conditions. The data-quality completeness, validation, and out-of-bounds, audit dimension attributes shown in [Figure 4.4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#error_event_table_schema) are examples of this technique.



**Figure 4.5. Audit dimension**

Similarly, you can count the total number of error events and the maximum severity score for any one event as interesting attributes to be carried into the audit dimension. Finally, the audit dimension is a perfect placeholder for all of the timestamps and ETL code lineage attributes that you have collected supporting the analysis and troubleshooting tasks of the data warehouse manager.

Perhaps the biggest payoff comes from exposing the audit dimension to the end user community. End user queries and reports can now be run in normal mode and instrumented mode. By simply dragging one of the audit dimension attributes into the query or report, the original results get exploded into the separate contributions made by records with various data-quality conditions. Thus, a reported sales total for a large number of stores could now be broken into three lines (stores with normal sales, stores with abnormally high sales, and stores with abnormally low sales) merely by dragging the out-of-bounds category attribute into the report.

Notice that we have sneaked some global metadata context into the audit dimension! At the bottom of the figure are some global release and version numbers. Thus, if you have changed your revenue allocation scheme in the middle of the reporting period, you can drag the allocation logic version number attribute into the report and it will expand each results set row into the parts that were computed with the old scheme and the parts that were computed with the new scheme. We have elevated metadata to the status of data.

Audit Dimension Fine Points

A broadly accepted method to calculate an overall data-quality score for a fact record has not yet matured. The challenge is to define a method that presents the level of data quality that has actually been validated, doing so in a form that survives anticipated adjustments to the set of data-quality screens performed over time. After all, you don't want to have to revisit all of your data-quality scores for all facts in the warehouse every time that the information-quality leader adjusts the screens. If very few screens are performed, for example, the level of data quality actually validated should be lower than if more comprehensive sets of screens are added to the ETL stream later.

One technique for calculating the validated overall data score for a fact is to sum the error-event severity scores for all error-event records associated to the fact. Of course, this assumes that a source-to-target mapping of IDs is produced as a byproduct of the ETL *matching* data integration function (described later in this chapter). This sum of observed event scores can be subtracted from a worst-case error score scenario to determine the overall validated data-quality score used in the audit dimension. Worst-case error scores represent the sum of the maximum error-severity scores for all screens performed against extracted, cleaned, and conformed data. Thus, if ten distinct screens are performed against a single fact record and nine dimension records—each capable of generating a worst-case, data-quality severity score of ten—the overall worst-cast score total is 100. Restated: If every screen found defects in every screen that it applied, the cumulative data-quality severity score would be 100. Knowing this, you might choose to give this *absolutely flawed* fact an overall score of zero and assign a fact that has zero error events an overall score of 100. This technique, therefore, provides a measure of the overall data quality against the set of screens actually applied. If the organization chooses to add more screens to the ETL process, validated data-quality scores have the potential to rise. This seems reasonable, since the organization is now validating its data to a higher level of quality.

The structure of the audit dimension can be made unique to each fact table. In other words, you may choose to build a family of audit-dimension designs rather than forcing al audit dimensions to contain the same information. This would allow individual diagnoses of the quality of separate facts to be represented in a single audit dimension record. The key here is to preserve the dimensional character of this table.

NOTE

This section has discussed the design of an audit dimension that describes the data-quality diagnoses and actions pertaining to fact table records. As such, it is cleanly modeled as a dimension on each fact table. But is it possible to have an audit dimension for a dimension? Our answer is no; you don't need this. We prefer to embed the data-quality diagnoses and actions directly in the dimension table itself. Data-quality diagnoses of the overall reliability of the data should be included as additional fields in the dimension itself. Type 1 changes to a dimension (overwrites) can also be described in this way. Type 2 changes (alterations to atrributes at a particular point in time) already have extensive machinery available, including time stamps and reason codes, that can accomplish much of the purposes of a separate audit dimension. If a full audit trail of all changes to the data warehouse structures is needed for compliance reporting, you need to design special structures that record all these changes individually.

Part 3: Screens and Their Measurements

We are now ready to do some detailed design. This section describes a set of fundamental checks and tests at the core of most data-cleaning engines. It describes what these functions do, how they do it, and how they build upon one another to deliver cleaned data to the dimensional data warehouse. We are greatly indebted to Jack Olsen for creating the organization and vocabulary of the following sections, as described in his book *Data Quality: The Accuracy Dimension*.

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → *Clean* → Conform → Deliver

Anomaly Detection Phase

A *data anomaly* is a piece of data that does not fit into the domain of the rest of the data it is stored with. Remember when as a child you would be given a picture and would be asked, "What is wrong with this picture?" You would point out the square tires on a bicycle or the upside-down stop sign. Data anomalies are the square tires in the database. Detecting these anomalies requires specific techniques and entails analytical scrutiny. In this section, we explain anomaly detection techniques that have been proven successful on our data warehouse projects.

What to Expect When You're Expecting

Exposure of unspecified data anomalies once the ETL process has been created is the leading cause of ETL deployment delays. Detecting data anomalies takes a great deal of time and analysis. By doing this analysis up front, you save time and reduce frustration. The alternative is to have your time consumed by rebuilding the same ETL jobs over and over again while attempting to correct failed mappings caused by undiscovered data anomalies.

NOTE

Finding data anomalies may be perceived by some as data-quality issues outside the data warehouse, and they may well be, but unless your project is budgeted for a full-blown data-quality analysis subproject, chances are that detecting data anomalies will be the responsibility of the ETL team.

Data Sampling

The simplest way to check for anomalies is to count the rows in a table while grouping on the column in question. This simple query, whose results are shown in [Figure 4.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#result_of_value_distribution_query), reveals the distribution of values and displays potentially corrupt data.

select state, count(\*)

from order\_detail

group by state

As you can see in [Figure 4.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#result_of_value_distribution_query), data anomalies are instantly exposed. The outliers in the result set are data anomalies and should be presented to the business owner with a strong recommendation that they be cleaned up in the source system.

NOTE

TECHNICAL NOTE Data-profiling tools are built to perform exactly this kind of data sampling.

Analyzing source data sounds easy, right? What happens when your source table has 100 million rows, with 250,000 distinct values? The best approach to analyzing monster data sources is with *data samples*. We've used many different techniques for sampling data, ranging from simply selecting the first 1,000 rows to using the most elaborate algorithms, none of which are especially remarkable. We find that the following query, which simply counts the rows in the table and slices the table evenly into a specified number of segments, accurately samples the data regardless of the values in the table:



**Figure 4.6. Result of value distribution query**

select a.\*

from employee a,

(select rownum counter, a.\*

from employee a) B

where a.emp\_id = b.emp\_id and

mod(b.counter, trunc((select count(\*)

from employee)/1000,0)) = 0

To examine more or less data, simply alter the 1,000 to the number of rows you'd like returned in your sample.

Another approach involves adding a random number column to data, which can be sorted to select any desired fraction of the total table.

Once you have this sample data, you can perform your value-distribution analysis as usual. Selecting data by any other means, besides selecting all of it, can skew your tests results.

NOTE

A common mistake we've noticed is selecting a specific range of dates to narrow a result set. Data corruption usually occurs by bugs in the application program or by untrained staff. Most anomalies we've come across happen temporarily; then either the application is corrected or the person is replaced, and the anomaly disappears. Selecting data within a date range can easily miss these anomalies.

Types of Enforcement

It is useful to divide the various kinds of data-quality checks into four broad categories:

* Column property enforcement
* Structure enforcement
* Data enforcement
* Value enforcement

Column Property Enforcement

Column property enforcement ensures that incoming data contains expected values from the providing system's perspective. Useful column property enforcement checks include screens for:

* Null values in required columns
* Numeric values that fall outside of expected high and low ranges
* Columns whose lengths are unexpectedly short or long
* Columns that contain values outside of discrete valid value sets
* Adherence to a required pattern or member of a set of patterns
* Hits against a list of known wrong values where list of acceptable values is too long
* Spell-checker rejects

A number of specific screening techniques are discussed later in this chapter for performing precisely this set of validity checks and for capturing exceptions. Based on the findings of these screens, the ETL job stream can choose to:

1. Pass the record with no errors
2. Pass the record, flagging offending column values
3. Reject the record
4. Stop the ETL job stream

The general case is option two, passing records through the ETL stream and recording any validation errors encountered to the error event fact table to make these errors visible to the end user community and to avoid situations where data warehouse credibility is hurt by *Swiss cheese* data completeness. Data records that are so severely flawed that inclusion in the warehouse is either impossible or is damaging to warehouse credibility should be skipped completely, the error event duly noted, of course, in the error event fact table. And finally, data-validation errors that call into question the data integrity of the entire ETL batch should stop the batch process completely, so that the data warehouse manager can investigate further. The screen dimension contains an exception action column that associates one of these three possible actions to each screen.

Structure Enforcement

Whereas column property enforcement focuses on individual fields, structure enforcement focuses on the relationship of columns to each other. We enforce structure by making sure that tables have proper primary and foreign keys and obey referential integrity. We check explicit and implicit hierarchies and relationships among groups of fields that, for example, constitute a valid postal mailing address. Structure enforcement also checks hierarchical parent-child relationships to make sure that every child has a parent or is the supreme parent in a family.

Data and Value Rule Enforcement

Data and value rules range from simple business rules such as *if customer has preferred status, the overdraft limit is at least* $*1000* to more complex logical checks such as *a commercial customer cannot simultaneously be a limited partnership and a type C corporation*. Value rules are an extension of these reasonableness checks on data and can take the form of aggregate value business rules such as *the physicians in this clinic are reporting a statistically improbable number of sprained elbows requiring MRIs*. Value rules can also provide a probabilistic warning that the data may be incorrect. There indeed are boys named *Sue*, at least in Johnny Cash's song, but maybe such a record should be flagged for inspection. A priori if this record is incorrect, you don't know whether it is the name or the gender that should be corrected.

NOTE

These kinds of findings are hard to include in the error event fact table because the violations involve multiple records. Individual incorrect records are impossible to identify. One is left with two choices: Either tag all such records (sprained elbow requiring MRI) as suspect, or establish a virtual aggregate table on which errors can be reported as a count of incidences.

Measurements Driving Screen Design

NOTE

PROCESS CHECK Planning & Design:

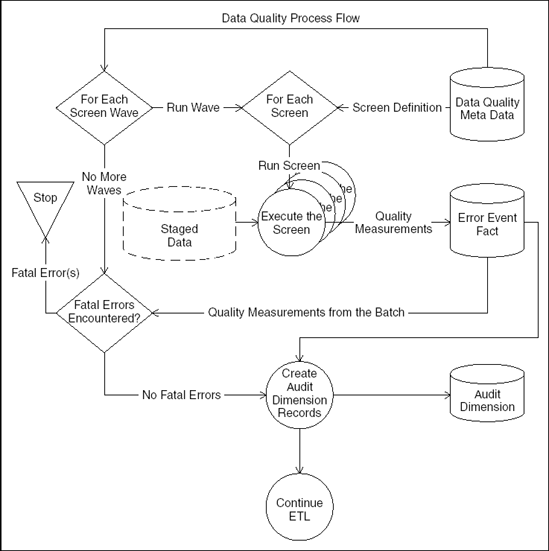
Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → *Clean* → Conform → Deliver

This section discusses what needs to go into the data-cleaning baseline for the data warehouse, including simple methods for detecting, capturing, and addressing common data-quality issues and procedures for providing the organization with improved visibility into data-lineage and data-quality improvements over time.

Overall Process Flow

A series of data-quality screens or *error checks* are queued for running—the rules for which are defined in metadata. Each screen is described in the screen dimension we build as part of the error event schema in the early part of this chapter. As each screen is run, each occurrence of errors encountered is recorded in an error event record. The metadata for each error check also describes the severity of the error event. The most severe data-quality errors are classified as fatal errors that will cause overall ETL processing to stop. An example of a condition that drives the creation of a fatal error event might be discovering that daily sales from several stores are completely missing or that an *impossible* invalid value for an important column has appeared for which there are no transformation rules.

When each of the data-quality checks has been run, the error event fact table is queried for fatal events encountered during the overall data-quality process. If none are found, normal ETL processing continues; otherwise, a *halt* condition is returned to the overall calling ETL process, which should then perform an orderly shutdown of the overall ETL process and proactively notify the data warehouse administrator and/or information-quality steward. This process is depicted in [Figure 4.7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#overall_process_flow-id2).



**Figure 4.7. Overall process flow**

For highest performance, the goal of the data-cleaning subsystem processing stream is to invoke *waves*of screens that can be run in parallel. These screens identify data-quality issues and insert records into the error event fact table. To minimize database contention issues, you should avoid unneeded indexing or constraints on the error event fact table so that records can stream into this table from several screen processes simultaneously without causing problems. The calling process waits for each *wave* of screens to complete before invoking the next wave of screens—until there are no more screen waves left to run. As indicated earlier in this chapter, the processing-order number of the screen metadata table is used for scheduling screens. Screens with the same processing order can be run in parallel. Standard data warehouse job scheduling tools can also be utilized for more comprehensive scheduling of screens and management of their dependencies.

When the cleaning subsystem completes its processing of the cleaned and conformed records, it performs some additional work in deriving an overall data-quality score for the audit dimension. It does this by aggregating the error event facts for the cleaned and conformed records in the stream and their associated source records (if this relationship is available)—saved as a byproduct of the ETL integration/matching processes. Interestingly, screens can also be applied to the error event fact table itself, allowing special screens to be established that measure the number and types of data-quality errors that have accumulated at any stage of the overall data-cleaning job stream. This technique is described further in the next section.

The recommended method for running screens is to build a generic software module that can execute any screen, given a batch ID and a screen surrogate key as parameters. This module extracts the metadata for the screen and constructs a dynamic INSERT statement that populates the error event fact table for each offending record returned by the screen. The general form of the dynamic INSERT statement is as follows:

INSERT INTO data\_quality\_error\_event\_fact

(etl\_batch\_surrogate\_key, day and time of day surrogate keys,

list of values from the Screen Meta Data record,

offending\_record\_surrogate\_key)

SELECT offending\_record\_surrogate\_keys provided by the Screen's SQL

Statement

The Show Must Go On—Usually

A guiding principle of the data-cleaning subsystem is to detect and record the existence of data-quality errors, not to skip records or to stop the ETL stream. Data-quality issues are an unfortunate fact of life in the data warehousing arena, and business managers are forced to make tough decisions every day in the face of incomplete and inaccurate data. This situation will not change overnight. Instead, you should aspire to provide the organization with tools to gauge the quality of the data they are utilizing and to measure their progress in improving data quality over time.

That said, the data-cleaning subsystem must also provide some mechanism for dealing with unexpected conditions, including data records that are simply too flawed to be permitted into the data warehouse or data records that indicate a systemic flaw so severe as to warrant a halt to the overall ETL process. For practical and political reasons, the thresholds for triggering these exceptional remedies must be balanced to allow the data warehouse to remain a viable and useful tool to the business, yet still provide enough assuredness of data quality to maintain system credibility within the end user community. This can be a tough balance to strike and is likely to be adjusted over time. So the ETL data-quality subsystem should support the ability to tune these thresholds and change the course of action to take when data-quality errors are encountered.

In some cases, exceptional actions might need to be taken if too many low-level error conditions are detected. For example, the existence of an invalid U.S. state code in a customer address record would typically cause an error event to be written to the data-quality subject area but would not stop the overall ETL process. If *all* of the records in the batch have invalid U.S. state codes, though, this probably indicates a severe problem in some upstream process—severe enough to call into question the overall integrity of all data in the ETL stream. It is recommended that cases like this be handled by creating additional data-quality screens run directly against the error event fact table, counting the number of data quality error event records captured in the overall data-cleaning batch and triggering exception processing.

NOTE

Take care with these special screens in their writing of their error findings back to the error event fact. They are reading from and writing to the same table—a recipe for database contention problems. Rather than writing error events for each offending record back to the fact, as do most other data-quality screens, they should instead aggregate error conditions of a specific type from a specific source table and write a single record error event fact if the aggregate exceeds the allowable threshold. This should sidestep most common contention issues.

Screens

Before screens can be run, you should have established an overall data-profiling baseline. This should include defining column specifications for nullity, numeric column ranges, character column length restrictions, and table counts. There is no substitute for performing in-depth research on data, on a source-by-source basis, for determining the characteristics of high-quality examples of data. This research should contain a review of the technical documentation of the data providers and a column-by-column review of the source data itself. For each data source to be loaded into the data warehouse, a data-profiling checklist should include:

* Providing a history of record counts by day for tables to be extracted
* Providing a history of totals of key business metrics by day
* Identifying required columns
* Identifying column sets that should be unique
* Identifying columns permitted (and not permitted) to be null
* Determining acceptable ranges of numeric fields
* Determining acceptable ranges of lengths for character columns
* Determining the set of explicitly valid values for all columns where this can be defined
* Identifying frequently appearing invalid values in columns that do not have explicit valid value sets

Without dedicated data-profiling tools, a limited subset of the data-profiling benefits can be obtained with hand-coded SQL, a team of subject matter experts, and time and effort. This make-versus-buy tradeoff mirrors the discussion of choosing an overall ETL tool we present at the beginning of this book. In other words, the vendor-supplied tools are continuously raising the bar, making it less and less practical to roll your own, unless your needs and aspirations are very modest. The findings from the data-profiling exercise should be maintained by the information-quality leader—who can then apply them directly to the data-quality screen metadata definitions that drive the ETL data-quality process.

Known Table Row Counts

In some cases, the information-quality leader absolutely knows, through business policy, the number of records to be expected of a given data type from a given data provider. An example of this might be a weekly inventory of parts from a warehouse, where the inventory of all active parts must be provided—even if zero. In other cases, the information-quality leader can infer a range of acceptable records to expect from a given data-provider-based history and build screens that detect record counts that are uncharacteristically high or low. The known table record count case can be handled by simple screen SQL, such as the following:

SELECT COUNT(\*)

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name''

HAVING COUNT(\*) <> 'Known\_Correct\_Count'

Because this is a table-level screen, the cleaned or conformed record identifier of the error event fact should be stored as a NULL.

Column Nullity

The determination of which columns are required (versus allowed to be null) in data records is very important and typically varies by source system. For example, a point-of-sale operational system might be permitted to have missing customer address attributes, but a record from a shipping system might demand non-null values. The metadata structures proposed capture nullity rules on a source-by-source basis. In dimensional models, integrated records often have more restrictive nullity rules than source data, because nearly all dimensional attribute columns are required to be populated—even if with only *Unknown, Not Applicable*, or *Not Available* descriptive strings.

NOTE

Systematically populating null text fields with an actual value removes the ambiguity of whether the field is missing or legitimately empty. This technique also simplifies many SQL lookups; unfortunately, relational databases treat the empty text field differently from the null text field. Even if a value is not supplied for the null text field, we recommend at least converting all null text fields to empty text fields.

The proposed approach for testing nullity is to build a library of source-specific nullity SQL statements that return the unique identifiers of the offending rows, such as the following:

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name'

AND column IS NULL

For screening errors from integrated records, you might adjust the SQL slightly to use your special *dummy* source system name, as follows:

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Integrated'

AND column IS NULL

Rows are inserted into the error event fact for each offending record returned by this screen, and the unique identifiers of the offending rows are written into the fact table as degenerate dimensions.

Column Numeric and Date Ranges

Although many numeric and date columns in relational database tables tolerate a wide range of values, from a data-quality perspective, they may have ranges of validity that are far more restrictive. Is it believable that a single customer transaction is for one million units? Perhaps yes, if our business is a global B2B exchange, but no, if this is a consumer retail point-of-sale transaction. You want your data-cleaning subsystem to be able to detect and record instances of numeric columns that contain values that fall outside of what the information-quality leader defines as valid ranges. In some cases, these valid value ranges will be defined by the source system. In other cases, especially for numeric columns that participate in sensitive ETL calculations, these ranges might need to be set by the information steward. Here again, columns of integrated data may have valid numeric ranges different from those of any data source, so you need to validate these with separate screens. An example of a SQL SELECT statement to screen these potential errors follows:

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name'

AND numeric\_column IS NOT BETWEEN min AND max

NOTE

Suppose we have a fact table that tracks daily sales in 600 stores, each of which has 30 departments. We therefore receive 18,000 sales numbers each day. This note describes a quick statistical check, based on calculating standard deviations, that allows us to judge each of the 18,000 incoming numbers for *reasonableness*. The technique also lets us quickly update the statistical base of numbers to get ready for tomorrow's data load.

Remember that the standard deviation is the square root of the variance. The variance is the sum of the squares of the differences between each of the historical data points and the mean of the data points, divided by N-1, where N is the number of days of data. Unfortunately, this formulation requires us to look at the entire time history of sales, which, although possible, makes the computation unattractive in a fast-moving ETL environment. But if we have been keeping track of SUM SALES and SUM SQUARE SALES, we can write the variance as (1/(N-1))\*(SUM SQUARE SALES - (1/N)\*SUM SALES\*SUM SALES). Check the algebra!

So if we abbreviate our variance formula with VAR, our data-validity check looks like:

SELECT s.storename, p.departmentname, sum(f.sales)

FROM fact f, store s, product p, time t, accumulatingdept a

WHERE

(first, joins between tables...)

f.storekey = s.storekey and f.productkey = p.productkey and

f.timekey = t.timekey and s.storename = a.storename and

p.departmentname = a.departmentname and

(then, constrain the time to today to get the newly loaded data...)

t.full\_date = #October 13, 2004# and

(finally, invoke the standard deviation constraint...)

HAVING ABS(sum(f.sales) - (1/a.N)\*a.SUM\_SALES) > 3\*SQRT(a.VAR)

We expand VAR as in the previous explanation and use the *a*. prefixonN, SUM SALES and SUM SQUARE SALES. We have assumed that departments are groupings of products and hence are available as a rollup in the product dimension.

Embellishments on this scheme could include running two queries: one for the sales MORE than three standard deviations above the mean and another for sales LESS than three standard deviations below the mean. Maybe there is a different explanation for these two situations. This would also get rid of the ABS function if your SQL doesn't like this in the HAVING clause. If you normally have significant daily fluctuations in sales (for example, Monday and Tuesday are very slow compared to Saturday), you could add a DAY\_OF\_WEEK to the accumulating department table and constrain to the appropriate day. In this way, you don't mix the normal daily fluctuations into our standard deviation test.

When you are done checking the input data with the preceding SELECT statement, you can update the existing SUM\_SALES and SUM\_SQUARE\_SALES just by adding today's sales and today's square of the sales, respectively, to these numbers in the accumulating department table.

Column Length Restriction

Screening on the length of strings in textual columns is useful in both staged and integrated record errors. An example of this screen might check customer last names that you believe are too long or too short to be credible. Here is an example of a SQL SELECT that performs such a screening:

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name'

AND LENGTH(numeric\_column) IS NOT BETWEEN min AND max.

Column Explicit Valid Values

In cases where a given column has a set of known discrete valid values as defined by its source system, you can screen for exceptions by looking for occurrences of default *unknown* values in the processed columns. Alternatively, you can treat this as a staging screen by using the generic column validity reference table of valid values for columns from any data providers. Therefore, a representative SQL statement might be:

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table Q

WHERE source\_system\_name = 'Source System Name'

AND column NOT EXISTS

( SELECT anything

FROM column\_validity\_reference\_table

WHERE column\_name = "column\_name"

AND source\_system\_name = 'Source System Name'

AND valid\_column\_value = Q.column\_value

)

Column Explicit Invalid Values

In cases where a given column is routinely populated with values known to be incorrect and for which there is no known set of discreet valid values, the information-quality leader might choose to explicitly screen for these invalid values. An example might be the periodic appearance of strings like UNKNOWN in a customer last name field—where the set of all potentially valid customer last names is undefined. The explicit invalid values screen should obviously not attempt to exhaustively filter out all possible invalid values—just pick off the frequent offenders. Other data-cleaning technologies, such as name and address standardization and matching, are far more appropriate for these tasks. For simplicity's sake, the example that follows hard-codes the offending strings into the screen's SQL statement.

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name'

AND UPPER(column) IN ("UNKNOWN", "?", list of other

frequent offenders...)

A slightly more elegant approach might compare the data values to a table full of frequent offenders, as in:

SELECT unique\_identifier\_of\_offending\_records

FROM work\_in\_queue\_table Q

WHERE source\_system\_name = 'Source System Name'

AND EXISTS

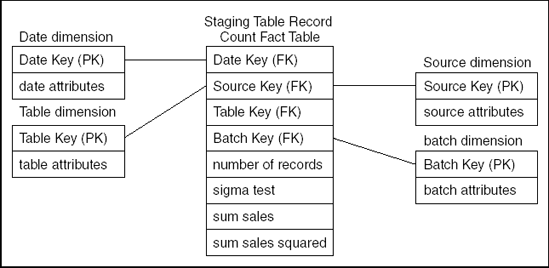
( SELECT 'Got One' FROM Table\_Of\_Frequent\_Offenders WHERE column\_name =

Q.column\_name)

If the set of valid values for a column is too large to be explicitly defined or is unknown, this type of screen has limited value, but in some useful cases the set of recently found violations can be used; data-entry people tend to repeat these violations over and over.

Checking Table Row Count Reasonability

This class of screens is quite powerful but a bit more complex to implement. It attempts to ensure that the number of rows received from a data source is reasonable—meaning that the row counts fall within a credible range based on previously validated record count histories. To test table row count reasonability, you can choose from a number of simple statistical tests such as calculating the number of standard deviations a value falls from the mean of previous similar values or opting for more advanced and professional value predictors such as the X.12 standard or even ARIMA (Autoregressive Integrated Moving Average) techniques. If you are interested in some of these powerful statistical tools, you'll need a few weeks of consulting with a good statistician. A good place to find such a statistician is in your marketing research department, if you have such a department.



**Figure 4.8. Table level reasonability metadata**

The data-staging table record count table shown in [Figure 4.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#table_level_reasonability_metadata) captures the number of records processed from each data source each day for each table—one row per data source per day.

[Figure 4.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#table_level_reasonability_metadata) presents these tables in a dimensional structure. Some ETL tools create similar tables as a byproduct of their normal operation. Using the data-staging table record count table, the SQL for this screen might be handled in two passes, as follows:

SELECT AVERAGE(Number\_of\_Records)-3 \* STDDEV(Number\_of\_Records),

AVERAGE(Number\_of\_Records) + 3 \* STDDEV(Number\_of\_Records)

INTO Min\_Reasonable\_Records,

Max\_Reasonable\_Records

FROM data\_staging\_table\_record\_count

WHERE source\_system\_name = 'Source System Name"

;

SELECT COUNT(\*)

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name"

HAVING COUNT(\*) NOT BETWEEN

Min\_Reasonable\_Records AND Max\_Reasonable\_Records

;

Clever SQL gurus can implement the preceding screen as either multipass SQL (as shown), single pass SQL for each data source, or a single screen that validates table record count reasonability from all sources—depending on specific data-quality requirements and severity score flexibility needed. The information-quality leader might also choose to define multiple screens for the same table and source system, with a different number of standard deviation tolerances applied and different severity scores, for example, recording low severity errors at two standard deviations from mean, graduating to high severity errors at three standard deviations from mean and to outright stoppage of the entire ETL stream at four standard deviations.

The table row count screen can easily be extended to support reasonability testing of any additive metric in the data warehouse. For example, by adding a total sales metric to the table in [Figure 4.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#table_level_reasonability_metadata), screens can be written that identify situations when sales metrics are inexplicitly skewed:

SELECT AVERAGE(Total\_Sales\_Dollars)-3

\* STDDEV(Total\_Sales\_Dollars),

AVERAGE(Total\_Sales\_Dollars) + 3

\* STDDEV(Total\_Sales\_Dollars)

INTO Min\_Reasonable\_Sales\_Dollars,

Max\_Reasonable\_Sales\_Dollars

FROM staging\_table\_record\_count

WHERE source\_system\_name = 'Source System Name"

;

SELECT SUM(Total\_Sales\_Dollars)

FROM work\_in\_queue\_table

WHERE source\_system\_name = 'Source System Name"

HAVING SUM(Total\_Sales\_Dollars) NOT BETWEEN

Min\_Reasonable\_Sales Dollars AND

Max\_Reasonable\_Sales\_Dollars

Checking Column Distribution Reasonability

The ability to detect when the distribution of data across a dimensional attribute has strayed from normalcy is another powerful screen. This screen enables you to detect and capture situations when a column with a discrete set of valid values is populated with a data distribution that is *skewed*abnormally. For example, the column being screened might be the product presented in a sales call fact from a sales force automation (SFA) system. Assume that history tells you that most sales calls are devoted to the presentation of product A (for which sales are highly compensated) and that very few present product B (which offers little reward to the sales force). You want to design a screen that will alert the information-quality leader if, say, you suddenly see too few sales calls for product A or too many sales calls for product B.

You build this screen by following an approach similar to the table row count reasonability technique described previously. Again, you are going to need a staging table to keep historical counts of the number of records seen for the valid values of a column over time, from which you can calculate means and standard deviations for your screen. Because there are often many possible values for a given column, and many columns with a discrete set of valid values, you will need to deviate from your metadata norms and propose staging tables that are specific to the table and sets of columns that are to be scrutinized by the screen. Of course, this increases the number of data-staging tables needed, but it affords the ETL architect much greater flexibility in physical implementation of these potentially large tables. In some cases, even this less-generalized data-staging approach generates a table that is too large to be used for high-performance ETL processing, so one can use the statistical technique described in a previous section for judging the mean and standard deviation of the data.

Note that the statistical approach described can also be used to support multicolumn screening—that is, testing for reasonability across several column combinations of valid values. Earlier in this chapter, we refer to this as value rule enforcement. An example of this might be scrutinizing daily sales by product and store, or daily sales by product, store, and day of the week, looking for results that are unreasonably skewed from historical norms.

Modifying the table in [Figure 4.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#table_level_reasonability_metadata) to add product as a dimension allows us capture daily sales-call counts by product. Using this table, the screen can compare the average sales-call totals by product code and source seen historically to those in the current ETL batch. Those products whose averages exceed the established threshold of standard deviations (as defined in the block of SQL in the screen definition) should have error event records written to the fact table.

Processing this type of screen using the technique described requires procedural programming on a level well supported by mainstream procedural SQL language extensions. This procedural SQL can be included in the screen SQL statement definition or handled outside of it. The important thing is for the ETL architect to be consistent in maintaining a screen metadata instance for all screens and in populating the error event fact for all error events surfaced by all screens.

Regardless of the implementation method chosen, the error event facts created by this screen are considered to be table-level screens, so the cleaned/conformed record identifier of the error event fact should be NULL.

General Data and Value Rule Reasonability

Data and value rules as defined earlier in the chapter are subject-matter specific, so we cannot give a list of specific checks for you to implement. But the form of the reasonableness queries clearly is similar to the simple data column and structure checks given in this section as examples.

Part 4: Conforming Deliverables

Integration of data means creating *conformed* dimension and fact instances built by combining the best information from several data sources into a more comprehensive view. To do this, incoming data somehow needs to be made structurally identical, filtered of invalid records, standardized in terms of its content, deduplicated, and then distilled into the new conformed image. In this section, we describe a three-step process for building conformed dimensions and facts:

* Standardizing
* Matching and deduplication
* Surviving

When we conform data, we may convert Gender Codes of (M, F), (M, W), and (Man, Woman) from three different data providers into a standard gender dimension attribute of (Male, Female). Similarly we can conform name and address information using specialized tools.

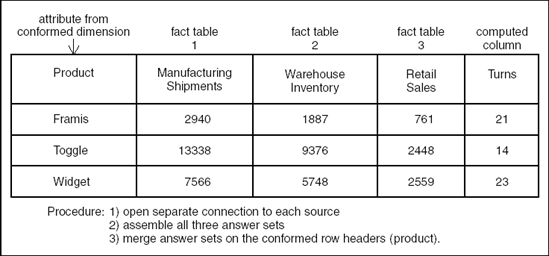
NOTE

Conforming descriptive attributes across multiple data sources, multiple data marts, and multiple remote clients participating in a distributed data warehouse is one of the key development steps for the data warehouse architect and the ETL team. Much has been written on the technical, administrative, and organizational affects this of this subject in the other Toolkit books. The immediate concerns of the ETL team are capturing the full range of overlapping and conflicting inputs and supporting the needs of the dimension manager and the fact-table provider.

Conformed Dimensions

Regardless of the hardware architecture, every data warehouse is distributed in a certain sense because separate kinds of measurements must always exist in separate fact tables. The same statement is true in an ER-modeled environment. So, for an end user application to combine data from separate fact tables, we must implement consistent interfaces to these fact tables so that data can be combined. We call these consistent interfaces *conformed dimensions* and *conformed facts*.

A conformed dimension means the same thing with every possible fact table to which it can be joined. Often, this means that a conformed dimension is identical for each fact table. A more precise definition of conformed dimensions is:



**Figure 4.9. Drilling across three fact tables**

*Two dimensions are conformed if they share one or more attributes whose values are drawn from the same domains. A requesting application must use only these common attributes as the basis for constraints and groupings when using the conformed dimensions to drill across separate fact tables*.

[Figure 4.9](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#drilling_across_three_fact_tables) illustrates the drill-across process for three fact tables supporting a conformed product dimension.

Examples of dimensions frequently conformed include customer, product, location, deal (promotion), and calendar (time). A major responsibility of the central data warehouse design team is to establish, publish, maintain, and enforce conformed dimensions.

The establishment of a conformed dimension is a very significant step for an organization. We describe the organization decisions and the overall procedure for arriving at the definitions of conformed dimensions in *Data Warehouse Lifecycle Toolkit*. A conformed customer dimension is a master table of customers with a clean surrogate customer key and many well-maintained attributes describing each customer. It is likely that the conformed customer dimension is an amalgamation and a distillation of data from several legacy systems and possibly outside sources. The address fields in the customer dimension, for instance, should constitute the best mailable address known for each customer anywhere within the enterprise. It is often the responsibility of the central data warehouse team to create the conformed customer dimension and provide it as a resource to the rest of the enterprise, both for legacy use and for data warehouse use.

The conformed product dimension is the enterprise's agreed-upon master list of products, including all product rollups and all product attributes. A good product dimension, like a good customer dimension, should have at least 50 separate textual attributes.

The conformed calendar dimension will almost always be a table of individual days, spanning a decade or more. Each day will have many useful attributes drawn from the legal calendars of the various states and countries the enterprise deals with, as well as special fiscal calendar periods and marketing seasons relevant only to internal managers.

Conformed dimensions are enormously important to the data warehouse. Without strict adherence to conformed dimensions, the data warehouse cannot function as an integrated whole. If a dimension like customer or product is used in a nonconformed way, either the separate fact tables simply cannot be used together or, worse, attempts to use them together will produce wrong results. To state this more positively, conformed dimensions make possible a single dimension table to be used against multiple fact tables in the same database space, consistent user interfaces and consistent data content whenever the dimension is used, and a consistent interpretation of attributes and therefore rollups across different fact tables.

Designing the Conformed Dimensions

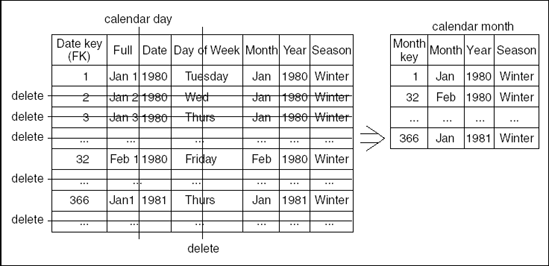
Identifying and designing the conformed dimensions should take a few weeks. Most conformed dimensions will naturally be defined at the most granular (atomic) level possible. The grain of the customer and product dimensions will naturally be the lowest level at which those entities are tracked in the source systems. The grain of the date dimension will usually be a day.

Taking the Pledge

If the central data warehouse team succeeds in defining and providing a set of master conformed dimensions for the enterprise, it is extremely important for the owners of separate fact tables to use these dimensions. The commitment to use the conformed dimensions is much more than a technical decision. It is a business-policy decision that is key to making the enterprise data warehouse function. The use of the conformed dimensions should be supported at the highest executive levels. This issue should be a sound bite for the enterprise CIO.

Permssible Variation of Conformed Dimensions

It is possible to create a subset of a conformed dimension table for certain fact tables if you know that the domain of the associated fact table contains only that subset. For example, the master product table can be restricted to just those products manufactured at a particular location if the data mart in question pertains only to that location. We can call this a simple data subset, since the reduced dimension table preserves all the attributes of the original dimension and exists at the original granularity.



**Figure 4.10. Building a conformed calendar month table**

A rollup data subset systematically removes both rows and columns from the original dimension table. For example, it is common to restrict the date dimension table from days down to months. In this case, we may keep only the record describing the first day of each month, but we must also remove all those attributes like Day-of-Week and Holiday-Flag that make sense only at a daily grain. See [Figure 4.10](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#building_a_conformed_calendar_month_tabl).

Perhaps you are wondering how to create queries in an environment where the conformed dimensions can be subsetted? Which dimension table should be used where? Actually, it is much simpler than it sounds. Each dimension table is naturally paired with its companion fact table. Any application that drills across fact tables must inevitably use multipass SQL to query each data mart separately and in sequence. It is usually the case that a separate SQL query is generated for each column in a drill-across report. The beauty of using conformed dimensions is that the report will run to completion *only if* the dimension attributes used in the report are found in each dimension table. Since the dimensions are conformed, the business answers are guaranteed to be consistent. The numbers will also be comparable if we have established conformed fact definitions.

Conformed Facts

We have talked thus far about the central task of setting up conformed dimensions to tie our data marts together. This is 80 percent of the up-front architectural effort. The remaining 20 percent is establishing standard fact definitions.

Fortunately, identifying the standard fact definitions is done at the same time as the identification of the conformed dimensions. We need standard fact definitions when we use the same terminology across fact tables and when we build single reports that drill across fact tables as described in the previous section.

Establishing conformed dimensions is a collaborative process wherein the stakeholders for each fact table agree to use the conformed dimensions. During conforming meetings, stakeholders also need to identify similar facts present in each of the fact tables. For instance, several fact tables may report revenue. If end user applications expect to add or compare these revenue measures from separate fact tables, the business rules that define these revenue measures must be the same. Perhaps revenue is measured by one group at the end of the month, whereas another group measures revenue on a rolling billing period. Or perhaps one group measures total sales, but another group measures only the Generally Accepted Accounting Principles (GAAP) recognized portion of the sale.

Conformed facts can be directly compared and can participate in mathematical expressions such as sums or ratios. If the stakeholders of the fact tables can reach agreement, the data-preparation steps for some or all of the fact tables may involve transformations of the facts in order to meet the common definition.

The Fact Table Provider

Although this section is more of an operational discussion, we want to complete the picture of the conforming dance we have described in this part of the chapter. In the next section, we define the role of a dimension manager, a centralized authority who prepares and publishes conformed dimensions to the community. The *fact table provider* is the receiving client of the dimension manager. The fact table provider owns one or more fact tables and is responsible for how they are accessed by end users. If fact tables participate in any enterprise-wide drill across applications, by definition they must use conformed dimensions provided by the dimension manager, and they must carefully prepare the numeric facts that have been identified by the organization as conformed (standardized) facts.

The Dimension Manager: Publishing Conformed Dimensions to Affected Fact Tables

A conformed dimension is by necessity a centrally managed object. A master *dimension manager*needs to be appointed by the organization to administer and publish each conformed dimension.

When the dimension manager releases a new version of a dimension, it is incumbent on the fact table provider to update local copies of the dimension as soon as possible. Ideally, the published dimension contains a version number field in every record, and all drill-across applications are enforcing the equality of this version number as they combine separate answer sets in the final step of preparing reports. If the fact table provider is tardy in updating dimensions, the drill-across application should fail because the version numbers don't match. Although this sounds harsh, it is very important for this discipline to be enforced; different versions of the same dimension can lead to insidious, unobservable errors in the drill-across results.

Each conformed dimension should possess a Type 1 version number field in every record (see the discussion of Type 1, 2, and 3 slowly changing dimensions in the [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) if this is unfamiliar vocabulary). This version number field is overwritten in every record whenever the dimension manager releases the dimension to the separate fact tables. Any drill-across query that combines data from two or more fact tables using two or more separate copies of a dimension must make sure that the version numbers of the dimensions match exactly. This requires the dimension manager to replicate any revised dimensions to all client fact tables simultaneously. In an environment supporting drill-across queries between fact tables, failure to enforce the equality of dimension versions is a very serious error, because applications may well run to completion, but sums and groupings can be insidiously wrong, with no real way to detect inconsistencies in the final reports.

In a single tablespace in a single DBMS on a single machine, managing conformed dimensions is somewhat simpler because there needs to be only one copy of a dimension. This single copy is joined at query time to all the fact tables resident in the tablespace. However, this benefit can be realized only in the smallest and simplest data warehouses. As soon as fact tables become situated in multiple tablespaces, multiple DBMSs, or multiple remote machines, the dimension manager must exercise the full set of responsibilities described in the previous paragraph, in order to support drilling across multiple data sets.

It is worth mentioning one more time that the roles described for the dimension manager and the fact table provider apply not only to geographically distributed and autonomous data warehouse environments but also to highly centralized warehouses on a single machine administered by a single set of DBAs. As soon as fact tables reside in separate table spaces, all these issues are relevant because there must be multiple physical copies of the dimensions.

Detailed Delivery Steps for Conformed Dimensions

The creation of conformed dimensions is more than just reaching agreement on certain standard descriptive attributes in a dimension. In the following steps, the references to Type 1, 2, and 3 slowly changing dimensions (SCDs) are explained in detail in [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html). The dimension manager must:

1. Add fresh new records to the conformed dimension, generating new surrogate keys.
2. Add new records for Type 2 changes to existing dimension entries (true physical changes at a point in time), generating new surrogate keys.
3. Modify records in place for Type 1 changes (overwrites) and Type 3 changes (alternate realities), without changing the surrogate keys. Update the version number of the dimension if any of these Type 1 or Type 3 changes are made.
4. Replicate the revised dimension simultaneously to all fact table providers.

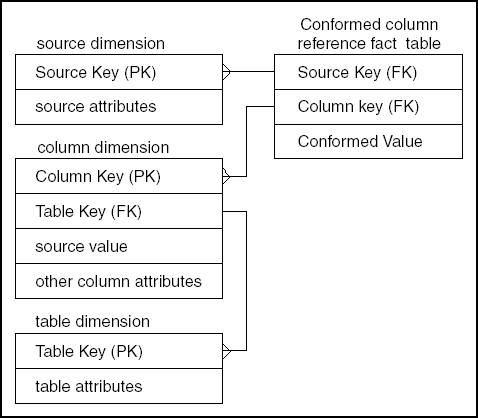
The receiving fact table provider has a more complex task. This person must:

1. Receive or download dimension updates.
2. Process dimension records marked as new and current to update current key maps in the surrogate key pipeline.
3. Process dimension records marked as new but postdated. This triggers a complex alternative to the normal surrogate key pipeline processing (described in [Chapters 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) and [6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html)).
4. Add all new records to fact tables after replacing their natural keys with correct surrogate keys.
5. Modify records in all fact tables for error correction, accumulating snapshots, and postdated dimension changes. You probably do this on a partition by partition basis. See comment in Step 9.
6. Remove aggregates that have become invalidated. An existing historical aggregate becomes invalidated only when a Type 1 or Type 3 change occurs on the attribute that is the target of the aggregation or if historical fact records have been modified in Step 5. Changes to other attributes do not invalidate an aggregate. For instance, a change in the Flavor attribute of a product does not invalidate aggregates based on the Category attribute.
7. Recalculate affected aggregates. If the new release of a dimension does not change the version number, aggregates have to be extended to handle only newly loaded fact data. If the version number of the dimension has changed, the entire historical aggregate may have to be recalculated if it was removed in Step 6. OLAP systems may handle these steps automatically.
8. Quality-assure all base and aggregate fact tables. Be satisfied that the aggregate tables are correctly calculated.
9. Bring updated fact and dimension tables on line. The detailed strategy for taking a fact table (or more likely a partition of a fact table) offline for the briefest possible duration can be found in the *Lifecycle Toolkit* book starting on page 645.
10. Inform end users that the database has been updated. Tell users if major changes have been made, including dimension version changes, postdated records being added, and changes to historical aggregates.

Implementing the Conforming Modules

To implement conformed dimensions and facts, the conforming subsystem needs reference metadata that captures the relationships between explicitly valid values from source systems to conformed dimension attribute values and conformed fact values.

Many ETL tools support these types of domain mappings, either with prebuilt metadata attributes or by allowing the ETL team to use extensible metadata attributes for the source table objects. [Figure 4.11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#conformed_column_support_schema)shows an example of metadata tables that support data conforming. The table and column entities capture metadata about each table and its associated columns, respectively. The fact table records the officially defined conformed values from each source system. The identities of the overall source systems are captured in the source system table. The column dimension contains the source value mapped into the official conformed value. Thus, in the simple example cited earlier, if Male and Female were the target conformed values for gender, the fact table would associate M with Male, F with Female from source system A; and M with Male but W with Female from source system B; and Man with Male and Woman with Female from source system C.



**Figure 4.11. Conformed column support schema**

Columns in records that contain invalid values—that is, values that are not in the set of explicit valid values in the column dimension table—should be replaced with a predefined value like Unknown from the standardized value reference table; replacement should be noted in the error event fact table.

NOTE

It is important that bogus or invalid data that cannot be standardized be removed from the visibility of downstream ETL processes (like matching) and the end user community.

More complex forms of standardization are now routinely used to deal with cleansing of names and addresses. Specialized software tools provide support in this area that would be very difficult for ETL teams to attempt to duplicate. By all means, have a look at the leading players in this arena listed in [Chapter 7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch07.html). In some cases, the standardized value is arrived at *probabilistically* by a technique that uses statistical techniques to align imperfect data with some known universe of common names or geographies. Be aware that some probabilistic standardization tools also have self-tuning integration engines that learn over time more about the distribution of data specific to the particular application and adjust their processing algorithms appropriately. This is a powerful feature but one that can challenge the ETL architect's ability to test a data-integration engine whose behavior changes/evolves as it gets smarter. Most standardization tools produce feedback on their success in reengineering data and in exceptions/errors encountered in processing of the data. It is important to capture, retain, and mine data-integration lineage observations—using the log tables left behind to generate error event facts.

Matching Drives Deduplication

Matching, or deduplication, involves the elimination of duplicate standardized records. In some cases, the duplicate can be easily detected through the appearance of identical values in some key column—like social security number, telephone number, or charge card number. This happy situation is all-too rare, unfortunately. In other cases, no such definitive match is found, and the only clues available for deduplicating are the similarity of several columns that almost match. In still tougher cases, more than one definitive-match columns are found to be identical, but they contradict one another.

Specialized data integration matching tools are now mature and in widespread use and deal with these very specialized data-cleansing issues. Often, these tools are closely associated with data-standardization tools and are sold together.

The matching software must compare the set of records in the data stream to the universe of conformed dimension records and return:

* A numeric score that quantifies the likelihood of a match
* A set of match keys that link the input records to conformed dimension instances and/or within the standardized record universe alone

Thus, an input record running through the match processes can be a match to zero or one conformed dimension records *and* zero, one, or more other input records in the batch process queue. In either case, the matching software's job is to associate match keys to these input records that detail these derived match relationships. These match keys are used by the survivor-ship module described in the next section in figuring which records have been matched to one another and are therefore candidates for distillation into a single integrated record.

Many data-matching tools also include a *match score*, or matching confidence metric, that describes the likelihood of match obtained. Often, these match scores are derived by creating several matching approaches, or passes, scoring match probabilities from each pass and then distilling results into a recommended set of match keys and an overall weighted score.

Organizations with a need for very robust deduplication capabilities can choose also to maintain a persistent library of previously matched data, each still associated with a single data provider, and use this consolidated library to improve their matching results. In this way, the matching engine can apply its match passes not just to the conformed dimension records but also to the complete set of previously matched dimension records that it has from all source systems. This approach might result in better matches, because the universe of match candidates is richer, and it is far more resilient to gracefully handling matching rule changes, which can now be satisfied without having to run every source system's data through the entire data-integration process. But this approach complicates match processing because matches can occur within and across the source and fully conformed data universes.

NOTE

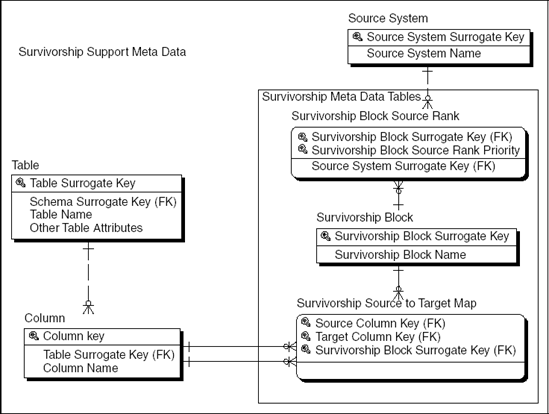
As of this writing, matching tools are far from turn-key implementations that plug into ETL streams and somehow know what to do. On the contrary, they require much profiling and training based on the organization's data, the establishing of matching strategies for deciding which combinations of attributes (matching passes or perspectives) are most predictive of duplication, the tuning to distill these different perspectives into a matching policy, and the setting of *match* and *no match* thresholds based on the organization's tolerance for over (aggressive) and under (conservative) matching. ETL tool-suite vendors have recently been targeting this application, however, and you should examine their plug-in matching transformers.

Surviving: Final Step of Conforming

Survivorship refers to the process of distilling a set of matched (deduplicated) records into a unified image that combines the highest-quality column values from each of the matched records to build conformed dimension records. This entails establishing business rules that define a hierarchy for column value selections from all possible sources and capturing the source-to-target mapping to be applied when writing out for survived (conformed) records.

In addition, survivorship must be capable of distilling combinations of columns together, rather than individually. This is needed for situations where the combining of individually survived columns could result in a nonsensical mishmash, such as combining address lines 1, 2, and 3 from three different source systems and ending up with a distilled address that is less credible than all three. It is far better in situations like this to create rules that mandate that certain combinations of columns (survivorship blocks) must be survived together: all or nothing. The metadata tables shown in [Figure 4.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html#survivorship_support_metadata) support the most common requirements of survivorship.

* The **Survivorship Source to Target Map table** captures data-integration mappings between source columns (input data that has been cleaned but not conformed) and target columns (conformed dimension table columns). For both flexibility and simplicity, it allows any combination of columns to be used as sources into any combination of targets—thus placing a burden on the ETL architect (rather than referential integrity that might have been included in a more complex structure) to populate it properly.
* The **Survivorship Block table** groups mapped source and target columns into blocks that must be survived together (to properly address the address 1, 2, 3 types of issues described earlier). Survivorship blocks are allowed to be of one source and one target, too, so by forcing all survivorship to be performed by block, you can simplify both the metadata model and survivorship processing. This table includes a rank that allows the priority of source system blocks of fields to be determined with dynamic SQL, which looks for non-null values in each block ordered by survivorship block source rank priority, and builds an appropriate INSERT or UPDATE statement depending on whether the match key already exists as a conformed record surrogate key (UPDATE) or not (INSERT).



**Figure 4.12. Survivorship support metadata**

NOTE

In cases where the deduplication process successfully coalesces separate source entities (such as customers) into a single entity, if the source entities have been assigned separate primary keys in the source system, a table of those obsolete primary keys should be maintained to speed subsequent deduplication runs using data from that source system.

Delivering

Delivering is the final essential ETL step. In this step, cleaned and conformed data is written into the dimensional structures actually accessed by the end users and application systems. In the smallest data warehouses consisting of a single tablespace for end user access, dimensional tables are simply written to this table space. But in all larger data warehouses, ranging from multiple table spaces to broadly distributed and autonomous networks of data marts, the dimensional tables must be carefully published in a consistent way. Delivering is so important that we devote [Chapters 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) and [6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html) to its details.

Summary

Stepping back from all the detail, this chapter covered four big topics: objectives, techniques, metadata, and measurements.

The *objectives* of data cleaning and conforming are to reduce errors in data, improve the quality and usefulness of the contents of data, and standardize key descriptive attributes and numerical measures shared across the organization.

*Data-quality techniques* range from examining individual field definitions at the data-base level (column property enforcement), to checking for fieldto-field consistency (structure enforcement), and finally to business-rule specific checks on data (data and value rule enforcement). The final phase of data-quality processing (conforming and deduplicating) is the most far reaching, since in this phase we resolve differences across separate data sources.

*Data-quality metadata* contains declarations and business rules that hold our techniques together. We described a methodology for building a family of screens, each representing a data-quality investigation. Some of the screens are run routinely as part of every ETL episode, and some are run occasionally as periodic sanity checks or special investigations. The routine screens supply diagnostic indicators and measurements that we store in a detailed error event fact table and in audit dimensions attached to our fact tables. These audit dimensions are interesting because in a sense they elevate metadata to real data. Data-quality indicators can participate in *instrumented* end user queries just as if they were normal data.

Finally, *data-quality measurements* we proposed are a starter set of measurements that the ETL team needs in order to build a comprehensive data-quality processing pipeline.

When data has made it through data-quality processing pipeline, it is ready for the final delivering step, laid out in detail in [Chapters 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) and [6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html).

Chapter 5. Delivering Dimension Tables

Dimension tables provide the context for fact tables and hence for all the measurements presented in the data warehouse. Although dimension tables are usually much smaller than fact tables, they are the heart and soul of the data warehouse because they provide entry points to data. We often say that a data warehouse is only as good as its dimensions. We think the main mission of the ETL team is the handoff of the dimension tables and the fact tables in the delivery step, leveraging the end user applications most effectively.

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → *Architecture* → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

[Chapters 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) and [Chapters 6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html) are the pivotal elements of this book; they describe in a highly disciplined way how to deliver data to end users and their analytic applications. While there is considerable variability in the data structures and delivery-processing techniques leading up to this handoff, the final ETL step of preparing the dimensional table structures is much more constrained and disciplined.

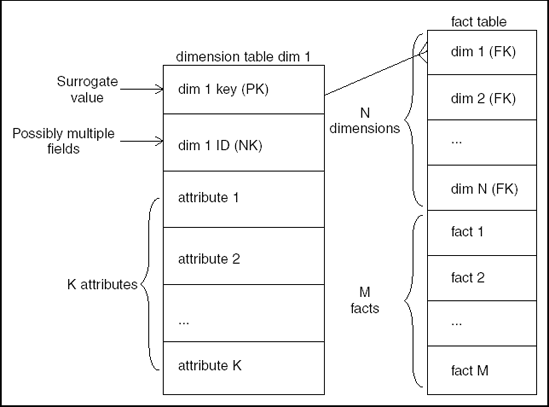
Please keep in mind that our insistence on using these highly constrained design techniques is not adherence to a foolish consistency of a dimensional modeling methodology but rather is the key to building data warehouse systems with replicable, scalable, usable, and maintainable architectures. The more a data warehouse design deviates from these standardized dimensional modeling techniques, the more it becomes a custom programming job. Most IT developers are clever enough to take on a custom programming job, and most find such development to be intellectually stimulating. But custom programming is the kiss of death for building replicable, scalable, usable, and maintainable systems.

The Basic Structure of a Dimension

All dimensions should be physically built to have the minimal set of components shown in [Figure 5.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#the_basic_structure_of_a_dimension). The *primary key* is a single field containing a meaningless, unique integer. We call a meaningless integer key a *surrogate*. The data warehouse ETL process should always create and insert the surrogate keys. In other words, the data warehouse owns these keys and never lets another entity assign them.

The primary key of a dimension is used to join to fact tables. Since all fact tables must preserve referential integrity, the primary dimension key is joined to a corresponding *foreign key* in the fact table. This is shown in our insurance example in [Figure 2.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch02.html#a_dimensional_model_for_an_insurance_pol) in [Chapter 2](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch02.html). We get the best possible performance in most relational databases when all joins between dimension tables and fact tables are based on these single field integer joins. And finally, our fact tables are much more compact when the foreign key fields are simple integers.

All dimension tables should possess one or more other fields that compose the *natural key* of the dimension. We show this in [Figure 5.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#the_basic_structure_of_a_dimension) as an *ID* and designate the natural key field(s) with *NK*. The natural key is not a meaningless surrogate quantity but rather is based on one or more meaningful fields extracted from the source system. For instance, a simple static (nonchanging) employee dimension would probably have the familiar EMP ID field, which is probably the employee number assigned by the human resources production system. EMP\_ID would be the natural key of this employee dimension. We still insist on assigning a data warehouse surrogate key in this case, because we must insulate ourselves from weird administrative steps that an HR system might take. For instance, in the future we might have to merge in bizarrely formatted EMP\_IDs from another HR system in the event of an acquisition.



**Figure 5.1. The basic structure of a dimension**

When a dimension is static and is not being updated for historical changes to individual rows, there is a 1-to-1 relationship between the primary surrogate key and the natural key. But we will see a little later in this chapter that when we allow a dimension to change slowly, we generate many primary surrogate keys for each natural key as we track the history of changes to the dimension. In other words, in a slowly changing dimension, the relationship between the primary surrogate key and the natural key is many-to-1. In our employee dimension example, each of the changing employee profile snapshots would have different and unique primary surrogate keys, but the profiles for a given employee would all have the same natural key (EMP\_ID). This logic is explained in detail in the section on slowly changing dimensions in this chapter.

The final component of all dimensions, besides the primary key and the natural key, is the set of *descriptive attributes*. Descriptive attributes are predominately textual, but numeric descriptive attributes are legitimate. The data warehouse architect probably will specify a very large number of descriptive attributes for dimensions like employee, customer, and product. Do not be alarmed if the design calls for 100 descriptive attributes in a dimension! Just hope that you have clean sources for all these attributes. More on this later.

The data warehouse architect should not call for numeric fields in a dimension that turn out to be periodically measured quantities. Such measured quantities are almost certainly facts, not descriptive attributes. All descriptive attributes should be truly static or should only change slowly and episodically. The distinction between a measured fact and a numeric descriptive attribute is not as difficult as it sounds. In 98 percent of the cases, the choice is immediately obvious. In the remaining two percent, pretty strong arguments can be made on both sides for modeling the quantity either as a fact or as a dimensional attribute. For instance, the standard (catalog) price of a product is a numeric quantity that takes on both roles. In the final analysis, it doesn't matter which choice is made. The requesting applications will look different depending on where this numeric quantity is located, but the information content will be the same. The difference between these two choices will start to become important if it turns out that the standard price is actually slowly changing. As the pace of the change accelerates, modeling the numeric quantity as a measured fact becomes more attractive.

NOTE

Generating Surrogate Keys for Dimensions

Creating surrogate keys via the DBMS is probably the most common technique used today. However, we see this trend changing. In the past, it was common practice to have surrogate keys created and inserted by database triggers. Subsequently, it has been determined that triggers cause severe bottlenecks in the ETL process and should be eliminated from any new processes being created. Even though it is still acceptable for the integers for a surrogate key to be maintained by the DBMS, these integers should be called by the ETL process directly. Having the ETL process call the database sequence will produce a significant improvement in ETL performance over the use of database triggers.

Also, using the database to generate surrogate keys almost guarantees that the keys will be out of sync across the different environments of the data warehouse—development, test, and production. As each environment gets loaded at different intervals, their respective database could generates different surrogate key values for the same incoming dimension records. This lack of synchronization will cause confusion during testing for developers and users alike.

For ultimate efficiency, consider having an ETL tool or third-party application generate and maintain your surrogate keys. Make sure that efficient generation and maintenance of surrogate keys are in your ETL proof-of-concept success criteria.

A tempting solution seen repeatedly during design reviews is concatenating the natural key of the source system and a date stamp that reflects when the record was either created in the source system or inserted into the data warehouse. Giving the surrogate key intelligence—the exact time of its creation—may be useful in some situations, but it is not an acceptable alternative to a true integer-based surrogate key. Intelligent or smart keys fail as an acceptable surrogate key for the following reasons:

* *By definition*. Surrogate keys, by definition, are supposed to be meaningless. By applying intelligence to the surrogate key, their responsibility is broadened, making them need to be maintained. What happens if a primary key in the source system changes—or gets corrected in some way? The concatenated smart key would need to be updated, as will all of its associated records in fact tables throughout the entire data warehouse.
* *Performance*. Concatenating the source system key with a date stamp degrades query performance. As part of the data warehouse team, you have no control over the content of source system keys and must be able to handle any data type. This fact forces you to use the CHAR or VARCHAR data types to accommodate alpha, numeric, or alphanumeric keys coming from the source systems. Moreover, by appending the date stamp to the key, potentially 16 characters or more, the field can become unwieldy. What's worse, this key will need to be propagated into huge fact tables throughout the entire warehouse. The space to store the data and indexes would be excessive, causing ETL and end user query performance to diminish. Additionally, joining these large VARCHAR concatenated columns during query time will be slow when compared to the same join using INTEGER columns.
* *Data type mismatch*. Veteran data warehouse data modelers will know to build the dimensional model surrogate keys with the NUMBER or INTEGER data type. This data type prevents alpha characters from being inserted, thwarting the use of the concatenated date stamp method.
* *Dependency on source system*. The use of the smart-key approach is dependent on the source system revealing exactly when an attribute in a dimension changed. In many cases, this information is simply not available. Without reliable maintenance of some kind of audit columns, attaining the exact timestamp of a change can be impossible.
* *Heterogeneous sources*. The concatenation of the natural key and date stamp supports only a homogeneous environment. In virtually all enterprise data warehouses, common dimensions are sourced by many different source systems. These source systems each have their own purpose and can uniquely identify the same values of a dimension differently. The concatenated natural key, date-stamp approach falls short with the introduction of a second source system. Natural keys from each system must be stored equally, in dedicated nonkey columns in the dimension. Imagine attempting to concatenate each natural key and their respective timestamps—a maintenance nightmare.

The attractive characteristic of using this forbidden smart-key strategy is its simplicity at ETL development time when building the first data mart, when it is quite simple to implement a smart key by appending the SYSDATE to the natural key upon insertion. Avoid the temptation of this prohibited shortcut. This approach doesn't scale to your second data mart.

The Grain of a Dimension

Dimensional modelers frequently refer to the *grain* of a dimension. By this they mean the definition of the key of the dimension, in business terms. It is then a challenge for the data warehouse architect and the ETL team to analyze a given data source and make sure that a particular set of fields in that source corresponds to the definition of the grain. A common and notorious example is the commercial customer dimension. It is easy to say that the grain of the dimension is the commercial customer. It is often quite another thing to be absolutely sure that a given source file always implements that grain with a certain set of fields. Data errors and subtleties in the business content of a source file can violate your initial assumptions about the grain. Certainly, a simple test of a source file to demonstrate that fields A, B, and C implement the key to the candidate dimension table source is the query:

Select A, B, C, count(\*)

From dimensiontablesource

Group by A, B, C

Having Count(\*) > 1

If this query returns any rows, the fields A, B, and C do not implement the key (and hence the grain) of this dimension table. Furthermore, this query is obviously useful, because it directs you to exactly the rows that violate your assumptions.

NOTE

It's possible that the extract process itself can be the culprit for exploding the rows being extracted, creating duplicates. For example, in a denormalized Orders transaction system, instead of referring to a source table that stores the distinct *Ship Via* values for the Order, the textual values of the attribute may very well be stored repeatedly directly in the Orders transaction table. To create the dimensional model, you build the Ship Via dimension by performing a SELECT DISTINCT on the Orders table. Any data anomalies in the original Orders table will create bogus duplicate entries in the Ship Via dimension.

The Basic Load Plan for a Dimension

A few dimensions are created entirely by the ETL system and have no real outside source. These are usually small lookup dimensions where an operational code is translated into words. In these cases, there is no real ETL processing. The little lookup dimension is simply created directly as a relational table in its final form.

But the important case is the dimension extracted from one or more outside sources. We have already described the four steps of the ETL data flow thread in some detail. Here are a few more thoughts relating to dimensions specifically.

Dimensional data for the big, complex dimensions like customer, supplier, or product is frequently extracted from multiple sources at different times. This requires special attention to recognizing the same dimensional entity across multiple source systems, resolving the conflicts in overlapping descriptions, and introducing updates to the entities at various points. These topics are handled in this chapter.

Data *cleaning* consists of all the steps required to clean and validate the data feeding a dimension and to apply known business rules to make the data consistent. For some simple, smaller dimensions, this module may be almost nonexistent. But for the big important dimensions like employee, customer, and product, the data-cleaning module is a very significant system with many subcomponents, including column validity enforcement, cross-column value checking, and row deduplication.

Data *conforming* consists of all the steps required to align the content of some or all of the fields in the dimension with fields in similar or identical dimensions in other parts of the data warehouse. For instance, if we have fact tables describing billing transactions and customer-support calls, they probably both have a customer dimension. In large enterprises, the original sources for these two customer dimensions could be quite different. In the worst case, there could be no guaranteed consistency between fields in the billing-customer dimension and the support-customer dimension. In all cases where the enterprise is committed to combining information across multiple sources, like billing and customer support, the conforming step is required to make some or all of the fields in the two customer dimensions *share the same domains*. We describe the detailed steps of conforming dimensions in the [Chapter 4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch04.html). After the conforming step has modified many of the important descriptive attributes in the dimension, the *conformed data* is staged again.

Finally, the *data-delivering module* consists of all the steps required to administer slowly changing dimensions (SCDs, described in this chapter) and write the dimension to disk as a physical table in the proper dimensional format with correct primary keys, correct natural keys, and final descriptive attributes. Creating and assigning the surrogate keys occur in this module. This table is definitely staged, since it is the object to be loaded into the presentation system of the data warehouse. The rest of this chapter describes the details of the data-delivering module in various situations.

Flat Dimensions and Snowflaked Dimensions

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

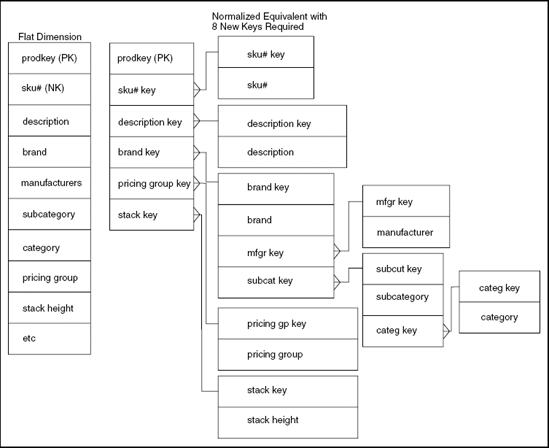
Dimension tables are denormalized flat tables. All hierarchies and normalized structures that may be present in earlier staging tables should be flattened in the final step of preparing the dimension table, if this hasn't happened already. All attributes in a dimension must take on a single value in the presence of the dimension's primary key. Most of the attributes will be of medium and low cardinality. For instance, the gender field in an employee dimension will have a cardinality of three (male, female, and not reported), and the state field in a U.S. address will have a cardinality of 51 (50 states plus Washington, DC). If earlier staging tables are in third normal form, these flattened second normal form dimension tables are easily produced with a simple query against the third normal form source. If all the proper data relationships have been enforced in the data-cleaning step, these relationships are preserved perfectly in the flattened dimension table. This point is consistently misunderstood by proponents of delivering data to end users via a normalized model. In the dimensional-modeling world, the data-cleaning step is separated from the data-delivery step, in such a way that all proper data relationships are delivered to the end user, without the user needing to navigate the complex normalized structures.

It is normal for a complex dimension like store or product to have multiple simultaneous, embedded hierarchical structures. For example, the store dimension could have a normal geographic hierarchy of location, city, county, and state and also have a merchandising-area hierarchy of location, district, and region. These two hierarchies should coexist in the same store dimension. All that is required is that every attribute be single valued in the presence of the dimension table's primary key.

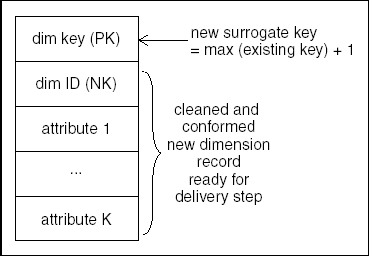
If a dimension is normalized, the hierarchies create a characteristic structure known as a snowflake, if indeed the levels of the hierarchies obey perfect many-to-1 relationships. See [Figure 5.2](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#flat_and_snowflaked_versions_of_a_dimens). It is important to understand that there is no difference in the information content between the two versions of dimensions in this figure. The difference we do care about is the negative impact the normalized, snowflaked model has on the end user environment. There are two problems. First, if the strict many-to-1 relationships in a hierarchical model change, the normalized table schema and the declared joins between the tables must change, and the end user environment must be recoded at some level for applications to continue working. Flat versions of the dimension do not have this problem. Second, complex schemas are notorious for confusing end users, and a normalized schema requires *masking*this complexity in the presentation area of the data warehouse. Generally, flat dimension tables can appear directly in user interfaces with less confusion.

Having railed against snowflaked dimensions, there are nevertheless some situations where a kind of snowflaking is recommended. These are best described as subdimensions of another dimension. Please refer to the section with this name later in this chapter.

If an attribute takes on multiple values in the presence of the dimension's primary key, the attribute cannot be part of the dimension. For example, in a retail-store dimension, the cash register ID attribute takes on many values for each store. If the grain of the dimension is the individual store, the cash register ID cannot be an attribute in that dimension. To include the cash register attribute, the grain of the dimension must be redeclared to be cash register, not store. But since cash registers roll up to stores in a perfect many-to-1 relationship, the new cash-register dimension contains all of the store attributes, since they are all single valued at the cash-register level.



**Figure 5.2. Flat and snowflaked versions of a dimension.**



**Figure 5.3. Assigning the surrogate key in the dimensionalizing step**

Each time a new dimension record is created, a fresh surrogate key must be assigned. See [Figure 5.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#assigning_the_surrogate_key_in_the_dimen). This meaningless integer is the primary key of the dimension. In a centralized data warehouse environment, the surrogate keys for all dimensions could be generated from a single source. In that case, a master metadata element contains the highest key used for all the dimensions simultaneously. However, even in a highly centralized data warehouse, if there are enough simultaneous ETL jobs running, there could be contention for reading and writing this single metadata element. And of course, in a distributed environment, this approach doesn't make much sense. For these reasons, we recommend that a surrogate key counter be established for each dimension table separately. It doesn't matter whether two different surrogate keys have the same numeric value; the data warehouse will never confuse the separate dimensional domains, and no application ever analyzes the value of a surrogate key, since by definition it is meaningless.

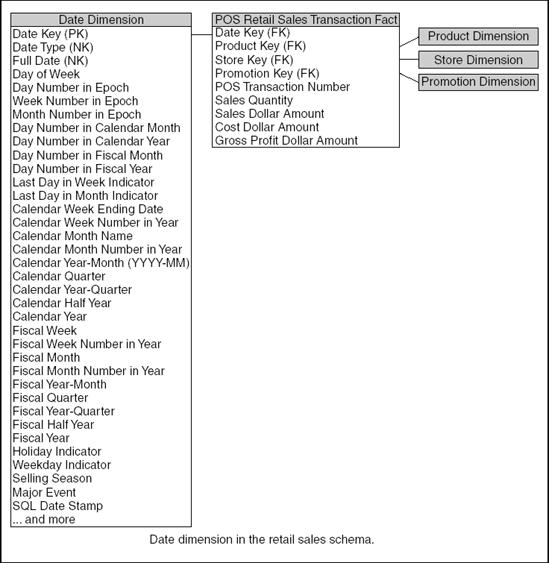
Date and Time Dimensions

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

Virtually every fact table has one or more time-related dimension foreign keys. Measurements are defined at specific points and most measurements are repeated over time.



**Figure 5.4. Attributes needed for a calendar date dimension**

The most common and useful time dimension is the calendar date dimension with the granularity of a single day. This dimension has surprisingly many attributes, as shown in [Figure 5.4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#attributes_needed_for_a_calendar_date_di). Only a few of these attributes (such as month name and year) can be generated directly from an SQL date-time expression. Holidays, work days, fiscal periods, week numbers, last day of month flags, and other navigational attributes must be embedded in the calendar date dimension and all date navigation should be implemented in applications by using the dimensional attributes. The calendar date dimension has some very unusual properties. It is one of the only dimensions completely specified at the beginning of the data warehouse project. It also doesn't have a conventional source. The best way to generate the calendar date dimension is to spend an afternoon with a spreadsheet and build it by hand. Ten years worth of days is fewer than 4000 rows.

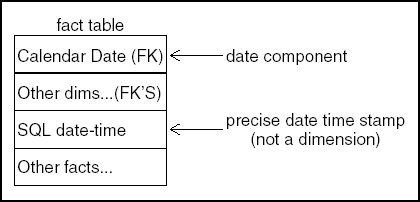
Every calendar date dimension needs a date type attribute and a full date description attribute as depicted in [Figure 5.4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#attributes_needed_for_a_calendar_date_di). These two fields compose the natural key of the table. The date type attribute almost always has the value *date*, but there must be at least one record that handles the special nonappli-cable date situation where the recorded date is inapplicable, corrupted, or hasn't happened yet. Foreign key references in fact tables referring to these special data conditions must point to a nondate date in the calendar date table! You need at least one of these special records in the calendar date table, but you may want to distinguish several of these unusual conditions. For the inapplicable date case, the value of the date type is *inapplicable* or *NA*. The full date attribute is a full relational date stamp, and it takes on the legitimate value of null for the special cases described previously. Remember that the foreign key in a fact table can never be null, since by definition that violates referential integrity.

The calendar date primary key ideally should be a meaningless surrogate key, but many ETL teams can't resist the urge to make the key a readable quantity such as 20040718, meaning July 18, 2004. However, as with all smart keys, the few special records in the time dimension will make the designer play tricks with the smart key. For instance, the smart key for the inapplicable date would have to be some nonsensical value like 99999999, and applications that tried to interpret the date key directly without using the dimension table would always have to test against this value because it is not a valid date.

Even if the primary surrogate key of the calendar date dimension table is a true meaningless integer, we recommend assigning date surrogate keys in numerical order and using a standard starting date for the key value of zero in every date dimension table. This allows any fact table with a foreign key based on the calendar date to be physically partitioned by time. In other words, the oldest data in a fact table could be on one physical medium, and the newest data could be on another. Partitioning also allows the DBA to drop and rebuild indexes on just the most recent data, thereby making the loading process faster, if only yesterday's data is being loaded. Finally, the numeric value of the surrogate key for the special inapplicable time record should probably be a high number so that the inapplicable time-stamped records are in the most active partition. This assumes that these fact records are more likely to be rewritten in an attempt to correct data.

Although the calendar date dimension is the most important time dimension, we also need a calendar month dimension when the fact table's time grain is a month. In some environments, we may need to build calendar week, quarter, or year dimensions as well if there are fact tables at each of these grains. The calendar month dimension should be a separate physical table and should be created by physically eliminating selected rows and columns from the calendar day dimension. For example, either the first or the last day of each month could be chosen from the day dimension to be the basis of the month dimension. It is possible to define a view on a calendar day dimension that implements a calendar month dimension, but this is not recommended. Such a view would drag a much larger table into every month-based query than if the month table were its own physical table. Also, while this view technique can be made to work for calendar dimensions, it cannot be made to work for dimensions like customer or product, since individual customers and products come and go. Thus, you couldn't build a brand table with a view on the base product table, for instance, because you wouldn't know which individual product to choose to permanently represent a brand.

In some fact tables, time is measured below the level of calendar day, down to minute or even second. One cannot build a time dimension with every minute or every second represented. There are more than 31 million seconds in a year! We want to preserve the powerful calendar date dimension and simultaneously support precise querying down to the minute or second. We may also want to compute very precise time intervals by comparing the exact time of two fact table records. For these reasons, we recommend the design shown in [Figure 5.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#fact_table_design_for_handling_precise_t). The calendar day component of the precise time remains as a foreign key reference to our familiar calendar day dimension. But we also embed a full SQL date-time stamp directly in the fact table for all queries requiring the extra precision. Think of this as special kind of fact, not a dimension. In this interesting case, it is not useful to make a dimension with the minutes or seconds component of the precise time stamp, because the calculation of time intervals across fact table records becomes too messy when trying to deal with separate day and time-of-day dimensions. In previous *Toolkit* books, we have recommended building such a dimension with the minutes or seconds component of time as an offset from midnight of each day, but we have come to realize that the resulting end user applications became too difficult when trying to compute time spans that cross daily boundaries. Also, unlike the calendar day dimension, in most environments there are very few descriptive attributes for the specific minute or second within a day.



**Figure 5.5. Fact table design for handling precise time measurements**

If the enterprise does have well-defined attributes for time slices within a day, such as shift names or advertising time slots, an additional time-of-day dimension can be added to the design where this dimension is defined as the number of minutes (or even seconds) past midnight. Thus, this time-ofday dimension would either have 1440 records if the grain were minutes or 86,400 records if the grain were seconds. The presence of such a timeof-day dimension does not remove the need for the SQL date-time stamp described previously.

Big Dimensions

NOTE

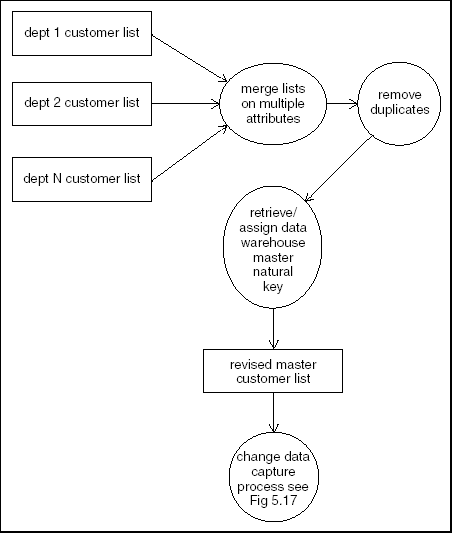
PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

The most interesting dimensions in a data warehouse are the big, wide dimensions such as customer, product, or location. A big commercial customer dimension often has millions of records and a hundred or more fields in each record. A big individual customer record can have tens of millions of records. Occasionally, these individual customer records have dozens of fields, but more often these monster dimensions (for example, grocery store customers identified by a shopper ID) have only a few behaviorally generated attributes.

The really big dimensions almost always are derived from multiple sources. Customers may be created by one of several account management systems in a large enterprise. For example, in a bank, a customer could be created by the mortgage department, the credit card department, or the checking and savings department. If the bank wishes to create a single customer dimension for use by all departments, the separate original customer lists must be de-duplicated, conformed, and merged. These steps are shown in [Figure 5.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#merging_and_de-duplicating_multiple_cust).

In the deduplication step, which is part of the *data-cleaning module*, each customer must be correctly identified across separate original data sources so that the total customer count is correct. A master natural key for the customer may have to be created by the data warehouse at this point. This would be a kind of enterprise-wide customer ID that would stay constant over time for any given customer.



**Figure 5.6. Merging and de-duplicating multiple customer sets**

In the conforming step, which is part of the *data-conforming module*, all attributes from the original sources that try to describe the same aspect of the customer need to be converted into single values used by all the departments. For example, a single set of address fields must be established for the customer. Finally, in the merge (survival) step, which is part of the *delivery-module*, all the remaining separate attributes from the individual source systems are unioned into one big, wide dimension record.

Later in this chapter, when we discuss slowly changing dimensions, we will see that the biggest dimensions are very sensitive to change, if it means that we generate new dimension records for each change. Hold that thought for a moment.

Small Dimensions

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

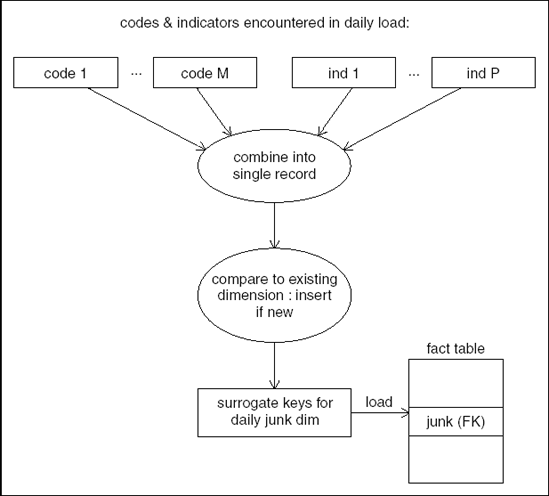
Many of the dimensions in a data warehouse are tiny lookup tables with only a few records and one or two columns. For example, many transactiongrained fact tables have a transaction type dimension that provides labels for each kind of transaction. These tables are often built by typing into a spreadsheet and loading the data directly into the final physical dimension table. The original source spreadsheet should be kept because in many cases new records such as new transaction types could be introduced into the business.

Although a little dimension like transaction type may appear in many different data marts, this dimension cannot and should not be conformed across the various fact tables. Transaction types are unique to each production system.

In some cases, little dimension tables that serve to decode operational values can be combined into a single larger dimension. This is strictly a tactical maneuver to reduce the number of foreign keys in a fact table. Some data sources have a dozen or more operational codes attached to fact table records, many of which have very low cardinalities. Even if there is no obvious correlation between the values of the operational codes, a single *junk dimension* can be created to bring all these little codes into one dimension and tidy up the design. The ETL data flow for a typical junk dimension is shown in [Figure 5.7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#etl_data_flow_for_a_typical_junk_dimensi). The records in the junk dimension should probably be created as they are encountered in the data, rather than beforehand as the Cartesian product of all the separate codes. It is likely that the incrementally produced junk dimension is much smaller than the full Cartesian product of all the values of the codes. The next section extends this kind of junk-dimension reasoning to much larger examples, where the designer has to grapple with the problem of one dimension or two.

One Dimension or Two

In dimensional modeling, we normally assume that dimensions are independent. In a strictly mathematical sense, this is almost never true. Although you may sell many products in many stores, the product dimension and the store dimension are probably not truly independent. Some products are sold in only selected stores. A good statistician would be able to demonstrate a degree of correlation between the product dimension and the store dimension. But such a finding normally does not deter us from creating separate product and store dimensions. The correlation that does exist between these dimensions can be faithfully and accurately depicted in the sales fact table.



**Figure 5.7. ETL data flow for a typical junk dimension**

Modeling the product dimension with the store dimension in this example would be a disaster. If you had a million-row product dimension and a 100-row store dimension, a combined dimension might approach 100 million rows! Bookkeeping the cross-correlations between dimensions solely in the fact table is an example of a powerful dimensional-modeling step:*demoting the correlations between dimensions* into a fact table.

A final nail in the coffin for combining product and store is that there may be more than one independent type of correlation between these two dimensions. We have discussed the merchandising correlation between these two dimensions, but there could be a pricing-strategy correlation, a warehousing correlation, or a changing-seasonality correlation. In general, tracking all of these complex relationships must be handled by leaving the dimensions simple and independent and by bookkeeping the cross-dimensional relationships in one or more fact tables.

At this point, you may be convinced that all dimensions can be independent and separate. But that's because we have been discussing a somewhat extreme example of two big dimensions where the correlation is statistically weak. Are other situations not so black and white?

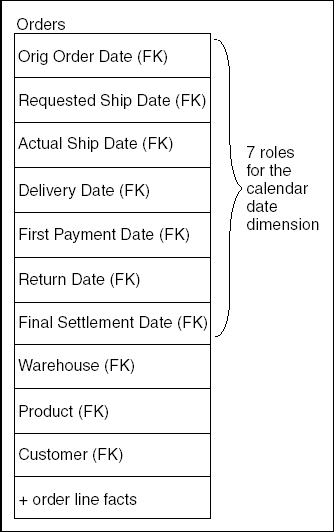
First, let us immediately dispense with the completely correlated overlap of two dimensions. We should never have a single fact table with both a product dimension and a brand dimension if product rolls up to brand in a perfect many-to-1 relationship. In this case, product and brand are part of a hierarchy, and we should always combine these into a single dimension.

There are other cases where two candidate dimensions do not form a perfect hierarchy but are strongly correlated. To jump to the bottom line, if the correlation is reasonably high and the resulting combined dimension is reasonably small, the two dimensions should be combined into one. Otherwise, the dimensions should be left separate. The test for a reasonably high correlation should be made from the end user's perspective. If the pattern of overlap between the two dimensions is interesting to end users and is constant and unchanging, the combined dimension may be attractive. Remember that the combined dimension in this case serves as an efficient target for queries, independent of any fact table. In our opinion, a dimension is no longer reasonably small when it becomes larger than 100,000 rows. Over time, perhaps technology will relax this arbitrary boundary, but in any case a 100,000 row dimension will always present some user-interface challenges!

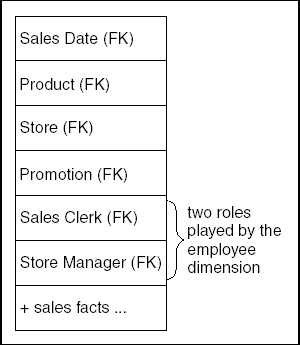
Dimensional Roles

The data warehouse architect will frequently specify a dimension to be attached multiple times to the same fact table. These are called *dimensional roles*. Probably the most common role-playing dimension is the calendar date dimension. Many fact tables, especially the accumulating snapshot fact tables, have multiple date foreign keys. We discuss accumulating snapshot fact tables in [Chapter 6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html). See [Figure 5.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_typical_accumulating_snapshot_fact_tab). Another common example of a roleplaying dimension is the employee dimension, where different foreign keys in the fact table represent different types of employees being involved in a single transaction. See [Figure 5.9](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#two_employee_role_playing_dimensions).

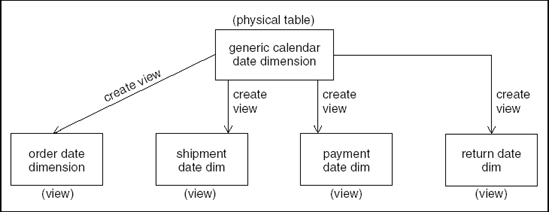
In all role-playing dimension implementations, we recommend first building a generic single dimension table and then implementing each of the roles with a view on this generic table. See [Figure 5.10](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#multiple_calendar_role_playing_dimension). For instance, if we have an order-date, a shipment-date, a payment-date, and a return-date on an orders transaction accumulating snapshot fact table, we would first build a generic calendar date dimension and the create four views corresponding to the four dates needed. If the fields in each view are identically named, the application developer and possibly the end user will need to see the fully qualified names to distinguish similar fields from the different views in the same query. For that reason, we recommend creating distinguishable field names in the original view definitions so that every tool, even those not supported by metadata, will display the fields unambiguously.



**Figure 5.8. A typical accumulating snapshot fact table**



**Figure 5.9. Two employee role playing dimensions**



**Figure 5.10. Multiple calendar role playing dimensions**

NOTE

The recommended design of dimensional roles described previously makes the impact of dimensional roles on the ETL team equal to zero. So why do we discuss it? Our objective is to make sure the ETL team doesn't generate multiple physical tables in cases where view definitions (roles) accomplish the same purpose.

NOTE

Don't use the dimensional-role techniques as an excuse to build abstract, super-large dimensions. For instance, in a telco environment, nearly everything has a location. If every possible location of every entity is represented in a single location dimension, this dimension could have millions of rows. Using a view on a multimillion row dimension in every application with a location dimension is probably a performance killer. In this case, actual physical dimensions created as extracted subsets of the big location dimension are probably better.

Dimensions as Subdimensions of Another Dimension

NOTE

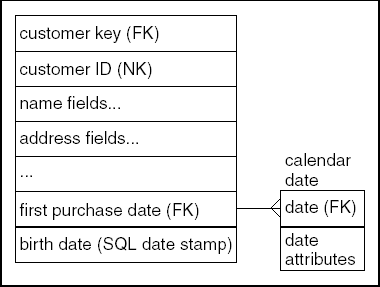
PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

PROCESS CHECK **Planning & Design:**

Requirements/Realities → Architecture → *Implementation* → Test/Release

Usually, we think of a reference to a dimension as a foreign key in a fact table. However, references to dimensions occasionally appear in other dimensions, and the proper foreign key should be stored in the parent dimension in the same way as a fact table. In other writings, we have sometimes referred to these subdimensions as *outriggers*. Let's discuss two common examples.



**Figure 5.11. Customer dimension showing two date treatments**

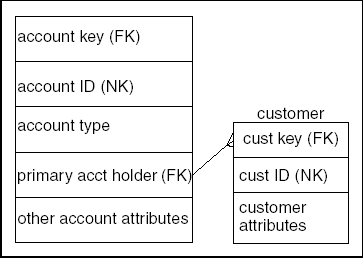
Many dimensions have calendar dates embedded in them. Customer dimension records often have a first purchase date attribute. This should be modeled as a foreign key reference to the calendar date dimension, not as an SQL date stamp. See [Figure 5.11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#customer_dimension_showing_two_date_trea). In this way, applications have access to all the extended calendar attributes when constraining on the first purchase date. This foreign key reference is really another kind of dimensional role played by the calendar date dimension. A separate view, in this case on the calendar date dimension, must be defined for each such reference.

Note that not all dates stored in dimensions can be modeled as foreign key references to the calendar date dimension, since the calendar date dimension has a bounded duration. A customer's birth date may well precede the first entry in the calendar date dimension. If that could happen, the customer birth date attribute must always be a simple SQL date stamp, not a foreign key reference. This is also shown in [Figure 5.11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#customer_dimension_showing_two_date_trea).

A second common example of a dimension attached to a dimension is attaching an individual customer dimension to a bank account dimension. Although there might be many account holders in an account, usually a single customer is designated as the primary account holder. This primary account holder should be modeled as a foreign key reference in the account dimension to the customer dimension. See [Figure 5.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_customer_dimension_used_as_a_subdimens).

In this banking example, we have not handled the problem of many customers being associated with an account. We have dealt only with the single primary account holder customer. We will associate an open-ended number of customers to an account later in this chapter when we discuss multivalued dimensions and bridge tables.

To summarize this section, the ETL dimensional delivery module must convert selected fields in the input data for the dimension to foreign key references.



**Figure 5.12. A customer dimension used as a subdimension**

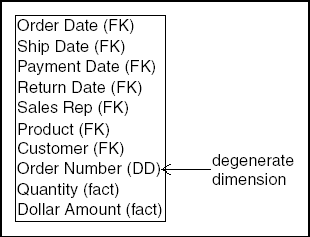
Degenerate Dimensions

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

Whenever a parent-child data relationship is cast in a dimensional framework, the natural key of the parent is left over as an orphan in the design process. For example, if the grain of a fact table is the line item on an order, the dimensions of that fact table include all the dimensions of the line itself, as well as the dimensions of the surrounding order. Remember that we attach all single-valued dimensional entities to any given fact table record. When we have attached the customer and the order date and other dimensions to the design, we are left with the original order number. We insert the original order number directly into the fact table as if it were a dimension key. See [Figure 5.13](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#an_order_line_accumulating_snapshot_fact). We could have made a separate dimension out of this order number, but it would have turned out to contain only the order number, nothing else. For this reason, we give this natural key of the parent a special status and call it a *degenerate* (or empty) dimension. This situation arises in almost every parent-child design, including order numbers, shipment numbers, bill-of-lading numbers, ticket numbers, and policy numbers.



**Figure 5.13. An order line accumulating snapshot fact table**

NOTE

There is a danger that these source-system-generated numbers can get reused by different ERP instances installed in separate business units of an overall organization. For this reason, it may be a good idea to make a smart degenerate key value in these cases by prepending an organization ID onto the basic order number or sales ticket number.

Slowly Changing Dimensions

NOTE

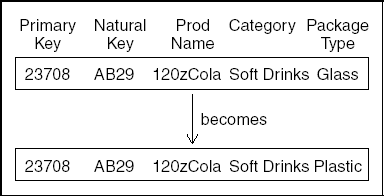
PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

When the data warehouse receives notification that an existing row in a dimension has in some way changed, there are three basic responses. We call these three basic responses Type 1, Type 2, and Type 3 slowly changing dimensions (SCDs).

Type 1 Slowly Changing Dimension (Overwrite)

The Type 1 SCD is a simple overwrite of one or more attributes in an existing dimension record. See [Figure 5.14](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#processing_a_type_1_scd). The ETL processing would choose the Type 1 approach if data is being corrected or if there is no interest in keeping the history of the previous values and no need to run prior reports. The Type 1 overwrite is always an UPDATE to the underlying data, and this overwrite must be propagated forward from the earliest permanently stored staging tables in the ETL environment so that if any of them are used to recreate the final load tables, the effect of the overwrite is preserved. This point is expanded in [Chapter 8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch08.html).



**Figure 5.14. Processing a Type 1 SCD**

Although inserting new records into a Type 1 SCD requires the generation of new dimension keys, processing changes in a Type 1 SCD never affects dimension table keys or fact table keys and in general has the smallest impact on the data of the three SCD types. The Type 1 SCD can have an effect on the storage of aggregate fact tables, if any aggregate is built directly on the attribute that was changed. This issue will be described in more detail in [Chapter 6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html).

NOTE

Some ETL tools contain *UPDATE else INSERT* functionality. This functionality may be convenient for the developer but is a performance killer. For maximum performance, existing records (UPDATEs) should be segregated from new ones (INSERTs) during the ETL process; each should be fed to the data warehouse independently. In a Type 1 environment, you may not know whether an incoming record is an UPDATE or an INSERT. Some developers distinguish between a VERY SCD (very slowly changing dimension) where INSERTs predominate and a Fastly Changing Dimension (FCD?). They use INSERT else UPDATE logic for VERY SCDs and UPDATE else INSERT logic for the FCDs. We hope this terminology doesn't catch on.

In most data warehouse implementations, the size of the majority of dimensions is insignificant. When you are loading small tables that do not warrant the complexity of invoking a bulk loader, Type 1 changes can be applied via normal SQL DML statements. Based on the natural key extracted from the source system, any new record is assigned a new surrogate key and appended to the existing dimension data. Existing records are updated in place. Performance of this technique may be poorer as compared with being loaded via a bulk loader, but if the tables are of reasonable size, the impact should be negligible.

Some ETL tools offer specialized transformations that can detect whether a record needs to be inserted or updated. However, this utility must *ping* the table using the primary key of the candidate record to see if it exists. This approach is process intensive and should be avoided. To minimize the performance hit when using SQL to load a Type 1 dimension, the ETL process should explicitly segregate existing data that requires UPDATE statements from data that requires INSERT.

NOTE

Type 1 SCD changes can cause performance problems in ETL processing. If this technique is implemented using SQL data-manipulation language (DML), most database management systems will *log* the event, hindering performance.

A database log is implicitly created and maintained by the DBMS. Database logging is constructive for transaction processing where data is entered by many users in an uncontrolled fashion. *Uncontrolled* is used because in the on-line transaction processing (OLTP) environment, there is no way to control unpredicted user behavior, such as closing a window midway through an update. The DBMS may need to ROLLBACK, or undo, a failed update. The database log enables this capability.

Conversely, in the data warehouse, all data loading is controlled by the ETL process. If the process fails, the ETL process should have the capability to recover and pick-up where it left off, making the database log superfluous.

With database logging enabled, large dimensions will load at an unacceptable rate. Some database management systems allow you to turn logging off during certain DML processes, while others require their bulk loader to be invoked for data to be loaded without logging.

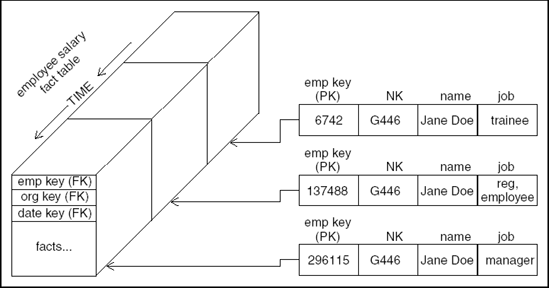
Bulk Loading Type 1 Dimension Changes

Because Type 1 overwrites data, the easiest implementation technique is to use SQL UPDATE statements to make all of the dimension attributes correctly reflect the current values. Unfortunately, as a result of database logging, SQL UPDATE is a poor-performing transaction and can inflate the ETL load window. For very large Type 1 changes, the best way to reduce DBMS overhead is to employ its bulk loader. Prepare the new dimension records in a separate table. Then drop the records from the dimension table and reload them with the bulk loader.

Type 2 Slowly Changing Dimension (Partitioning History)

The Type 2 SCD is the standard basic technique for accurately tracking changes in dimensional entities and associating them correctly with fact tables. The basic idea is very simple. When the data warehouse is notified that an existing dimension record needs to be changed, rather than overwriting, the data warehouse issues a new dimension record at the moment of the change. This new dimension record is assigned a fresh surrogate primary key, and that key is used from that moment forward in all fact tables that have that dimension as a foreign key. As long as the new surrogate key is assigned promptly at the moment of the change, no existing keys in any fact tables need to be updated or changed, and no aggregate fact tables need to be recomputed. The more complex case of handling late-arriving notifications of changes is described later in this chapter.

We say that the Type 2 SCD *perfectly partitions history* because each detailed version of a dimensional entity is correctly connected to the span of fact table records for which that version is exactly correct. In [Figure 5.15](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#the_type_2_scd_perfectly_partitions_hist), we illustrate this concept with a slowly changing employee dimension where a particular employee named Jane Doe is first a trainee, then a regular employee, and finally a manager. Jane Doe's natural key is her employee number and that remains constant throughout her employment. In fact, the natural key field always has the unique business rule that it cannot change, whereas every other attribute in the employee record can change. Jane Doe's primary surrogate key takes on three different values as she is promoted, and these surrogate primary keys are always correctly associated with contemporary fact table records. Thus, if we constrain merely on the employee Jane Doe, perhaps using her employee number, we pick up her entire history in the fact table because the database picks up all three surrogate primary keys from the dimension table and joins them all to the fact table. But if we constrain on *Jane Doe, manager*, we get only one surrogate primary key and we see only the portion of the fact table for which Jane Doe was a manager.



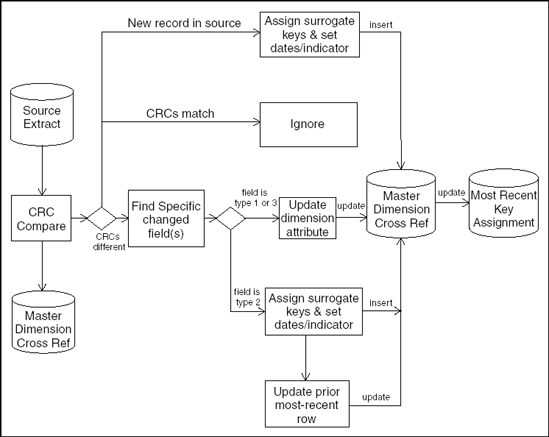
**Figure 5.15. The Type 2 SCD perfectly partitions history**

NOTE

If the natural key of a dimension can change, from the data warehouse's point of view, it isn't really a natural key. This might happen in a credit card processing environment where the natural key is chosen as the card number. We all know that the card number can change; thus, the data warehouse is required to use a more fundamental natural key. In this example, one possibility is to use the original customer's card number forever as the natural key, even if it subsequently changes. In such a design, the customer's current contemporary card number would be a separate field and would not be designated as a key.

The Type 2 SCD requires a good *change data capture system* in the ETL environment. Changes in the underlying source data need to be detected as soon as they occur, so that a new dimension record in the data warehouse can be created. We discuss many of the issues of change data capture at extract time in [Chapter 3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch03.html). In the worst scenario, the underlying source system does not notify the data warehouse of changes and does not date-stamp its own updates. In this case, the data warehouse is forced to download the complete dimension and look record by record and field by field for changes that have occurred since the last time the dimension was downloaded from the source. Note that this requires the prior extract (the master dimension cross reference file) from the dimension's source to be explicitly staged in the ETL system. See [Figure 5.16](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#dimension_table_surrogate_key_management).

For a small dimension of a few thousand records and a dozen fields, such as a simple product file, the detection of changes shown in [Figure 5.16](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#dimension_table_surrogate_key_management) can be done by brute force, comparing every field in every record in today's download with every field in every record from yesterday. Additions, changes, and deletions need to be detected. But for a large dimension, such as a list of ten million insured health care patients with 100 descriptive fields in each record, the brute-force approach of comparing every field in every record is too inefficient. In these cases, a special code known as a CRC is computed and associated with every record in yesterday's data. The CRC (cyclic redundancy checksum) code is a long integer of perhaps 20 digits that is exquisitely sensitive to the information content of each record. If only a single character in a record is changed, the CRC code for that record will be completely different. This allows us to make the change data capture step much more efficient. We merely compute the CRC code for each incoming new record by treating the entire record as a single text string, and we compare that CRC code with yesterday's code for the same natural key. If the CRCs are the same, we immediately skip to the next record. If the CRCs are different, we must stop and compare each field to find what changed. The use of this CRC technique can speed up the change data capture process by a factor of 10. At the time of this writing, CRC calculation modules are available from all of the leading ETL package vendors, and the code for implementing a CRC comparison can be readily found in textbooks and on the Internet.



**Figure 5.16. Dimension table surrogate key management**

Once a changed dimension record has been positively identified, the decision of which SCD type is appropriate can be implemented. Usually, the ETL system maintains a policy for each column in a dimension that determines whether a change in that attribute triggers a Type 1, 2, or 3 response, as shown in [Figure 5.16](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#dimension_table_surrogate_key_management).

NOTE

To identify records deleted from the source system, you can either read the source transaction log file (if it is available) or note that the CRC comparison step described previously cannot find a record to match a natural key in the ETL system's comparison file. But in either case, an explicit business rule must be invoked to deal with the deletion. In many cases, the deleted entity (such as a customer) will have a continuing presence in the data warehouse because the deleted entity was valid in the past. If the business rule conclusively states that the deleted entity can no longer appear in subsequent loads from the dimension-table source, the deleted entity can be removed from the daily comparison step, even though in the historical dimension tables and fact tables it will live on.

NOTE

Without transaction log files, checking for deleted data is a process- intensive practice and usually is implemented only when it is demanded. An option that has proven to be effective is utilizing the MINUS set operator to compare the natural keys from the dimension in the data warehouse against the natural keys in the source system table. UNION and MINUS are SET operators supported by most database management systems used to compare two sets of data. These operators are extremely powerful for evaluating changes between the source and target. However, for these SET operators to work, the two tables need to be in the same database, or a database link must be created. Some ETL tools support SET operations between heterogeneous systems. If this is a critical requirement for your environment, be sure it is included in your proof-of-concept criteria when selecting your ETL toolset.

Notice in [Figure 5.16](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#dimension_table_surrogate_key_management) that when we have created the new surrogate key for a changed dimension entity, we update a two-column lookup table, known as the *most recent key lookup table* for that dimension. These little tables are of immense importance when loading fact table data. Hold this thought until you read the surrogate key pipeline section in [Chapter 6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html).

NOTE

The same benefits that the lookup-table solution offers can be accomplished by storing all of the relevant natural keys directly in the dimension table. This approach is probably the most common for determining whether natural keys and dimension records have been loaded. This approach makes the associated natural keys available to the users, right in the dimension. The major benefit of this strategy over the lookup table is that the surrogate key exists only in one place, eliminating the risk of the dimension and the mapping table becoming out of sync. During the ETL, the process selects the natural key from the appropriate column within the dimension where it equals the incoming natural key. If a match is found, the process can apply any of the SCD strategies described later in this chapter. If the key is not found, it can generate a surrogate key using any of the methods discussed in the next section and insert a new record.

Looking directly to dimensions is favored by many data warehouse designers because it exposes the data lineage to users. By having the natural keys directly in the dimension, users know exactly where the data in the dimension came from and can verify it in the source system. Moreover, natural keys in the dimension relieve the ETL and DBA teams from having to maintain a separate mapping table for this purpose. Finally, this approach makes a lot of sense in environments where there is a large fraction of late-arriving data where the *most*recent advantages of a lookup table cannot be used.

In this section, we have described a change-data-capture scenario in which the data warehouse is left to guess if a change occurred in a dimension record and why. Obviously, it would be preferable if the source system handed over only the changed records (thereby avoiding the complex comparison procedure described previously) and ideally accompanied the changed records with *reason codes* that distinguished the three SCD responses. What a lovely dream.

Our approach allows us to respond to changes in the source for a dimension as they occur, even when the changes are not marked. A more difficult situation takes place when a database is being loaded for the first time from such an uncooperative source. In this case, if the dimension has been overwritten at the source, it may be difficult to reconstruct the historical changes that were processed unless original transaction log files are still available.

Precise Time Stamping of a Type 2 Slowly Changing Dimension

The discussion in the previous section requires only that the ETL system generate a new dimension record when a change to an existing record is detected. The new dimension record is correctly associated with fact table records automatically because the new surrogate key is used promptly in all fact table loads after the change takes place. No date stamps in the dimension are necessary to make this correspondence work.

Having said that, it is desirable in many situations to *instrument* the dimension table to provide optional useful information about Type 2 changes. We recommend adding the following five fields to dimension tables processed with Type 2 logic:

* Calendar Date foreign key (date of change)
* Row Effective DateTime (exact date-time of change)
* Row End DateTime (exact date-time of next change)
* Reason for Change (text field)
* Current Flag (current/expired)

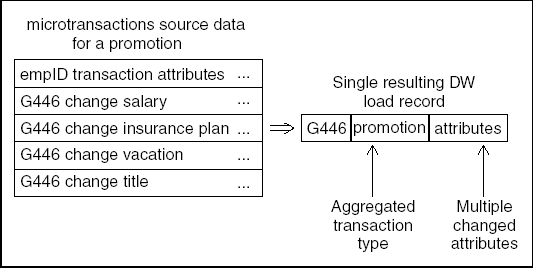
These five fields make the dimension a powerful query target by itself, even if no fact table is mentioned in the query. The calendar date foreign key allows an end user to use the business calendar (with seasons, holidays, paydays, and fiscal periods) to ask how many changes of a particular type were made in certain business-significant periods. For instance, if the dimension is a human resources employee table, one could ask how many promotions occurred in the last fiscal period.

The two SQL date-time stamps define an exact interval in which the current dimension record correctly and completely describes the entity. When a new change is processed, the Row Effective DateTime is set to the current date and time, and the Row End DateTime is set to an arbitrary time far in the future. When a subsequent change to this dimension entity is processed, the previous record must be revisited and the Row End DateTime set to the proper value. If this procedure is followed, the two date-time stamps always define an *interval of relevance* so that queries can specify a random specific date and time and use the SQL BETWEEN logic to immediately deliver records that were valid at that instant. We need to set the Row End DateTime to a real value, even when the record is the most current, so that the BETWEEN logic doesn't return an error if the second date is represented as null.

NOTE

It is seems to be universal practice for back-end scripts to be run within the transaction database to modify data without updating respective metadata fields, such as the last\_modified\_date. Using these fields for the dimension row effective\_datetime will cause inconsistent results in the data warehouse. Do not depend on metadata fields in the transaction system. Always use the system or *as of* date to derive the row effective\_datetime in a Type 2 slowly changing dimension.

The Reason for Change field probably needs to come from the original data-entry process that gave rise to the changed dimension record. For instance, in the human resources example, you would like a promotion to be represented as a single new record, appropriately time stamped, in the employee dimension. The Reason for Change field should say *promotion*. This may not be as easy as it sounds. The HR system may deliver a number of change records at the time of an employee's promotion if several different employee attributes (job grade, vacation benefits, title, organization and so on) change simultaneously. The challenge for the data warehouse is to coalesce these changes into a single new dimension record and correctly label this new record with *promotion*. Such a coalescing of underlying transaction records into a kind of aggregated *super-transaction* may be necessary with some source systems, even if no attempt is made to ascribe a reason code to the overall change. We have seen relatively simple updates such as employee promotions represented by dozens of micro transactions. The data warehouse should not carry these microtransactions all the way to the final end user tables, because the individual microtransactions may not have real business significance. This processing is depicted in [Figure 5.17](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#consolidating_source_system_microtransac). Finally, the Current Flag is simply a convenient way to retrieve all the most-current records in a dimension. It is indeed redundant with the two SQL date-time stamps and therefore can be left out of the design. This flag needs to be set to EXPIRED when a superceding change to the dimension entity takes place.

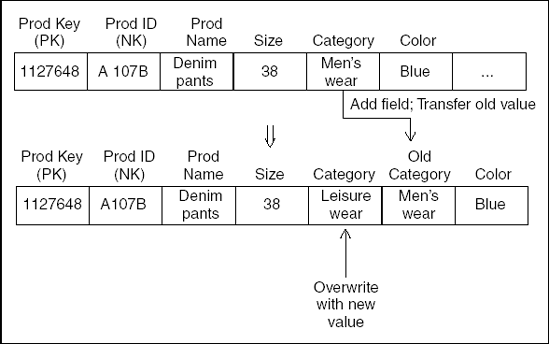


**Figure 5.17. Consolidating source system microtransactions**

Type 3 Slowly Changing Dimension (Alternate Realities)

The Type 3 SCD is used when a change happens to a dimension record but the old value of the attribute remains valid as a second choice. The two most common business situations where this occurs are changes in salesterritory assignments, where the old territory assignment must continue to be available as a second choice, and changes in product-category designations, where the old category designation must continue to be available as a second choice. The data warehouse architect should identify fields that require Type 3 administration.

In a Type 3 SCD, instead of issuing a new row when a change takes place, a new column is created (if it does not already exist), and the old value is placed in this new field before the primary value is overwritten. For the example of the product category, we assume the main field is named Category. To implement the Type 3 SCD, we alter the dimension table to add the field Old Category. At the time of the change, we take the original value of Category and write it into the Old Category field; then we overwrite the Category field as if it were a Type 1 change. See [Figure 5.18](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#implementing_the_type_3_scd_for_a_produc). No keys need to be changed in any dimension table or in any fact table. Like the Type 1 SCD, if aggregate tables have been built directly on the field undergoing the Type 3 change, these aggregate tables need to be recomputed. This procedure is described in [Chapter 6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html).



**Figure 5.18. Implementing the Type 3 SCD for a product-category description**

NOTE

Type 3 changes often do not come down the normal data-flow pipeline. Rather, they are executive decisions communicated to the ETL team, often verbally. The product-category manager says, "Please move Brand X from Mens Sportswear to Leather Goods, but let me track Brand X optionally in the old category." The Type 3 administration is then kicked off by hand, and can even involve a schema change, if the changed attribute (in this case, *brand*) does not have an *alternate* field.

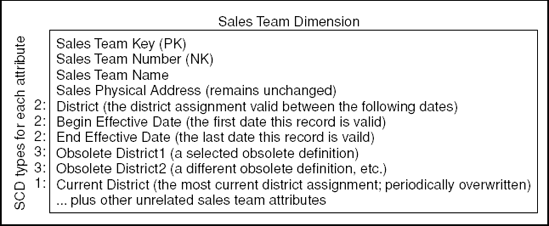
When a new record is added to a dimension that contains Type 3 fields, a business rule must be invoked to decide how to populate the old value field. The current value could be written into this field, or it could be NULL, depending on the business rule.

We often describe the Type 3 SCD as supporting an *alternate reality*. In our product-category example, the end user could choose between two versions of the mapping of products to categories.

The Type 3 SCD approach can be extended to many alternate realities by creating an arbitrary number of alternate fields based on the original attribute. Occasionally, such a design is justified when the end user community already has a clear vision of the various interpretations of reality. Perhaps the product categories are regularly reassigned but the users need the flexibility to interpret any span of time with any of the category interpretations. The real justification for this somewhat awkward design is that the user interface to this information *falls out* of every query tool with no programming, and the underlying SQL requires no unusual logic or extra joins. These advantages trump the objections to the design using positionally dependent attributes (the alternate fields).

Hybrid Slowly Changing Dimensions

The decision to respond to changes in dimension attributes with the three SCD types is made on a field-by-field basis. It is common to have a dimension containing both Type 1 and Type 2 fields. When a Type 1 field changes, the field is overwritten. When a Type 2 field changes, a new record is generated. In this case, the Type 1 change needs to be made to all copies of the record possessing the same natural key. In other words, if the ethnicity attribute of an employee profile is treated as a Type 1, if it is ever changed (perhaps to correct an original erroneous value), the ethnicity attribute must be overwritten in all the copies of that employee profile that may have been spawned by Type 2 changes.



**Figure 5.19. A hybrid SCD showing all three types**

It is possible to combine all the SCD types in a single dimension record. See [Figure 5.19](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_hybrid_scd_showing_all_three_types). In this example, the district assignment field for the sales team is a Type 2 attribute. Whenever the district assignment changes, a new record is created, and the beginning and effective dates are set appropriately. The set of yearly old district assignments are Type 3 fields, implementing many alternate realities. And finally, the current district assignment is a Type 1 field, and it is overwritten in all copies of the sales team dimension records whenever the current district is reassigned.

Late-Arriving Dimension Records and Correcting Bad Data

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

Late-arriving data may need to be extracted via a different application or different constraints compared to normal contemporary data. Bad data obviously is picked up in the data-cleaning step.

A late-arriving dimension record presents a complex set of issues for the data warehouse. Suppose that we have a fictitious product called Zippy Cola. In the product dimension record for Zippy Cola 12-ounce cans, there is a formulation field that has always contained the value *Formula A*. We have a number of records for Zippy Cola 12-ounce cans because this is a Type 2 slowly changing dimension and other attributes like the package type and the subcategory for Zippy Cola 12-ounce cans have changed over the past year or two.

Today we are notified that on July 15, 2003 (a year ago) the formulation of Zippy Cola 12-ounce cans was changed to *Formula B* and has been Formula B ever since. We should have processed this change a year ago, but we failed to do so. Fixing the information in the data warehouse requires the following steps:

1. Insert a fresh new record with a new surrogate key for Zippy Cola 12-ounce cans into the Product dimension with the formulation field set to *Formula B*, the row effective datetime set to July 15, 2003, and the row end datetime set to the row effective datetime of the next record for Zippy Cola in the product dimension table. We also need to find the closest previous dimension record for Zippy Cola and set its row end datetime to the datetime of our newly inserted record. Whew!
2. Scan forward in the Product dimension table from July 15, 2003, finding any other records for Zippy Cola 12-ounce cans, and destructively overwrite the formulation field to *Formula B* in all such records.
3. Find all fact records involving Zippy Cola 12-ounce cans from July 15, 2003, to the first next change for that product in the dimension after July 15, 2003, and destructively change the Product foreign key in those fact records to be the new surrogate key created in Step 1.

NOTE

Updating fact table records (in Step 3) is a serious step that should be tested carefully in a test environment before performing it on the production system. Also, if the update is protected by a database transaction, be careful that some of your updates don't involve an astronomical number of records. For operational purposes, such large updates should be divided into chunks so that you don't waste time waiting for an interrupted update of a million records to roll back.

There are some subtle issues here. First, we need to check to see if some other change took place for Zippy Cola 12-ounce cans on July 15, 2003. If so, we need only to perform Step 2. We don't need a new dimension record in this special case.

In general, correcting bad data in the data warehouse can involve the same logic. Correcting a Type 1 field in a dimension is simplest because we just have to overwrite all instances of that field in all the records with the desired natural key. Of course, aggregate tables have to be recalculated if they have specifically been built on the affected attribute. Please see the aggregate updating section for fact tables in [Chapter 6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html). Correcting a Type 2 field requires thoughtful consideration, since it is possible that the incorrect value has a specific time span.

This discussion of late-arriving dimension records is really about late-arriving *versions* of dimension records. In real-time systems (discussed in [Chapter 11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch11.html)), we deal with true late-arriving dimension records that arrive after fact records have already been loaded into the data warehouse. In this case, the surrogate key in the fact table must point to a special temporary placeholder in the dimension until the real dimension record shows up.

Multivalued Dimensions and Bridge Tables

NOTE

PROCESS CHECK Planning & Design:

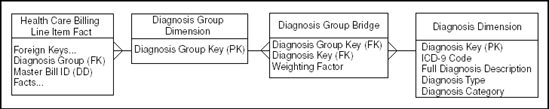
Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

Occasionally a fact table must support a dimension that takes on multiple values at the lowest level of granularity of the fact table. Examples described in the other Toolkit books include multiple diagnoses at the time of a billable health care treatment and multiple account holders at the time of a single transaction against a bank account.

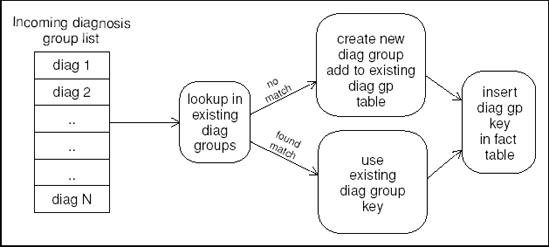
If the grain of the fact table is not changed, a multivalued dimension must be linked to the fact table through an associative entity called a *bridge table*. See [Figure 5.20](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#using_a_bridge_table_to_represent_multip) for the health care example.

To avoid a many-to-many join between the bridge table and the fact table, one must create a *group entity* related to the multivalued dimension. In the health care example, since the multivalued dimension is diagnosis, the group entity is diagnosis group. The diagnosis group becomes the actual normal dimension to the fact table, and the bridge table keeps track of the many-to-many relationship between diagnosis groups and diagnoses. In the bank account example, when an account activity record is linked to the multivalued customer dimension (because an account can have many customers), the group entity is the familiar account dimension.

The challenge for the ETL team is building and maintaining the group entity table. In the health care example, as patient-treatment records are presented to the system, the ETL system has the choice of either making each patient's set of diagnoses a unique diagnosis group or reusing diagnosis groups when an identical set of diagnoses reoccurs. There is no simple answer for this choice. In an outpatient setting, diagnosis groups would be simple, and many of the same ones would appear with different patients. In this case, reusing the diagnosis groups is probably the best choice. See [Figure 5.21](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#processing_diagnosis_groups_in_an_outpat). But in a hospital environment, the diagnosis groups are far more complex and may even be explicitly time varying. In this case, the diagnosis groups should probably be unique for each patient and each hospitalization. See [Figure 5.22](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_time-varying_diagnosis_group_bridge_ta) and the discussion of time-varying bridge tables that follows. The admission and discharge flags are convenient attributes that allow the diagnosis profiles at the time of admission and discharge to be easily isolated.



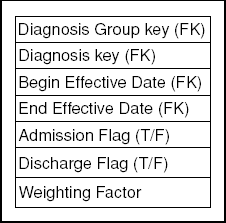
**Figure 5.20. Using a bridge table to represent multiple diagnoses**



**Figure 5.21. Processing diagnosis groups in an outpatient setting**

Administering the Weighting Factors

The diagnosis group tables illustrated in [Figure 5.20](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#using_a_bridge_table_to_represent_multip) and [Figure 5.22](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_time-varying_diagnosis_group_bridge_ta) include weighting factors that explicitly prorate the additive fact (charge dollars) by each diagnosis. When a requesting query tool constrains on one or more diagnoses, the tool can chose to multiply the weighting factor in the bridge table to the additive fact, thereby producing a *correctly weighted* report. A query without the weighting factor is referred to as an *impact* report. We see that the weighting factor is nothing more than an explicit allocation that must be provided in the ETL system. These allocations are either explicitly fetched from an outside source like all other data or can be simple computed fractions depending on the number of diagnoses in the diagnosis group. In the latter case, if there are three diagnoses in the group, the weighting factor is 1/3 = 0.333 for each diagnosis.



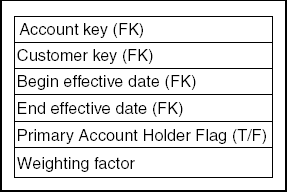
**Figure 5.22. A time-varying diagnosis group bridge table appropriate for a hospital setting**

In many cases, a bridge table is desirable, but there is no rational basis for assigning weighting factors. This is perfectly acceptable. The user community in this case cannot expect to produce correctly weighted reports. These front-room issues are explored in some depth in the *Data Warehouse Toolkit, Second Edition* in the discussion of modeling complex events like car accidents.

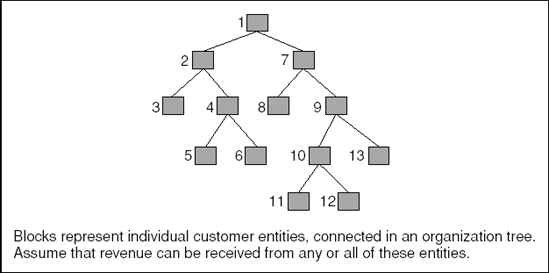
Time-Varying Bridge Tables

If the multivalued dimension is a Type 2 SCD, the bridge table must also be time varying. See [Figure 5.23](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_time-varying_bridge_table_for_accounts) using the banking example. If the bridge table were not time varying, it would have to use the natural keys of the customer dimension and the account dimension. Such a bridge table would potentially misrepresent the relationship between the accounts and customers. It is not clear how to administer such a table with natural keys if customers are added to or deleted from an account. For these reasons, the bridge table must always contain surrogate keys. The bridge table in [Figure 5.23](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_time-varying_bridge_table_for_accounts) is quite sensitive to changes in the relationships between accounts and customers. New records for a given account with new begin-date stamps and end-date stamps must be added to the bridge table whenever:

* The account record undergoes a Type 2 update
* Any constituent customer record undergoes a Type 2 update
* A customer is added to or deleted from the account or
* The weighting factors are adjusted



**Figure 5.23. A time-varying bridge table for accounts and customers**



**Figure 5.24. A representative ragged organization hierarchy**

Ragged Hierarchies and Bridge Tables

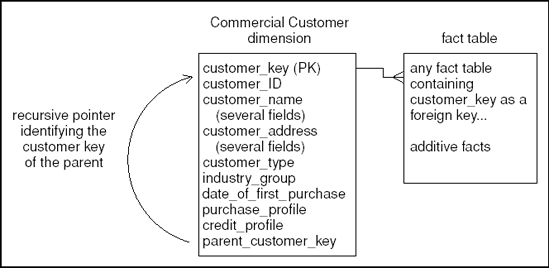
NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow: Extract → Clean → Conform → *Deliver*

Ragged hierarchies of indeterminate depth are an important topic in the data warehouse. Organization hierarchies such as depicted in [Figure 5.24](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_representative_ragged_organization_hie) are a prime example. A typical organization hierarchy is unbalanced and has no limits or rules on how deep it might be.

There are two main approaches to modeling a ragged hierarchy, and both have their pluses and minuses. We'll discuss these tradeoffs in terms of the customer hierarchy shown in [Figure 5.24](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_representative_ragged_organization_hie).



**Figure 5.25. A customer dimension with a recursive pointer**

The *recursive pointer* approach shown in [Figure 5.25](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_customer_dimension_with_a_recursive_po).

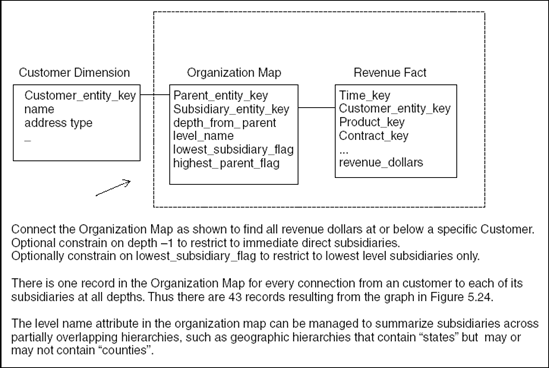
* (+) embeds the hierarchy relationships entirely in the customer dimension
* (+) has simple administration for adding and moving portions of the hierarchy

but

* (-) requires nonstandard SQL extensions for querying and may exhibit poor query performance when the dimension is joined to a large fact table
* (-) can only represent simple trees where a customer can have only one parent (that is, disallowing shared ownership models)
* (-) cannot support switching between different hierarchies
* (-) is very sensitive to time-varying hierarchies because the entire customer dimension undergoes Type 2 changes when the hierarchy is changed

The *hierarchy bridge table* approach shown in [Figure 5.26](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_hierarchy_bridge_table_representing_cu):

* (+) isolates the hierarchy relationships in the bridge table, leaving the customer dimension unaffected



**Figure 5.26. A hierarchy bridge table representing customer ownership**

* (+) is queried with standard SQL syntax using single queries that evaluate the whole hierarchy or designated portions of the hierarchy such as just the leaf nodes
* (+) can be readily generalized to handle complex trees with shared ownership and repeating subassemblies
* (+) allows instant switching between different hierarchies because the hierarchy information is entirely concentrated in the bridge table and the bridge table is chosen at query time
* (+) can be readily generalized to handle time-varying Type 2 ragged hierarchies without affecting the primary customer dimension

but

* (-) requires the generation of a separate record for each parent-child relationship in the tree, including second-level parents, third-level parents, and so on. Although the exact number of records is dependent on the structure of the tree, a rough rule of thumb is three times the number of records as nodes in the tree. Forty-three records are required in the bridge table to support the tree shown in [Figure 5.24](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#a_representative_ragged_organization_hie).
* (-) involves more complex logic than the recursive pointer approach in order to add and move structure within the tree
* (-) requires updating the bridge table when Type 2 changes take place within the customer dimension

Technical Note: POPULATING HIERARCHY BRIDGE TABLES

In February 2001, the following technical note on building bridge tables for ragged hierarchies was published as one of Ralph Kimball's monthly design tips. Because it is so relevant to the ETL processes covered in this book, we reproduce it here, edited slightly, to align the vocabulary precisely with the book.

This month's tip follows on from Ralph's September 1998 article "Help for Hierarchies" (http://www.dbmsmag.com/9809d05.html), which addresses hierarchical structures of variable depth which are traditionally represented in relational databases as recursive relationships. Following is the usual definition of a simple company dimension that contains such a recursive relationship between the foreign key PARENT\_KEY and primary key COMPANY\_KEY.

Create table COMPANY (

COMPANY\_KEY INTEGER NOT NULL,

COMPANY\_NAME VARCHAR2(50),

(plus other descriptive attributes...),

PARENT\_KEY INTEGER);

While this is efficient for storing information on organizational structures, it is not possible to navigate or rollup facts within these hierarchies using the nonprocedural SQL that can be generated by commercial query tools. Ralph's original article describes a bridge table similar to the one that follows that contains one record for each separate path from each company in the organization tree to itself and to every subsidiary below it that solves this problem.

Create table COMPANY\_STRUCTURE (

PARENT\_KEY INTEGER NOT NULL,

SUBSIDIARY\_KEY INTEGER NOT NULL,

SUBSIDIARY\_LEVEL INTEGER NOT NULL,

SEQUENCE\_NUMBER INTEGER NOT NULL,

LOWEST\_FLAG CHAR(1),

HIGHEST\_FLAG CHAR(1),

PARENT\_COMPANY VARCHAR2(50),

SUBSIDIARY\_COMPANY VARCHAR2(50));

The last two columns in this example, which denormalize the company names into this table, are not strictly necessary but have been added to make it easy to see what's going on later.

The following PL/SQL stored procedure demonstrates one possible technique for populating this *hierarchy explosion* bridge table from the COMPANY table on Oracle:

CREATE or Replace procedure COMPANY\_EXPLOSION\_SP as

CURSOR Get\_Roots is

select COMPANY\_KEY ROOT\_KEY,

decode(PARENT\_KEY, NULL,'Y','N') HIGHEST\_FLAG,

COMPANY\_NAME ROOT\_COMPANY

from COMPANY;

BEGIN

For Roots in Get\_Roots

LOOP

insert into COMPANY\_STRUCTURE

(PARENT\_KEY,

SUBSIDIARY\_KEY,

SUBSIDIARY\_LEVEL,

SEQUENCE\_NUMBER,

LOWEST\_FLAG,

HIGHEST\_FLAG,

PARENT\_COMPANY,

SUBSIDIARY\_COMPANY)

select

roots.ROOT\_KEY,

COMPANY\_KEY,

LEVEL - 1,

ROWNUM,

'N',

roots.HIGHEST\_FLAG,

roots.ROOT\_COMPANY,

COMPANY\_NAME

from

COMPANY

Start with COMPANY\_KEY = roots.ROOT\_KEY

connect by prior COMPANY\_KEY = PARENT\_KEY;

END LOOP;

update COMPANY\_STRUCTURE

SET LOWEST\_FLAG = 'Y'

where not exists (select \* from COMPANY

where PARENT\_KEY = COMPANY\_STRUCTURE.SUBSIDIARY\_KEY);

COMMIT;

END; /\* of procedure \*/

This solution takes advantage of Oracle's CONNECT BY SQL extension to walk each tree in the data while building the bridge table. While CONNECT BY is very useful within this procedure, it could not be used by an ad-hoc query tool for general-purpose querying. If the tool can generate this syntax to explore the recursive relationship, it cannot in the same statement join to a fact table. Even if Oracle were to remove this somewhat arbitrary limitation, the performance at query time would probably not be too good.

The following fictional company data will help you understand the COMPANY\_STRUCTURE table and COMPANY\_EXPLOSION\_SP procedure:

/\* column order is Company\_key,Company\_name,Parent\_key \*/

insert into company values (100,'Microsoft',NULL);

insert into company values (101,'Software',100);

insert into company values (102,'Consulting',101);

insert into company values (103,'Products',101);

insert into company values (104,'Office',103);

insert into company values (105,'Visio',104);

insert into company values (106,'Visio Europe',105);

insert into company values (107,'Back Office',103);

insert into company values (108,'SQL Server',107);

insert into company values (109,'OLAP Services',108);

insert into company values (110,'DTS',108);

insert into company values (111,'Repository',108);

insert into company values (112,'Developer Tools',103);

insert into company values (113,'Windows',103);

insert into company values (114,'Entertainment',103);

insert into company values (115,'Games',114);

insert into company values (116,'Multimedia',114);

insert into company values (117,'Education',101);

insert into company values (118,'Online Services',100);

insert into company values (119,'WebTV',118);

insert into company values (120,'MSN',118);

insert into company values (121,'MSN.co.uk',120);

insert into company values (122,'Hotmail.com',120);

insert into company values (123,'MSNBC',120);

insert into company values (124,'MSNBC Online',123);

insert into company values (125,'Expedia',120);

insert into company values (126,'Expedia.co.uk',125);

/\* End example data \*/

The procedure will take the 27 COMPANY records and create 110 COMPANY\_STRUCTURE records make up of one big tree (Microsoft) with 27 nodes and 26 smaller trees. For large datasets, you may find that performance can be enhanced by adding a pair of concatenated indexes on the CONNECT BY columns. In this example, you could build one index on COMPANY\_KEY,PARENT\_KEY and the other on PARENT\_KEY, COMPANY\_KEY.

If you want to visualize the tree structure textually, the following query displays an indented subsidiary list for Microsoft:

select LPAD( ' ', 3\*(SUBSIDIARY\_LEVEL)) || SUBSIDIARY\_COMPANY from

COMPANY\_STRUCTURE order by SEQUENCE\_NUMBER

where PARENT KEY = 100.

The SEQUENCE\_NUMBER has been added since Ralph's original article; it numbers nodes top to bottom, left to right. It allows the correct level-2 nodes to be sorted below their matching level-1 nodes.

For a graphical version of the organization tree, take a look at Visio 2000 Enterprise Edition, which has a database or text-file-driven organization chart wizard. With the help of VBA script, a view on the COMPANY\_STRUCTURE table, and a fact table, it might automate the generation of just the HTML pages you want.

Using Positional Attributes in a Dimension to Represent Text Facts

The SQL interface to relational databases places some severe restrictions on certain kinds of analyses that need to perform complex comparisons across dimension records. Consider the following example of a *text fact*.

Suppose that we measure numeric values for recency, frequency, and intensity (RFI) of all our customers. We call in our data-mining colleagues and ask them to identify the natural clusters of customers in this abstract cube labeled by recency, frequency, and intensity. We really don't want all the numeric results; we want behavioral clusters that are meaningful to our marketing department. After running the cluster identifier datamining step, we find, for example, eight natural clusters of customers. After studying where the centroids of the clusters are located in our RFI cube, we are able to assign behavior descriptions to the eight behavior clusters:

* A: High-volume repeat customer, good credit, few product returns
* B: High-volume repeat customer, good credit, but many product returns
* C: Recent new customer, no established credit pattern
* D: Occasional customer, good credit
* E: Occasional customer, poor credit
* F: Former good customer, not seen recently
* G: Frequent window shopper, mostly unproductive
* H: Other

We can view the tags A through H as *text facts* summarizing a customer's behavior. There aren't a lot of text facts in data warehousing, but these behavior tags seem to be pretty good examples. We can imagine developing a time series of behavior-tag measurements for a customer over time with a data point each month:

John Doe: C C C D D A A A B B

This little time series is pretty revealing. How can we structure our data warehouse to pump out these kinds of reports? And how can we pose interesting constraints on customers to see only those who have gone from cluster A to cluster B in the most recent time period? We require even more complex queries such as finding customers who were an A in the 5th,4th, or3rd previous time period and are a B or a C in either the 2nd or 1st previous period.

We can model this time series of textual behavior tags in several different ways. Each approach has identical information content but differs significantly in ease of use. Let's assume we generate a new behavior tag for each customer each month. Here are three approaches:

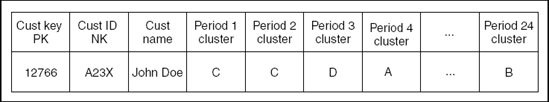
1. Fact table record for each customer for each month, with the behavior tag as a textual fact
2. Slowly changing customer dimension record (Type 2) with the behavior tag as a single attribute (field). A new customer record is created for each customer each month. Same number of new records each month as choice #1.
3. Single customer dimension record with a 24 month time series of behavior tags as 24 attributes, a variant of the Type 3 SCD many alternate realities approach

The whole point of this section is that choices 1 and 2, which create separate records for each behavior tag, leave the data warehouse with a structure that is effectively impossible to query. SQL has no direct approach for posing constraints across records. A sufficiently clever programmer can, of course, do anything, but each complex constraint would need to be programmed by hand. No standard query tool can effectively address design choices 1or2.

Design choice 3, shown in [Figure 5.27](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#using_positional_attributes_to_model_tex), elegantly solves the query problem. Standard query tools can issue extremely complex *straddle constraints* against this design involving as many of the behavior tags as the user needs, because all the targets of the constraints are in the same record. Additionally, the resulting dimension table can be efficiently indexed with bitmap indexes on each of the low-cardinality behavior tags, so that performance can be excellent, even for complex queries.

There are several ways to maintain this positionally dependent set of text facts over time. Depending on how the applications are built, the attributes could be moved backward each sampling period so that a specific physical field is always the most current period. This makes one class of applications simple, since no changes to a query would have to take place to track the current behavior each month. An additional field in the dimension should identify which real month is the most current, so end users will understand when the time series has been updated. Alternatively, the fields in the dimension can have fixed calendar interpretations. Eventually, all the fields originally allotted would be filled, and a decision would be made at that time to add fields.

Using positionally dependent fields in a dimension to represent a time series of text facts has much of the same logic as the *many alternate realities* design approach for the Type 3 SCD.



**Figure 5.27. Using positional attributes to model text facts**

Summary

This chapter has presented the state-of-the-art design techniques for building the dimensions of a data warehouse. Remember that while there are many different data structures in the ETL back room, including flat files, XML data sets, and entity-relation schemas, we transform all these structures into dimensional schemas to prepare for the final data-presentation step in the front room.

Although dimension tables are almost always much smaller than fact tables, dimension tables give the data warehouse its texture and provide the entry points into the universe of fact table measurements.

The dimension-table designs given here are both practical and universal. Every one of the techniques in this chapter can be applied in many different subject areas, and the ETL code and administrative practices can be reused. The three types of slowly changing dimensions (SCDs), in particular, have become basic vocabulary among data warehouse designers. Merely mentioning Type 2 SCD conveys a complete context for handling time variance, assigning keys, building aggregates, and performing queries.

Having exhaustively described the techniques for building dimensions, we now turn our attention to fact tables, the monster tables in our data warehouse containing all of our measurements.

hapter 6. Delivering Fact Tables

Fact tables hold the measurements of an enterprise. The relationship between fact tables and measurements is extremely simple. If a measurement exists, it can be modeled as a fact table row. If a fact table row exists, it is a measurement. What is a measurement? A common definition of a measurement is *an amount determined by observation with an instrument or a scale*.

In dimensional modeling, we deliberately build our databases around the numerical measurements of the enterprise. Fact tables contain measurements, and dimension tables contain the context surrounding measurements. This simple view of the world has proven again and again to be intuitive and understandable to the end users of our data warehouses. This is why we package and deliver data warehouse content through dimensional models.

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release

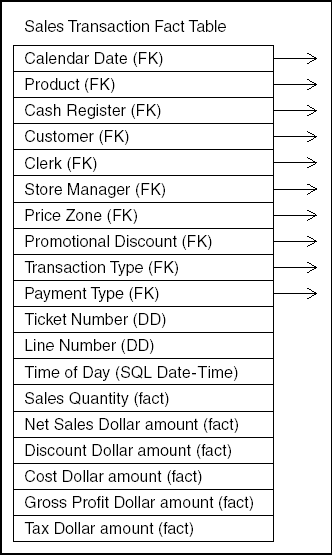
Data Flow: Extract → Clean → Conform → *Deliver*

[Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) describes how to build the dimension tables of the data warehouse. It might seem odd to start with the dimension tables, given that measurements and therefore fact tables are really what end users want to see. But dimension tables are the entry points to fact table data. Facts make no sense unless interpreted by dimensions. Since [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) does a complete job of describing dimensions, we find this chapter to be simpler in some ways.

The Basic Structure of a Fact Table

Every fact table is defined by the *grain* of the table. The grain of the fact table is the definition of the measurement event. The designer must always state the grain of the fact table in terms of how the measurement is taken in the physical world. For example, the grain of the fact table shown in [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo)could be *an individual line item on a specific retail sales ticket*. We will see that this grain can then later be expressed in terms of the dimension foreign keys and possibly other fields in the fact table, but we don't start by defining the grain in terms of these fields. The grain definition must first be stated in physical-measurement terms, and the dimensions and other fields in the fact table will follow.

All fact tables possess a set of foreign keys connected to the dimensions that provide the context of the fact table measurements. Most fact tables also possess one or more numerical measurement fields, which we call *facts*. See [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo). Some fact tables possess one or more special dimension-like fields known as *degenerate dimensions*, which we introduce [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html). Degenerate dimensions exist in the fact table, but they are not foreign keys, and they do not join to a real dimension. We denote degenerate dimensions in [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo) with the notation *DD*.



**Figure 6.1. A sales transaction fact table at the lowest grain**

In practice, fact tables almost always have at least three dimensions, but most fact tables have more. As data warehousing and the surrounding hardware and software technology has matured over the last 20 years, fact tables have grown enormously because they are storing more and more granular data at the lowest levels of measurement. Ironically, the smaller the measurement, the more dimensions apply. In the earliest days of retail-sales data warehouses, data sets were available only at high levels of aggregation. These early retail databases usually had only three or four dimensions (usually product, market, time, and promotion). Today, we collect retail data at the atomic level of the individual sales transaction. An individual sales transaction can easily have the ten dimensions shown in [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo)(calendar date; product; cash register, which rolls up to the store level; customer; clerk; store manager; price zone; promotional discount; transaction type; and payment type). Even with these ten dimensions, we may be tempted to add more dimensions over time including store demographics, marketplace competitive events, and the weather!

Virtually every fact table has a primary key defined by a subset of the fields in the table. In [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo), a plausible primary key for the fact table is the combination of the ticket number and line number degenerate dimensions. These two fields define the unique measurement event of a single item being run up at the cash register. It is also likely that an equivalent primary key could be defined by the combination of cash register and the date/time stamp.

It is possible, if insufficient attention is paid to the design, to violate the assumptions of the primary key on a fact table. Perhaps two identical measurement events have occurred on the same time period, but the data warehouse team did not realize that this could happen. Obviously, every fact table should have a primary key, even if just for administrative purposes. If two or more records in the fact table are allowed to be completely identical because there is no primary key enforced, there is no way to tell the records apart or to be sure that they represent valid discrete measurement events. But as long as the ETL team is sure that separate data loads represent legitimate distinct measurement events, fact table records can be made unique by providing a unique sequence number on the fact record itself at load time. Although the unique sequence number has no business relevance and should not be delivered to end users, it provides an administrative guarantee that a separate and presumably legitimate measurement event occurred. If the ETL team cannot guarantee that separate loads represent legitimate separate measurement events, a primary key on the data must be correctly defined before any data is loaded.

NOTE

The preceding example illustrates the need to make all ETL jobs *reentrant* so that the job can be run a second time, either deliberately or in error, without updating the target database incorrectly. In SQL parlance, UPDATES of constant values are usually safe, but UPDATES that increment values are dangerous. INSERTS are safe if a primary key is defined and enforced because a duplicate INSERT will trigger an error. DELETES are generally safe when the constraints are based on simple field values.

Guaranteeing Referential Integrity

In dimensional modeling, referential integrity means that every fact table is filled with legitimate foreign keys. Or, to put it another way, no fact table record contains corrupt or unknown foreign key references.

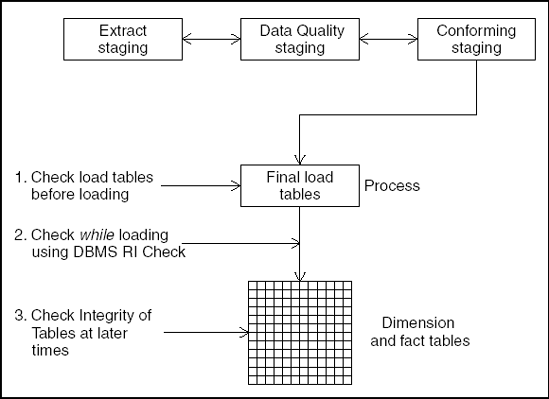
There are only two ways to violate referential integrity in a dimensional schema:

1. Load a fact record with one or more bad foreign keys.
2. Delete a dimension record whose primary key is being used in the fact table.

If you don't pay attention to referential integrity, it is amazingly easy to violate it. The authors have studied fact tables where referential integrity was not explicitly enforced; in every case, serious violations were found. A fact record that violates referential integrity (because it has one or more bad foreign keys) is not just an annoyance; it is dangerous. Presumably, the record has some legitimacy, as it probably represents a true measurement event, but it is stored in the database incorrectly. Any query that references the *bad dimension* in the fact record will fail to include the fact record; by definition, the join between the dimension and this fact record cannot take place. But any query that omits mention of the bad dimension may well include the record in a dynamic aggregation!

In [Figure 6.2](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#choices_for_enforcing_referential_integr), we show the three main places in the ETL pipeline where referential integrity can be enforced. They are:

1. Careful bookkeeping and data preparation just *before loading* the fact table records into the final tables, coupled with careful bookkeeping before deleting any dimension records
2. Enforcement of referential integrity in the database itself *at the moment* of every fact table insertion and every dimension table deletion
3. Discovery and correction of referential integrity violations *after loading* has occurred by regularly scanning the fact table, looking for bad foreign keys



**Figure 6.2. Choices for enforcing referential integrity**

Practically speaking, the first option usually makes the most sense. One of the last steps just before the fact table load is looking up the natural keys in the fact table record and replacing them with the correct contemporary values of the dimension surrogate keys. This process is explained in detail in the next section on the surrogate key pipeline. The heart of this procedure is a special lookup table for each dimension that contains the correct value of the surrogate key to be used for every incoming natural key. If this table is correctly maintained, the fact table records will obey referential integrity. Similarly, when dimension table records are to be deleted, a query must be done attempting to join the dimension record to the fact table. Only if the query returns null should the dimension record be deleted.

The second option of having the database enforce referential integrity continuously is elegant but often too slow for major bulk loads of thousands or millions of records. But this is only a matter of software technology. The Red Brick database system (now sold by IBM) was purpose-built to maintain referential integrity at all times, and it is capable of loading 100 million records an hour into a fact table where it is checking referential integrity on all the dimensions simultaneously!

The third option of checking for referential integrity after database changes have been made is theoretically capable of finding all violations but may be prohibitively slow. The queries checking referential integrity must be of the form:

select f.product\_key

from fact\_table f

where f.product\_key not in (select p.product\_key from

product\_dimension p)

In an environment with a million-row product dimension and a billion-row fact table, this is a ridiculous query. But perhaps the query can be restricted only to the data that has been loaded today. That assumes the time dimension foreign key is correct! But this is a sensible approach that probably should be used as a sanity check even if the first approach is the main processing technique.

Surrogate Key Pipeline

When building a fact table, the final ETL step is converting the natural keys in the new input records into the correct, contemporary surrogate keys. In this section, we assume that all records to be loaded into the fact table are current. In other words, we need to use the most current values of the surrogate keys for each dimension entity (like customer or product). We will deal with late-arriving fact records later in this chapter.

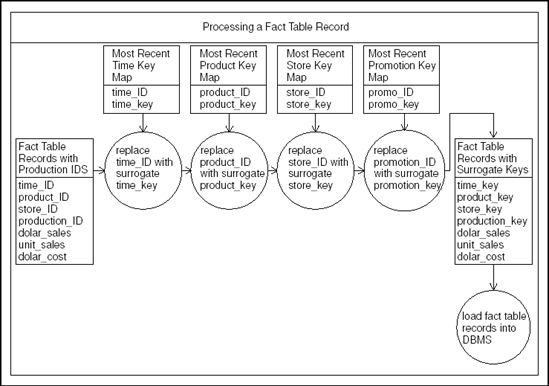
We could theoretically look up the current surrogate key in each dimension table by fetching the most recent record with the desired natural key. This is logically correct but slow. Instead, we maintain a special surrogate key lookup table for each dimension. This table is updated whenever a new dimension entity is created and whenever a Type 2 change occurs on an existing dimension entity. We introduce this table in the [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) when we discuss [Figure 5.16](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#dimension_table_surrogate_key_management).

The dimension tables must all be updated with insertions and Type 2 changes before we even think of dealing with the fact tables. This sequence of updating the dimensions first followed by updating the fact tables is the usual sequence when maintaining referential integrity between the dimension tables and fact tables. The reverse sequence is used when deleting records. First, we remove unwanted fact table records; then we are free to remove dimension records that no longer have any links to the fact table.

NOTE

Don't necessarily delete dimension records just because a fact table has no references to such records. The entities in a dimension may well exist and should be kept in the dimension even if there is no *activity* in the fact table.

When we are done updating our dimension tables, not only are all the dimension records correct; our surrogate key lookup tables that tie the natural keys to the current values of the data warehouse keys have been updated correctly.



**Figure 6.3. The surrogate key pipeline**

Our task for processing the incoming fact table records is simple to understand. See [Figure 6.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#the_surrogate_key_pipeline). We take each natural dimension key in the incoming fact table record and *replace* it with the correct current surrogate key. Notice that we say *replace*. We don't keep the natural key value in the fact record itself. If you care what the natural key value is, you can always find it in the associated dimension record.

If we have between four and 12 natural keys, every incoming fact record requires between four and twelve separate lookups to get the right surrogate keys. First, we set up a multithreaded application that streams all the input records through all the steps shown in [Figure 6.3](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#the_surrogate_key_pipeline). When we say *multithreaded*, we mean that as input record #1 is running the gantlet of successive key lookups and replacements, record #2 is simultaneously right behind record #1, and so on. We do not process all the incoming records in the first lookup step and then pass the whole file to the next step. It is essential for fast performance that the input records are not written to disk until they have passed though all the processing steps. They must literally *fly* through memory without touching ground (the disk) until the end.

If possible, all of the required lookup tables should be pinned in memory so that they can be randomly accessed as each incoming fact record presents its natural keys. This is one of the reasons for making the lookup tables separate from the original data warehouse dimension tables. Suppose we have a million-row lookup table for a dimension. If the natural key is 20 bytes and the surrogate key is 4 bytes, we need roughly 24 MB of RAM to hold the lookup table. In an environment where we can configure the data-staging machine with 4 to 8 GB of RAM, we should easily get all of the lookup tables in memory.

NOTE

The architecture described in the previous paragraph is the highest-performance configuration we know how to design. But let's keep things in perspective. If you are loading a few hundred-thousand records per day, and your load windows are forgiving, you don't need to emphasize performance to this degree. You can define a star join query between your fact tables and your dimension tables, swapping out the natural keys for the surrogate keys, and run the whole process in SQL. In this case, you would also define outer joins on certain dimensions if there were a possibility that any incoming fact records could not be matched to the dimensions (a referential integrity failure).

NOTE

The programming utility *awk* can also be used in this situation because it supports the creation of in-memory arrays for the natural key to surrogate key translation by allowing the natural key itself to serve as the array index. Thus, if you define Translate \_ Dimension \_ A[natural \_ key] = surrogate \_ key, processing each fact record is simply: print Translate \_ Dimension \_ A($1), Translate \_ Dimension \_ B($2), and so on.

In some important large fact tables, we may have a *monster dimension*, like residential customer, that might have a hundred-million members. If we have only one such huge dimension, we can still design a fast pipelined surrogate key system, even though the huge dimension lookup table might have to be read off the disk as it is being used. The secret is to presort both the incoming fact data and the lookup table on the natural key of this dimension. Now the surrogate key replacement is a single pass sort-merge through the two files. This should be pretty fast, although nothing beats in-memory processing. If you have two such monster lookup tables in your pipeline that you cannot pin in memory, you will suffer a processing penalty because of the I/O required for the random accesses on the nonsorted dimension key.

NOTE

It is possible for incoming fact records to make it all the way to the surrogate key pipeline with one or more bad natural keys because we may or may not be checking referential integrity of the underlying source system. If this happens, we recommend creating a new surrogate key and a new dimension record appropriately labeled *Unknown*. Each such incoming bad natural key should get a unique fresh surrogate key if there is any hope of eventually correcting the data. If your business reality is that bad natural keys must remain forever unresolved, a single well-known surrogate key value (and default Unknown record) can be used in all these cases within the affected dimension.

Using the Dimension Instead of a Lookup Table

The lookup table approach described in the previous section works best when the overwhelming fraction of fact records processed each day are *contemporary* (in other words, completely current). But if a significant number of fact records are late arriving, the lookup table cannot be used and the dimension must be the source for the correct surrogate key. This assumes, of course, that the dimension has been designed as described in [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html), with begin and end effective date stamps and the natural key present in every record.

Avoiding the separate lookup table also simplifies the ETL administration before the fact table load because the steps of synchronizing the lookup table with the dimension itself are eliminated.

NOTE

Certain ETL tool suites offer a high-speed, in-memory cache created by looking at the natural keys of the incoming fact records and then linking these natural keys to the right surrogate keys by querying the dimension table in real time. If this works satisfactorily, it has the advantage that the lookup table can be avoided entirely. A possible downside is the startup overhead incurred in dynamically creating this cache while accessing the possibly large dimension. If this dynamic lookup can successfully search for old values of surrogate keys by combining a given natural key with the time stamp on the incoming fact record, this technique could be very effective for handling late-arriving fact table records. You should ask your ETL vendor specific questions about this capability.

Fundamental Grains

Since fact tables are meant to store all the numerical measurements of an enterprise, you might expect that here would be many flavors of fact tables. Surprisingly, in our experience, fact tables can always be reduced to just three fundamental types. We recommend strongly that you adhere to these three simple types in every design situation. When designers begin to mix and combine these types into more complicated structures, an enormous burden is transferred to end user query tools and applications to keep from making serious errors. Another way to say this is that every fact table should have one, *and only one*, grain.

The three kinds of fact tables are: the transaction grain, the periodic snapshot, and the accumulating snapshot. We discuss these three grains in the next three sections.

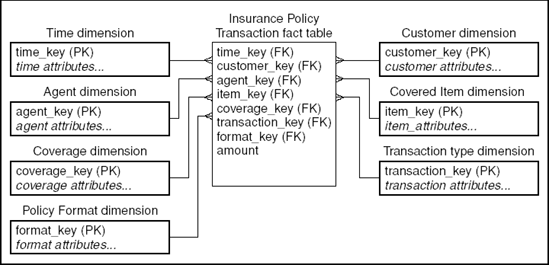
Transaction Grain Fact Tables

The transaction grain represents an instantaneous measurement at a specific point in space and time. The standard example of a transaction grain measurement event is a retail sales transaction. When the product passes the scanner and the scanner beeps (and only *if* the scanner beeps), a record is created. Transaction grain records are created only if the measurement events take place. Thus, a transaction grain fact table can be virtually empty, or it can contain billions of records.

We have remarked that the tiny atomic measurements typical of transaction grain fact tables have a large number of dimensions. You can refer to [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo), which shows the retail scanner event.

In environments like a retail store, there may be only one transaction type (the retail sale) being measured. In other environments, such as insurance claims processing, there may be many transaction types all mixed together in the flow of data. In this case, the numeric measurement field is usually labeled generically as *amount*, and a transaction type dimension is required to interpret the amount. See [Figure 6.4](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#A_standard_transaction_grain_fact_table). In any case, the numeric measures in the transaction grain tables must refer to the instant of the measurement event, not to a span of time or to some other time. In other words, *the facts must be true to the grain*.

Transaction grain fact tables are the largest and most detailed of the three kinds of fact tables. Since individual transactions are often carefully time stamped, transaction grain tables are often used for the most complex and intricate analyses. For instance, in an insurance claims processing environment, a transaction grain fact table is required to describe the most complex sequence of transactions that some claims undergo and to analyze detailed timing measurements among transactions of different types. This level of information simply isn't available in the other two fact-table types. However, it is not always the case that the periodic snapshot and the accumulating snapshot tables can be generated as routine aggregations of the transaction grain tables. In the insurance environment, the operational premium processing system typically generates a measure of *earned premium* for each policy each month. This earned premium measurement must go into the monthly periodic snapshot table, not the transaction grain table. The business rules for calculating earned premium are so complicated that it is effectively impossible for the data warehouse to calculate this monthly measure using only low-level transactions.

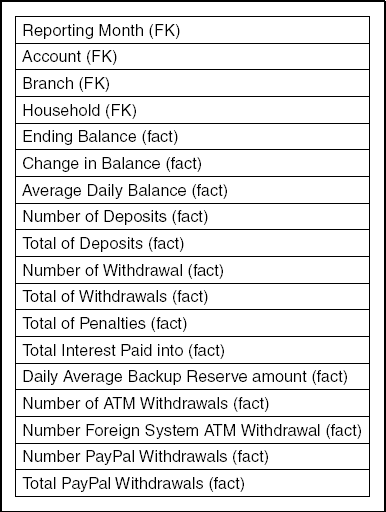


**Figure 6.4. A standard transaction grain fact table drawn from insurance**

Transactions that are time stamped to the nearest minute, second, or microsecond should be modeled by making the calendar day component a conventional dimension with a foreign key to the normal calendar date dimension, and the full date-time expressed as a SQL data type in the fact table, as shown in [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html) in [Figure 5.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html#fact_table_design_for_handling_precise_t).

Since transaction grain tables have unpredictable sparseness, front-end applications cannot assume that any given set of keys will be present in a query. This problem arises when a customer dimension tries to be matched with a demographic behavior dimension. If the constraints are too narrow (say, a specific calendar day), it is possible that no records are returned from the query, and the match of the customer to the demographics is omitted from the results. Database architects aware of this problem may specify a *factless* coverage table that contains every meaningful combination of keys so that an application is guaranteed to match the customer with the demographics. See the discussion of factless fact tables later in this chapter. We will see that the periodic snapshot fact table described in the next section neatly avoids this sparseness problem because periodic snapshots are perfectly dense in their primary key set.

In the ideal case, contemporary transaction level fact records are received in large batches at regular intervals by the data warehouse. The target fact table in most cases should be partitioned by time in a typical DBMS environment. This allows the DBA to drop certain indexes on the most recent time partition, which will speed up a bulk load of new records into this partition. After the load runs to completion, the indexes on the partition are restored. If the partitions can be renamed and swapped, it is possible for the fact table to be offline for only minutes while the updating takes place. This is a complex subject, with many variations in indexing strategies and physical data storage. It is possible that there are indexes on the fact table that do not depend on the partitioning logic and cannot be dropped. Also, some parallel processing database technologies physically distribute data so that the most recent data is not stored in one physical location.



**Figure 6.5. A periodic snapshot for a checking account in a bank**

When the incoming transaction data arrives in a streaming fashion, rather than in discrete file-based loads, we have crossed the boundary into realtime data warehouses, which are discussed in [Chapter 11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch11.html).

Periodic Snapshot Fact Tables

The periodic snapshot represents a span of time, regularly repeated. This style of table is well suited for tracking long-running processes such as bank accounts and other forms of financial reporting. The most common periodic snapshots in the finance world have a monthly grain. All the facts in a periodic snapshot must be true to the grain (that is, they must be measures of activity during the span). In [Figure 6.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_periodic_snapshot_for_a_checking_accou), we show a periodic snapshot for a checking account in a bank, reported every month. An obvious feature in this design is the potentially large number of facts. Any numeric measure of the account that measures activity for the time span is fair game. For this reason, periodic snapshot fact tables are more likely to be *gracefully modified* during their lifetime by adding more facts to the basic grain of the table. See the section on graceful modifications later in this chapter.

The date dimension in the periodic snapshot fact table refers to the period. Thus, the date dimension for a monthly periodic snapshot is a dimension of calendar months. We discuss generating such aggregated date dimensions in [Chapter 5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch05.html).

An interesting question arises about what the exact surrogate keys for all the nontime dimensions should be in the periodic snapshot records. Since the periodic snapshot for the period cannot be generated until the period has passed, the most logical choice for the surrogate keys for the nontime dimensions is their value at the exact end of the period. So, for example, the surrogate keys for the account and branch dimensions in [Figure 6.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_periodic_snapshot_for_a_checking_accou) should be those precise values at the end of the period, notwithstanding the possibility that the account and branch descriptions could have changed in complicated ways in the middle of the period. These intermediate surrogate keys simply do not appear in the monthly periodic snapshot.

Periodic snapshot fact tables have completely predictable sparseness. The account activity fact table in [Figure 6.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_periodic_snapshot_for_a_checking_accou) has one record for each account for each month. As long as an account is active, an application can assume that the various dimensions will all be present in every query.

NOTE

The final tables delivered to end user applications will have completely predictable sparseness, but your original sources may not! You should outer join the primary dimensions of your periodic snapshot fact table to the original data source to make sure that you generate records for every valid combination of keys, even when there is no reported activity for some of them in the current load.

Periodic snapshot fact tables have similar loading characteristics to those of the transaction grain tables. As long as data is promptly delivered to the data warehouse, all records in each periodic load will cluster in the most recent time partition.

However, there are two somewhat different strategies for maintaining periodic snapshot fact tables. The traditional strategy waits until the period has passed and then loads all the records at once. But increasingly, the periodic snapshot maintains a special *current hot rolling period*. The banking fact table of [Figure 6.5](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_periodic_snapshot_for_a_checking_accou) could have 36 fixed time periods, representing the last three years of activity, but also have a special 37th month updated incrementally every night during the current period. This works best if the 37th period is correctly stated when the last day has been loaded in normal fashion. This strategy is less appealing if the final periodic snapshot differs from the last day's load, because of behind-the-scenes ledger adjustments during a month-end-closing process that do not appear in the normal data downloads.

When the hot rolling period is updated continuously throughout the day by streaming the data, rather than through periodic file-based loads, we have crossed the line into real-time data warehouse systems, which we discuss in [Chapter 11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch11.html).

NOTE

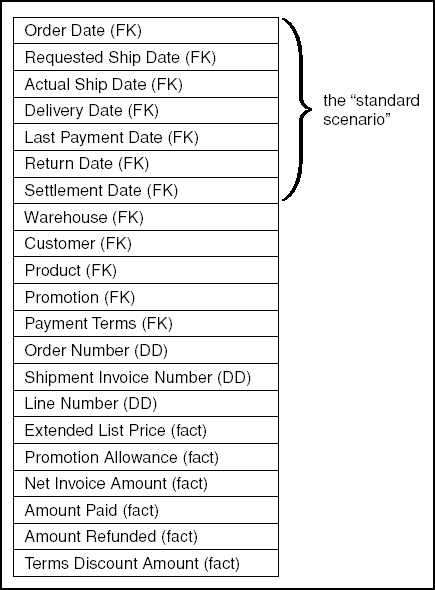
Creating a contunuously updated periodic snapshot can be difficult or even impossible if the business rules for calculating measures at period end are very complex. For example, in insurance companies, the calculation of earned premium at the end of the period is handled by the transaction system, and these measures are available only at the end of the period. The data warehouse cannot easily caluclate earned premium at midpoints of the reporting periods; the business rules are extraordinarily complex and are far beyond the normal ETL transformation logic.

Accumulating Snapshot Fact Tables

The accumulating snapshot fact table is used to describe processes that have a definite beginning and end, such as order fulfillment, claims processing, and most workflows. The accumulating snapshot is not appropriate for long-running continuous processes such as tracking bank accounts or describing continuous manufacturing processes like paper mills.

The grain of an accumulating snapshot fact table is the complete history of an entity from its creation to the present moment. [Figure 6.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#an_accumulating_snapshot_fact_table_wher) shows an accumulating snapshot fact table whose grain is the line item on a shipment invoice.

Accumulating snapshot fact tables have several unusual characteristics. The most obvious difference seen in [Figure 6.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#an_accumulating_snapshot_fact_table_wher) is the large number of calendar date foreign keys. All accumulating snapshot fact tables have a set of dates that implement the *standard scenario* for the table. The standard scenario for the shipment invoice line item in [Figure 6.6](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#an_accumulating_snapshot_fact_table_wher) is order date, requested ship date, actual ship date, delivery date, last payment date, return date, and settlement date. We can assume that an individual record is created when a shipment invoice is created. At that moment, only the order date and the requested ship date are known. The record for a specific line item on the invoice is inserted into the fact table with known dates for these first two foreign keys. The remaining foreign keys are all *not applicable* and their surrogate keys must point to the special record in the calendar date dimension corresponding to Not Applicable. Over time, as events unfold, *the original record is revisited and the foreign keys corresponding to the other dates are overwritten* with values pointing to actual dates. The last payment date may well be overwritten several times as payments are stretched out. The return date and settlement dates may well never be overwritten for normal orders that are not returned or disputed.



**Figure 6.6. An accumulating snapshot fact table where the grain is the shipment invoice line item**

The facts in the accumulating snapshot record are also revisited and overwritten as events unfold. Note that in Oracle, the actual width of an individual record depends on its contents, so accumulating snapshot records in Oracle will always grow. This will affect the residency of disk blocks. In cases where a lot of block splits are generated by these changes, it may be worthwhile to drop and reload the records that have been extensively changed, once the changes settle down, to improve performance. One way to accomplish this is to partition the fact table along two dimensions such as date and current status (Open/Closed). Initially partition along current status, and when the item is closed, move it to the other partition.

An accumulating snapshot fact table is a very efficient and appealing way to represent finite processes with definite beginnings and endings. The more the process fits the standard scenario defined by the set of dates in the fact table, the simpler the end user applications will be. If end users occasionally need to understand extremely complicated and unusual situations, such as a shipment that was damaged or shipped to the wrong customer, the best recourse is a companion transaction grain table that can be fully exploded to see all the events that occurred for the unusual shipment.

Preparing for Loading Fact Tables

In this section, we explain how to build efficient load processes and overcome common obstacles. If done incorrectly, loading data can be the worst experience for an ETL developer. The next three sections outline some of the obstructions you face.

Managing Indexes

Indexes are performance enhancers at query time, but they are performance killers at load time. Tables that are heavily indexed bring your process to a virtual standstill if they are not handled correctly. Before you attempt to load a table, drop all of its indexes in a preload process. You can rebuild the indexes after the load is complete in a post-load process. When your load process includes updates, separate the records required for the updates from those to be inserted and process them separately. In a nutshell, perform the steps that follow to prevent table indexes from causing a bottleneck in your ETL process.

1. Segregate updates from inserts.
2. Drop any indexes not required to support updates.
3. Load updates.
4. Drop all remaining indexes.
5. Load inserts (through bulk loader).
6. Rebuild the indexes.

Managing Partitions

Partitions allow a table (and its indexes) to be physically divided into *mini-tables* for administrative purposes and to improve query performance. The ultimate benefit of partitioning a table is that a query that requires a month of data from a table that has ten years of data can go directly to the partition of the table that contains data for the month without scanning other data. Table partitions can dramatically improve query performance on large fact tables. The partitions of a table are *under the covers*, hidden from the users. Only the DBA and ETL team should be aware of partitions.

The most common partitioning strategy on fact tables is to partition the table by the date key. Because the date dimension is preloaded and static, you know exactly what the surrogate keys are. We've seen designers add a timestamp to fact tables for partitioning purposes, but unless the timestamp is constrained by the user's query, the partitions are not utilized by the optimizer. Since users typically constrain on columns in the date dimension, you need to partition the fact table on the key that joins to the date dimension for the optimizer to recognize the constraint.

Tables that are partitioned by a date interval are usually partitioned by year, quarter, or month. Extremely voluminous facts may be partitioned by week or even day. Usually, the data warehouse designer works with the DBA team to determine the best partitioning strategy on a table-by-table basis. The ETL team must be advised of any table partitions that need to be maintained.

Unless your DBA team takes a proactive role in administering your partitions, the ETL process must manage them. If your load frequency is monthly and your fact table is partitioned by month, partition maintenance is pretty straightforward. When your load frequency differs from the table partitions or your tables are partitioned on an element other than time, the process becomes a bit trickier.

Suppose your fact table is partitioned by year and the first three years are created by the DBA team. When you attempt to load any data after December, 31, 2004, in Oracle you receive the following error:

ORA-14400: inserted partition key is beyond highest legal partition key

At this point, the ETL process has a choice:

* Notify the DBA team, wait for them to manually create the next partition, and resume loading.
* Dynamically add the next partition required to complete loading.

Once the surrogate keys of the incoming data have been resolved, the ETL process can proactively test the incoming data against the defined partitions in the database by comparing the highest date \_ key with the high value defined in the last partition of the table.

select max(date\_key) from 'STAGE\_FACT\_TABLE'

compared with

select high\_value from all\_tab\_partitions

where table\_name = 'FACT\_TABLE'

and partition\_position = (select max(partition\_position)

from all\_tab\_partitions where table\_name = 'FACT\_TABLE')

If the incoming data is in the next year after the defined partition allows, the ETL process can create the next partition with a preprocess script.

ALTER TABLE fact\_table

ADD PARTITION year\_2005 VALUES LESS THAN (1828)

--1828 is the surrogate key for January 1, 2005.

The maintenance steps just discussed can be written in a stored procedure and called by the ETL process before each load. The procedure can produce the required ALTER TABLE statement, inserting the appropriate January 1 surrogate key value as required, depending on the year of the incoming data.

Outwitting the Rollback Log

By design, any relational database management system attempts to support midtransaction failures. The system recovers from uncommitted transaction failures by recording every transaction in a log. Upon failure, the database accesses the log and *undoes* any transactions that have not been committed. To *commit* a transaction means that you or your application explicitly tells the database that the entry of the transaction is completely finished and that the transaction should be permanently written to disk.

The rollback log, also known as the redo log, is invaluable in transaction (OLTP) systems. But in a data warehouse environment where all transactions are managed by the ETL process, the rollback log is a superfluous feature that must be dealt with to achieve optimal load performance. Reasons why the data warehouse does not need rollback logging include:

* All data is entered by a managed process—the ETL system.
* Data is loaded in bulk.
* Data can easily be reloaded if a load process fails.

Each database management system has different logging features and manages its rollback log differently.

Loading the Data

The initial load of a new table has a unique set of challenges. The primary challenge is handling the one-time immense volume of data.

* **Separate inserts from updates**. Many ETL tools (and some databases) offer *update else insert*functionality. This functionality is very convenient and simplifies data-flow logic but is notoriously slow. ETL processes that require updates to existing data should include logic that separates records that already exist in the fact table from new ones. Whenever you are dealing with a substantial amount of data, you want to bulk-load it into the data warehouse. Unfortunately, many bulk-load utilities cannot update existing records. By separating the update records from the inserts, you can first process the updates and then bulk-load the balance of the records for optimal load performance.
* **Utilize a bulk-load utility**. Using a bulk-load utility rather than SQL INSERT statements to load data substantially decreases database overhead and drastically improves load performance.
* **Load in parallel**. When loading volumes of data, physically break up data into logical segments. If you are loading five years of data, perhaps you can make five data files that contain one year each. Some ETL tools allow you to partition data based on ranges of data values dynamically. Once data is divided into equal segments, run the ETL process to load all of the segments in parallel.
* **Minimize physical updates**. Updating records in a table requires massive amounts of overhead in the DBMS, most of which is caused by the database populating the rollback log. To minimize writing to the rollback log, you need to bulk-load data in the database. But what about the updates? In many cases, it is better to delete the records that would be updated and then bulk-load the new versions of those records along with the records entering the data warehouse for the first time. Since the ratio of records being updated versus the number of existing rows plays a crucial factor in selecting the optimal technique, some trial-and-error testing is usually required to see if this approach is the ultimate load strategy for your particular situation.
* **Build aggregates outside of the database**. Sorting, merging, and building aggregates outside of the database may be more efficient than using SQL with COUNT and SUM functions and GROUP BY and ORDER BY keywords in the DBMS. ETL processes that require sorting and/or merging high volumes of data should perform these functions before they enter the relational database staging area. Many ETL tools are adequate at performing these functions, but dedicated tools to perform sort/merges at the operating-system level are worth the investment for processing large datasets.

NOTE

Your ETL process should minimize updates and insert all fact data via the database bulk-load utility. If massive updates are necessary, consider truncating and reloading the entire fact table via the bulk loader to obtain the fastest load strategy. When minimal updates are required, segregate the updates from the inserts and process them separately.

Incremental Loading

The incremental load is the process that occurs periodically to keep the data warehouse synchronized with its respective source systems. Incremental processes can run at any interval or continuously (real-time). At the time of this writing, the customary interval for loading a data warehouse is daily, but no hard-and-fast rule or best practice exists where incremental load intervals are concerned. Users typically like daily updates because they leave data in the warehouse static throughout the day, preventing *twinkling* data, which would make the data ever-changing and cause intraday reporting inconsistencies.

ETL routines that load data incrementally are usually a result of the process that initially loaded the historic data into the data warehouse. It is a preferred practice to keep the two processes one and the same. The ETL team must parameterize the begin \_ date and end \_ date of the extract process so the ETL routine has the flexibility to load small incremental segments or the historic source data in its entirety.

Inserting Facts

When you create new fact records, you need to get data in as quickly as possible. Always utilize your database bulk-load utility. Fact tables are too immense to process via SQL INSERT statements. The database logging caused by SQL INSERT statements is completely superfluous in the data warehouse. The log is created for failure recovery. If your load routine fails, your ETL tool must be able to recover from the failure and pick up where it left off, regardless of database logging.

NOTE

Failure recovery is a feature prevalent in the major ETL tools. Each vendor handles failures, and recovering from them, differently. Make sure your ETL vendors explain exactly how their failure-recovery mechanism works and select the product that requires minimal manual intervention. Be sure to test failure-recovery functionality during your ETL proof-of-concept.

Updating and Correcting Facts

We've participated in many discussions that address the issue of updating data warehouse data—especially fact data. Most agree that dimensions, regardless of the slowly changing dimension strategy, must exactly reflect the data of their source. However, there are several arguments against making changes to fact data once it is in the data warehouse.

Most arguments that support the notion that the data warehouse must reflect all changes made to a transaction system are usually based on theory, not reality. However, the data warehouse is intended to support analysis of the *business*, not the system where the data is derived. For the data warehouse to properly reflect business activity, it must accurately depict its factual events. Regardless of any opposing argument, a data-entry error is not a business event (unless of course, you are building a data mart specifically for analysis of data-entry precision).

Recording unnecessary records that contradict correct ones is counterproductive and can skew analytical results. Consider this example: A company sells 1,000 containers of soda, and the data in the source system records that the package type is 12-ounce cans. After data is published to the data warehouse, a mistake is discovered that the package type should have been 20-ounce bottles. Upon discovery, the source system is immediately updated to reflect the true package type. The business never sold the 12-ounce cans. While performing sales analysis, the business does not need to know a data error occurred. Conversely, preserving the erroneous data might mis-represent the sales figures of 12-ounce cans. You can handle data corrections in the data warehouse in three essential ways.

* Negate the fact.
* Update the fact.
* Delete and reload the fact.

All three strategies result in a reflection of the actual occurrence—the sale of 1,000 20-ounce bottles of soda.

Negating Facts

Negating an error entails creating an exact duplicate of the erroneous record where the measures are a result of the original measures multiplied by −1. The negative measures in the reversing fact table record *cancel out* the original record.

Many reasons exist for negating an error rather than taking other approaches to correcting fact data. The primary reason is for audit purposes. Negating errors in the data warehouse is a good practice if you are specifically looking to capture data-entry errors for analytical purposes. Moreover, if capturing actual erroneous events is significant to the business, the transaction system should have its own data-entry audit capabilities.

Other reasons for negating facts, instead of updating or deleting, involve data volume and ETL performance. In cases where fact table rows are in the hundreds of millions, it could be argued that searching and affecting existing records makes ETL performance deteriorate. However, it is the responsibility of the ETL team to provide required solutions with optimum efficiency. You cannot dictate business policies based on technical criterion. If the business prefers to eliminate errors rather than negate them, it is your responsibility to fulfill that request. This chapter discusses several options to ensure your processes are optimal.

Updating Facts

Updating data in fact tables can be a process-intensive endeavor. In most database management systems, an UPDATE automatically triggers entries in the database log for ROLLBACK protection. Database logging greatly reduces load performance. The best approach to updating fact data is to REFRESH the table via the bulk-load utility. If you must use SQL to UPDATE fact tables, make sure you have the column(s) that uniquely identify the rows in the table indexed and drop all other indexes on the table. Unessential indexes drastically degrade performance of the updates.

Deleting Facts

Most agree that deleting errors is most likely the best solution for correcting data in your fact tables. An arguable drawback is that current versions of previously released reports will not reconcile. But if you accept that you are changing data, any technique used to achieve that goal amends existing reports. Most do not consider changing data a bad thing if the current version represents the truth.

Academically, deleting facts from a fact table is forbidden in data warehousing. However, you'll find that deleting facts is a common practice in most data warehouse environments. If your business requires deletions, two ways to handle them exist:

* **Physical deletes**. In most cases, the business doesn't want to see data that no longer exists in the source transaction systems. When physical deletes are required, you must adhere to the business rules and delete the unwanted records.
* **Logical deletes**. Logically deleting fact records is considered by some to be the *safe* deletion practice. A logical delete entails the utilization of an additional column aptly named *deleted*. It is usually a Bit or Boolean data type and serves as a flag in fact tables to identify deleted records. The caveat to the logical delete approach is that every query that includes the fact table must apply a constraint on the new Boolean field to filter out the logically deleted records.

Physically Deleting Facts

Physically deleting facts means data is permanently removed from the data warehouse. When you have a requirement to physically delete records, make sure the user completely understands that the data will never be able to be retrieved once it is deleted.

Users often carry a misconception that once data enters the data warehouse, it is there forever. So when users say they will *never* have a reason to *see* deleted data, *never* and *see* need to be clarified. Make sure they say exactly what they mean and mean what they say.

* **Never**. It is quite common for users to think in terms of today's requirements because it is based on their current way of thinking about the data they use. Users who've never been exposed to a data warehouse may not be used to having certain types of history available to them. There's an old aphorism: You can't miss what you've never had. In most cases, when a user says *never*, heorshe means *rarely*. Make sure your users are well aware that physical deletion is a permanent removal of the record.
* **See**. When a user says *see*, most likely he or she is referring to the appearance of data in reports. It's quite common that users have no idea what exists in raw data. All data is usually delivered through some sort of delivery mechanism such as business-intelligence tools or reports that may be automatically filtering unwanted data. It's best to check with the team responsible for data presentation to confirm such requirements. If no such team exists, make sure your users are well aware that physical deletion is a permanent removal of the record in the data warehouse.

Once the requirement for permanent physical deletion is confirmed, the next step is to plan a strategy for finding and deleting unwanted facts. The simplest option for resolving deleted records is to truncate and reload the fact tables. Truncating and reloading is usually a viable option only for smaller data warehouses. If you have a large data warehouse consisting of many fact tables, each containing millions or billions of records, it's not recommended to truncate and reload the entire data warehouse with each incremental load.

NOTE

If the source system doesn't contain audit tables to capture deleted data, you must store each data extraction in a staging area to be compared with the next data load—looking for any missing (deleted) records.

If you are lucky, the source system contains audit tables. Audit tables are common in transaction databases where deleted or changed data may have significance or may need to be traced in the future. If audit tables are not available in the source system, another way to detect deleted facts is to compare the source data to a staging table that contains the last data extraction prior to the current data being loaded, which means each day (or whatever your ETL interval is), you must leave a copy of the data extraction in your staging area.

During your ETL process, after both the prior day's extract and the current extract are in the staging area, perform a SQL MINUS on the two tables.

Insert into deleted\_rows nologging

select \* from prior\_extract

MINUS

select \* from current\_extract

The result of the MINUS query reveals rows that have been deleted in the source system but have been loaded into the data warehouse. After the process is complete, you can drop the prior\_extract table, rename the current\_extract table to prior\_extract, and create a new current\_extract table.

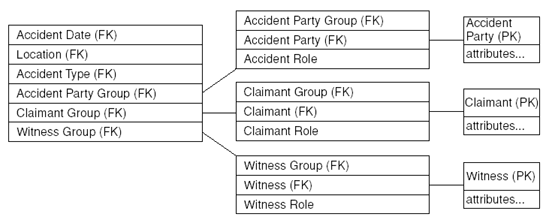
Logically Deleting Facts

When physical deletions are prohibited or need to be analyzed, you can logically delete the record, physically leaving it in place. A logical delete entails the utilization of an additional column named *deleted*. It is usually a Bit or Boolean data type and serves as a flag in fact tables to identify deleted records. The caveat to the logical delete approach is that every query that includes the fact table must apply a constraint on the new Boolean field to filter the logically deleted records.

Factless Fact Tables

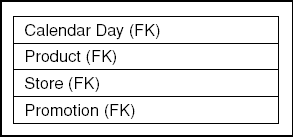
The grain of every fact table is a measurement *event*. In some cases, an event can occur for which there are no measured values! For instance, a fact table can be built representing car-accident events. The existence of each event is indisputable, and the dimensions are compelling and straightforward, as shown in [Figure 6.7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_factless_fact_table_representing_autom). But after the dimensions are assembled, there may well be no measured fact. Event tracking frequently produces factless designs like this example.

Actually, the design in [Figure 6.7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_factless_fact_table_representing_autom) has some other interesting features. Complex accidents have many accident parties, claimants, and witnesses. These are associated with the accident through bridge tables that implement accident party groups, claimant groups, and witness groups. This allows this design to represent accidents ranging from solo fender benders all the way to complex multicar pileups. In this example, it is likely that accident parties, claimants, and witnesses would be added to the groups for a given accident as time goes on. The ETL logic for this application would have to determine whether incoming records represent a new accident or an existing one. A master accident natural key would need to be assigned at the time of first report of the accident. Also, it might be very valuable to deduplicate accident party, claimant, and witness records to investigate fraudulent claims.



**Figure 6.7. A factless fact table representing automobile-accident events**

Another common type of factless fact table represents a *coverage*. The classic example is the table of products on promotion in certain stores on certain days. This table has four foreign keys and no facts, as shown in [Figure 6.8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_factless_coverage_table). This table is used in conjunction with a classic sales table in order to answer the question, *What was on promotion that did not sell?* The formulation of *what did not happen* queries is covered in some detail in *Data Warehouse Toolkit, Second Edition*, pages 251–253. Building an ETL data pipeline for promoted products in each store is easy for those products with a price reduction, because the cash registers in each store know about the special price. But sourcing data for other promotional factors such as special displays or media ads requires parallel separate data feeds probably not coming from the cash register system. Display utilization in stores is a notoriously tricky data-sourcing problem because a common source of this data is the manufacturing representative paid to install the displays. Ultimately, an unbiased third party may need to walk the aisles of each store to generate this data accurately.



**Figure 6.8. A factless coverage table**

Augmenting a Type 1 Fact Table with Type 2 History

Some environments are predominately Type 1, where, for example, the complete history of customer purchases is always accessed through a Type 1 customer dimension reflecting the most current profiles of the customers. In a pure Type 1 environment, historical descriptions of customers are not available. In these situations, the customer dimension is smaller and simpler than in a full-fledged Type 2 design. In a Type 1 dimension, the natural key and the primary key of the dimension have a 1-to-1 relationship.

But in many of these Type 1 environments, there is a desire to access the customer history for specialized analysis. Three approaches can be used in this case:

1. Maintain a full Type 2 dimension off to the side. This has the advantage of keeping the main Type 1 dimension clean and simple. Query the Type 2 dimension to find old customer profiles valid for certain spans of time, and then constrain the fact table using those time spans. This works pretty well for fact tables that represent *immediate actions* like retail sales where the customer is present at the measurement event. But there are some fact tables where the records represent *delayed actions* like a settlement payment made months after a disputed sale. In this case, the span of time defined by a specific customer profile does not overlap the part of the fact table it logically should. The same weird time-synchronization problem can arise when a product with a certain time-delimited profile is sold or returned months later, after the profile has been superceded. This is another example of a delayed-action fact table. If your fact table represents delayed action, you cannot use this option.
2. Build the primary dimension as a full Type 2 dimension. This has the disadvantage that the dimension is bigger and more complicated than a Type 1 dimension. But you can simulate the effect of a Type 1 dimension by formulating all queries with an embedded SELECT statement on the dimension that fetches the natural keys of only the current Customer dimension records; then you can use these natural keys to fetch all the historical dimensional records for the actual join to the fact table.
3. Build the primary dimension as a full Type 2 dimension and simultaneously embed the natural key of the dimension in the fact table alongside the surrogate key. This has the disadvantage that the dimension is bigger and more complicated than a Type 1 dimension. But if the end user application carefully constrains on just the most current customer records, the natural key can be used to join to the fact table, thereby fetching of the entire history. This eliminates the embedded SELECT of approach #2.

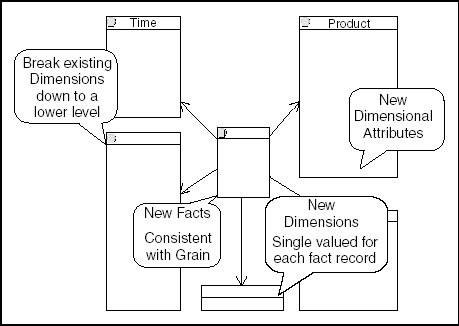
Graceful Modifications

One of the most important advantages of dimensional modeling is that a number of significant changes can be made to the final delivered schemas without affecting end user queries or applications. We call these *graceful modifications*. This is a powerful claim that distinguishes the dimensional modeling world from the normalized modeling world, where these changes are often not graceful and can cause applications to stop working because the physical schema changes.

There are four types of graceful modifications to dimensional schemas:

1. Adding a fact to an existing fact table at the same grain
2. Adding a dimension to an existing fact table at the same grain
3. Adding an attribute to an existing dimension
4. Increasing the granularity of existing fact and dimension tables.

These four types are depicted in [Figure 6.9](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#the_four_types_of_graceful_modification). The first three types require the DBA to perform an ALTER TABLE on the fact table or the dimension table. It is highly desirable that this ALTER TABLE operation be performed on populated tables, rather than requiring that the fact table or dimension table be dropped, redefined, and then reloaded.



**Figure 6.9. The four types of graceful modification**

The first three types raise the issue of how to populate the old history of the tables prior to the addition of the fact, dimension, or attribute. Obviously, it would be nice if old historical values were available. But more often, the fact, dimension, or attribute is added to the schema because it has just become available today. When the change is valid only from today forward, we handle the first three modifications as follows:

1. *Adding a Fact*. Values for the new fact prior to its introduction must be stored as nulls. Null is generally treated well by calculations that span the time during which the fact has been introduced. Counts and averages are correct.
2. *Adding a Dimension*. The foreign key for the new dimension must point to the Not Applicable record in the dimension, for all times in the fact table prior to the introduction of the dimension.
3. *Adding a Dimension Attribute*. In a Type 1 dimension, nothing needs to be done. The new attribute is simply populated in all the dimension records. In a Type 2 dimension, all records referring to time spans preceding the introduction of the attribute need to represent the attribute as null. Time spans that include the introduction of the new attribute are tricky, but probably a reasonable approach is to populate the attribute into these records even though part of their time spans predate the introduction of the attribute.

The fourth type of graceful modification, increasing the granularity of a dimensional schema, is more complicated. Imagine that we are tracking individual retail sales, as depicted in [Figure 6.1](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_sales_transaction_fact_table_at_the_lo). Suppose that we had chosen to represent the location of the sale with a store dimension rather than with the cash register dimension. In both cases, the number of fact table records is exactly the same, since the grain of the fact table is the individual retail sale (line item on a shopper ticket). The only difference between a cashregister view of the retail sales and a store view of the retail sales given the same fundamental grain is the choice of the location dimension. But since cash registers roll up to stores in a perfect many-to-1 relationship, the store attributes are available for both choices of the dimension. If this dimension is called location, with some care, no changes to the SQL of existing applications are needed if the design switches from a store-location perspective to a cash-register-location perspective.

It is even possible to increase the granularity of a fact table without changing existing applications. For example, weekly data could change into daily data. The date dimension would change from week to day, and all applications that constrained or grouped on a particular week would continue to function.

Multiple Units of Measure in a Fact Table

NOTE

PROCESS CHECK Planning & Design:

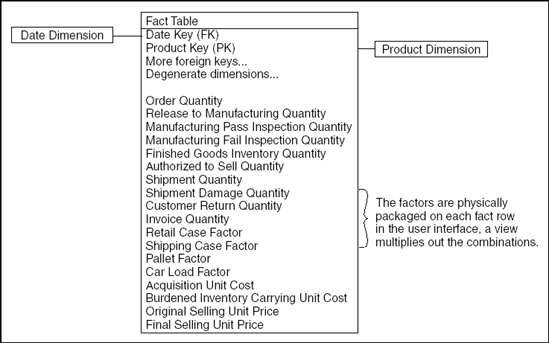
Requirements/Realities → Architecture → *Implementation* → Test/Release

Data Flow:Extract → Clean → Conform → *Deliver*

Sometimes in a value chain involving several business processes monitoring the flow of products through a system, or multiple measures of inventory at different points, a conflict arises in presenting the amounts. Everyone may agree that the numbers are correct but different parties along the chain may wish to see the numbers expressed in different units of measure. For instance, manufacturing managers may wish to see the entire product flow in terms of car loads or pallets. Store managers, on the other hand, may wish to see amounts in shipping cases, retail cases, scan units (sales packs), or consumer units (individual sticks of gum). Similarly, the same quantity of a product may have several possible economic valuations. We may wish to express the valuation in inventory-valuation terms, in list-price terms, in original-selling-price terms, or in final-selling-price terms. Finally, this situation may be exacerbated by having many fundamental quantity facts in each fact record.

Consider a situation where we have ten fundamental quantity facts, five unit-of-measure interpretations, and four valuation schemes. It would be a mistake to present just the 13 quantity facts in the fact table and then leave it up to the user or application developer to seek the correct conversion factors in remote dimension tables, especially if the user queries the product table at a separate time from the fact table without forcing the join to occur. It would be equally bad to try to present all the combinations of facts expressed in the different units of measure in the main fact table. This would require ten times five quantity facts, plus ten times four valuation facts or 90 facts in each fact table record! The correct compromise is to build an underlying physical record with ten quantity facts, four unit-of-measure conversion factors, and four valuation factors. We need only four unit-of-conversion factors rather than five, since the base facts are already expressed in one of the units of measure, preferably either the smallest unit of measure or the largest so that all the calculations to derive the other units of measure are consistently either multiplications or divisions. Our physical design now has ten plus four plus four, or 18 facts, as shown in [Figure 6.10](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_physical_fact_table_design_showing_ten).

The packaging of these factors in the fact table reduces the pressure on the product dimension table to issue new product records to reflect minor changes in these factors, especially the cost and price factors. These items, especially if they routinely evolve, are much more like facts than dimension attributes.

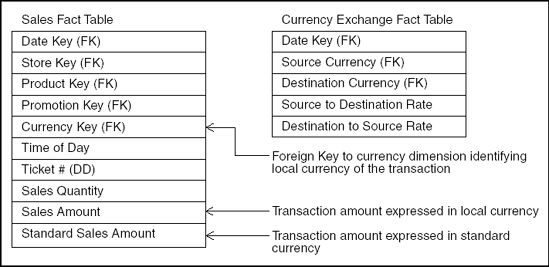


**Figure 6.10. A physical fact table design showing ten facts, five units of measure, and four valuation schemes**

We now actually deliver this fact table to users through one or more views. The most comprehensive view could actually show all 90 combinations of units of measure and valuations, but obviously we could simplify the user interface for any specific user group by only making available the units of measure and valuation factors that the group wanted to see.

Collecting Revenue in Multiple Currencies

Multinational businesses often book transactions, collect revenues, and pay expenses in many different currencies. A good basic design for all of these situations is shown in [Figure 6.11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_schema_design_for_dealing_with_multipl). The primary amount of the sales transaction is represented in the local currency. In some sense, this is always the *correct*value of the transaction. For easy reporting purposes, a second field in the transaction fact record expresses the same amount in a single standard currency, such as the euro. The equivalency between the two amounts is a basic design decision for the fact table and probably is an agreed upon daily spot rate for the conversion of the local currency into the global currency. Now all transactions in a single currency can be added up easily from the fact table by constraining in the currency dimension to a single currency type. Transactions from around the world can easily be added up by summing the global currency field. Note that the fact table contains a currency dimension separate from the geographic dimension representing the store location. Currencies and countries are closely correlated, but they are not the same. Countries may change the identity of their currency during periods of severe inflation. Also, the members of the European Monetary Union must be able to express historical transactions (before Jan 1, 2002) in both their original native currencies and in the euro.



**Figure 6.11. A Schema Design For Dealing With Multiple Currencies**

But what happens if we want to express the value of a set of transactions in a third currency or in the same currency but using the exchange rate at a different time, such as the last day of a reporting period? For this, we need a currency exchange table, also shown in [Figure 6.11](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_schema_design_for_dealing_with_multipl). The currency exchange table typically contains the daily exchange rates both to and from each the local currencies and one or more global currencies. Thus, if there are 100 local currencies and three global currencies, we need 600 exchange-rate records each day. It is probably not practical to build a currency exchange table between each possible pair of currencies, because for 100 currencies, there would be 10,000 daily exchange rates. It is not likely, in our opinion, that a meaningful market for every possible pair of exchange rates actually exists.

Late Arriving Facts

NOTE

PROCESS CHECK Planning & Design:

Requirements/Realities → Architecture → *Implementation* → Test/Release

Data Flow:Extract → Clean → Conform → *Deliver*

Using a customer-purchase scenario, suppose we receive today a purchase record that is several months old. In most operational data warehouses, we are willing to insert this late-arriving record into its correct historical position, even though our sales summary for this prior month will now change. But we must carefully choose the old contemporary dimension records that apply to this purchase. If we have been time-stamping the dimension records in our Type 2 SCDs, our processing involves the following steps:

1. For each dimension, find the corresponding dimension record in effect at the time of the purchase.
2. Using the surrogate keys found in the each of the dimension records from Step 1; replace the natural keys of the late-arriving fact record with the surrogate keys.
3. Insert the late-arriving fact record into the correct physical partition of the database containing the other fact records from the time of the late-arriving purchase.

There are a few subtle points here. We assume that our dimension records contain two time stamps, indicating the beginning and end of the period of validity of the detailed description. This makes the search for the correct dimension records simple.

A second subtle point goes back to our assumption that we have an *operational data warehouse*willing to insert these late-arriving records into old months. If your data warehouse has to *tie to the books*, you can't change an old monthly sales total, even if the old sales total was incorrect. Now you have a tricky situation in which the date dimension on the sales record is for a *booking date*, which may be today, but the other customer, store, and product dimensions should nevertheless refer to the old descriptions in the way we have described. If you are in this situation, you should have a discussion with your finance department to make sure that they understand what you are doing. An interesting compromise we have used in this situation is to carry two sets of date dimensions on purchase records. One refers to the actual purchase date, and the other refers to the booking date. Now you can roll up the sales either operationally or by the books.

The third subtle point is the requirement to insert the late-arriving purchase record into the correct physical partition of the database containing its contemporary *brothers and sisters*. This way, when you move a physical partition from one form of storage to another, or when you perform a backup or restore operation, you will be affecting all the purchase records from a particular span of time. In most cases, this is what you want to do. You can guarantee that all fact records in a time span occupy the same physical partition if you declare the physical partitioning of the fact table to be based on the date dimension. Since you should be using surrogate keys for the date dimension, this is the one case where the surrogate keys of a dimension should be assigned in a particular logical order.

Aggregations

NOTE

PROCESS CHECK Planning & Design:

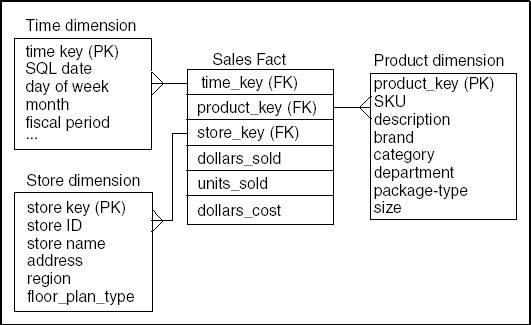
Requirements/Realities → Architecture → *Implementation* → Test/Release Data Flow:Extract → Clean → Conform → *Deliver*

The single most dramatic way to affect performance in a large data warehouse is to provide a proper set of aggregate (summary) records that coexist with the primary base records. Aggregates can have a very significant effect on performance, in some cases speeding queries by a factor of a hundred or even a thousand. No other means exist to harvest such spectacular gains. Certainly, the IT owners of a data warehouse should exhaust the potential for performance gains with aggregates before investing in major new hardware purchases. The benefits of a comprehensive aggregate-building program can be realized with almost every data warehouse hardware and software configuration, including all of the popular relational DBMSs such as Oracle, Red Brick, Informix, Sybase, and DB2, and uniprocessor, SMP and MPP parallel processing architectures. This section describes how to structure a data warehouse to maximize the benefits of aggregates and how to build and use those aggregates without requiring complex accompanying metadata.

Aggregate navigation is a standard data warehouse topic that has been discussed extensively in literature. Let's illustrate this discussion with the simple dimensional schema in [Figure 6.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_simple_dimensional_schema_at_the_grain).

The main points are:

* In a properly designed data warehouse environment, multiple sets of aggregates are built, representing common grouping levels within the key dimensions of the data warehouse. Aggregate navigation has been defined and supported only for dimensional data warehouses. There is no coherent approach for aggregate navigation in a normalized environment.
* An aggregate navigator is a piece of middleware that sits between the requesting client and the DBMS. See [Figure 6.13](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#aggregate_navigator_architecture).
* An aggregate navigator intercepts the client's SQL and, wherever possible, transforms base-level SQL into aggregate aware SQL.

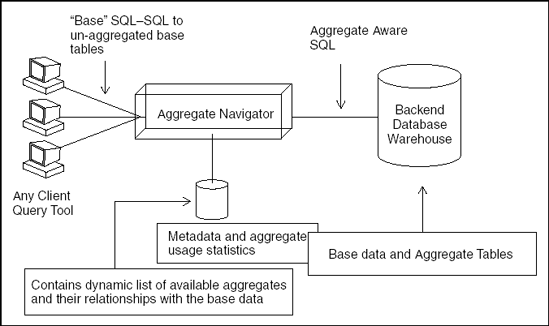


**Figure 6.12. A simple dimensional schema at the grain of day, product, and store**

* The aggregate navigator understands how to transform base-level SQL into aggregate-aware SQL because the navigator uses special metadata that describes the data warehouse aggregate portfolio.

The goals of an aggregate program in a large data warehouse need to be more than just improving performance. A good aggregate program for a large data warehouse should:

1. Provide dramatic performance gains for as many categories of user queries as possible
2. Add only a reasonable amount of extra data storage to the warehouse. *Reasonable* is in the eyes of the DBA, but many data



**Figure 6.13. Aggregate navigator architecture**

warehouse DBAs strive to increase the overall disk storage for the data warehouse by a factor of two or less.

1. Be completely transparent to end users and to application designers, except for the obvious performance benefits. In other words, no end user application SQL should reference the aggregates directly! Aggregates must also benefit all users of the data warehouse, regardless of which query tool they are using.
2. Affect the cost of the data-extract system as little as possible. It is inevitable that a lot of aggregates will have to be built every time data is loaded, but the specification of these aggregates should be as automated as possible.
3. Affect the DBA's administrative responsibilities as little as possible. In particular, the metadata supporting aggregates should be very limited and easy to maintain. Much of the metadata should be automatically created by monitoring user queries and suggesting new aggregates to be created.

A well-designed aggregate environment can achieve all these objectives. A poorly designed aggregate environment can fail all of the objectives! Here is a series of design requirements, which, if adhered to, will achieve our desired objectives.

Design Requirement #1

Aggregates must be stored in their own fact tables, separate from base-level data. Each distinct aggregation level must occupy its own unique fact table.

The separation of aggregates into their own fact tables is very important and has a whole series of beneficial side effects. First, the aggregate navigation scheme described in this section is much simpler when the aggregates occupy their own tables, because the aggregate navigator can learn almost everything it needs from the DBMS's ordinary system catalog, rather than needing additional metadata. Second, an end user is much less likely to accidentally double-count additive fact totals when the aggregates are in separate tables, because every query against a given fact table will by definition go against data of a uniform granularity. Third, the small number of giant numerical entries representing, for instance, national sales totals for the entire year do not have to be shoehorned into the base table. Often, the presence of these few giant numbers forces the database designer to increase the field with of all entries in the database, thereby wasting disk storage. Since the base table is huge and occupies perhaps half of the entire database, it is very helpful to keep its field widths as tight as possible. And fourth, the administration of aggregates is more modular and segmented when the aggregates occupy separate tables. Aggregates can be built at separate times, and with an aggregate navigator, individual aggregates can be taken off-line and placed back on-line throughout the day without affecting other data.

Design Requirement #2

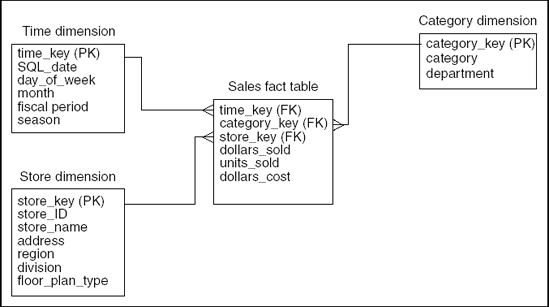
The dimension tables attached to the aggregate fact tables must, wherever possible, be shrunken versions of the dimension tables associated with the base fact table.

NOTE

The MOST shrunken version of a dimension is a dimension removed altogether!

In other words, assuming the base-level fact table as shown in [Figure 6.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_simple_dimensional_schema_at_the_grain), we might wish to build category-level aggregates, representing the product dimension rolled up from the individual product to the category. See [Figure 6.14](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#the_one-way_category_aggregate_schema). We call this the one-way category aggregate schema.

Notice that in this case we have not requested aggregates in either the time dimension or the store dimension. The table in [Figure 6.14](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#the_one-way_category_aggregate_schema) represents how much of a category of a product has sold in each store each day. Our design requirement tells us that the original product table must now be replaced with a shrunken product table, which we might as well call *category*. A simple way to look at this shrunken product table is to think of it as containing only fields that survive the aggregation from individual product up to the category level. Only a few fields will still be uniquely defined.



**Figure 6.14. The one-way category aggregate schema**

For example, both the category description and the department description would be well defined at the category level, and these must have the same field names they have in the base product dimension table. However, the individual UPC number, the package size, and the flavor would not exist at this level and must not appear in the category table.

Shrunken dimension tables are extremely important for aggregate navigation because the scope of any particular aggregation level can be determined by looking in the system catalog description of the shrunken table. In other words, when we look in the category table, all we find is category description and department description. If a query asks for product flavor, we know immediately that this aggregation level cannot satisfy the query, and thus the aggregate navigator must look elsewhere.

Shrunken dimension tables are also attractive because they allow us to avoid filling the original dimension tables with weird null values for all the dimension attributes that are not applicable at higher levels of aggregation. In other words, since we don't have flavor and package size in the category table, we don't have to dream up null values for these fields, and we don't have to encode user applications with tests for these null values.

Although we have focused on shrunken dimension tables, it is possible that the number of measures in the fact table will also shrink as we build ever-higher levels of aggregation. Most of the basic additive facts such as dollar sales, unit sales, and dollar cost will survive at all levels of aggregation, but some dimensions such as promotion and some facts such promotion cost may make sense only at the base level and need to be dropped in the aggregate fact tables.

NOTE

A simplification of requirement #2 builds aggregate fact tables only where specific dimensions are completely eliminated rather than just shrunk. For example, in a retail sales fact table, the location (or store) dimension could be eliminated, effectively creating a national total sales fact table. This approach, when it can be used, has the advantage that the aggregate fact table is now impervious to changes in the dropped dimension. Thus, you could change the definitions of your geographic regions and the table in our example would not change, whereas a partially shrunken location dimension that rolled up to region would need to be recalculated. This approach is not a panacea; in our example, the only queries that could use the proposed aggregate table would be ones that requested the national sales totals.

Design Requirement #3

The base fact table and all its related aggregate fact tables can be associated together as a *family of schemas* so that the aggregate navigator knows which tables are related to each other. Any single schema in the family consists of a fact table and its associated dimension tables. There is always exactly one base schema that is the unaggregated data, and there will be one or more aggregate schemas, representing computed summary data. [Figure 6.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_simple_dimensional_schema_at_the_grain) is a base schema, and [Figure 6.14](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#the_one-way_category_aggregate_schema) is one of perhaps many aggregate schemas in our family.

The registration of this family of fact tables, together with the associated full-size and shrunken dimension tables, is the sole metadata needed in this design.

Design Requirement #4

Force all SQL created by any end user or application to refer exclusively to the base fact table and its associated full-size dimension tables.

This design requirement pervades all user interfaces and all end user applications. When a user examines a graphical depiction of the database, he or she should see only the equivalent of [Figure 6.12](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch06.html#a_simple_dimensional_schema_at_the_grain). The user should not be aware that aggregate tables even exist! Similarly, all hand-coded SQL embedded in report writers or other complex applications should only reference the base fact table and its associated full-size dimension tables.

Administering Aggregations, Including Materialized Views

There are a number of different physical variations of aggregations, depending on the DBMS and the front-end tools. From our design requirements in the previous section, we see that the correct architecture of an aggregate navigator is a middleware module sitting in front of the DBMS intercepting all SQL and examining it for possible redirection. The wrong architecture is an aggregate navigation scheme embedded in a single proprietary front-end tool, where the aggregation benefit is not available to all SQL clients.

There are two fundamental approaches to aggregating navigation at the time of this writing. One is to support a variation of the explicit shrunken fact and dimension tables described in the previous section. The other is to dynamically create these tables by designating certain ephemeral queries to be *materialized* as actual data written to the disk so that subsequent queries can immediately access this data without recomputing the query. Oracle's Materialized Views are an example of the second approach.

Both the explicit-table approach and the materialized-view approach require the DBA to be aware of the effects on the aggregates of updating the underlying base fact tables. Immediately after updating the base tables, the aggregates are invalid and must not be used. In most environments, the base tables must be published to the user community immediately, before all of the aggregates have been recomputed.

Routine daily additions to fact tables may allow the aggregates to be updated by just adding into the various aggregate buckets. But significant changes to the content or logic of a dimension table may require aggregates to be completely dropped and recomputed. For example, Type 1 corrections to historical dimension data will require aggregates to be recomputed if the aggregate is based on the attribute that was changed. But the converse is true! If the changed attribute is not the target of an aggregate, the aggregate can be left alone. For example, one could completely remap the flavor attributes of a big product file and the Category aggregate would not be affected. Visualizing these dependencies is a critical skill in managing the portfolio of aggregates.

Note that Type 2 changes to a dimension generally will not require any aggregates to be rebuilt, as long as the change is administered promptly and does not involve the late-arriving data scenario. Type 2 changes do not affect the existing aggregates; they were correct when they were written.

Generally, a given aggregate fact table should be at least ten-times smaller than the base fact table in order to make the tradeoff between administrative overhead and performance gain worthwhile. Roughly speaking, the performance gain of an aggregate is directly proportional to the storage-shrinkage factor. In other words, an aggregate fact table ten-times smaller than the base table will be about ten-times as fast.

If the overall aggregate table portfolio occupies only 1 percent of the total fact table storage, not enough aggregates have been built. A total aggregate overhead approaching 100 percent (that is, a doubling of the total storage) is reasonable.

Large hardware-based, parallel-processing architectures gain exactly the same performance advantages from aggregates as uniprocessor systems with conventional disk storage, since the gain comes simply from reducing total I/O. However, the salesperson and system engineers of these hardware-intensive systems will deny this because their business is based on selling more hardware, not on improving the performance of the system with clever data structures. Beware!

Delivering Dimensional Data to OLAP Cubes

Server-based OLAP (online analytic processing) products are an increasingly popular component of the data warehouse infrastructure. OLAP servers deliver two primary functions:

* **Query performance**. Using aggregates and specialized indexing and storage structures. The OLAP servers automatically manage aggregates and indexes, a benefit whose value may become clear by reviewing the previous section that discusses how to manage aggregate tables.
* **Analytic richness**. Using languages that, unlike SQL, were designed for complex analytics. OLAP servers also have mechanisms for storing complex calculations and security settings on the server, and some are integrated with data-mining technologies.

In all cases, the best source for an OLAP cube is a dimensional data warehouse stored in an RDBMS. The language of the dimensional data warehouse, dimensions, keys, hierarchies, attributes, and facts, translates exactly to the OLAP world. OLAP engines are developed primarily to support fast and complex querying of dimensional structures. OLAP engines, unlike relational databases and ETL tools, are not designed primarily to cleanse data or ensure referential integrity. Even if your OLAP technology provides features that let you build a cube directly from transactional sources, you should seldom plan to use those features. Instead, you should design a relational data warehouse and populate it using the techniques described in this book. That relational store should feed data to the cube.

Cube Data Sources

The various server-based OLAP products and versions have different features. One area of significant difference is the proximate source of the data that flows into a cube. Some products require that data be sourced from a text file; others require that data be sourced from a single brand of relational database; others permit data to be loaded from practically any source.

If you are sourcing your cube from flat files, one of the last steps in your ETL process is, obviously, to write out the appropriate datasets. Though sourcing from flat files is inelegant and slower than sourcing directly from the relational database, it's not a terrible performance hit: All relational databases and ETL tools can efficiently write the results of a query to files.

Analyze any queries that your cube processing issues against the relational data warehouse. Ensure that such queries are well tuned. If necessary, add indexes or materialized views to improve the performance of these processing queries.

Processing Dimensions

Just as relational dimensions are processed before the fact tables that use them, so must OLAP dimensions be processed before facts. Depending on which OLAP tool you use, you may have the option of processing dimensions and facts in a single transaction, which rolls back if an error is encountered. This is appealing in theory but tends not to scale very well for a large OLAP database. Most large OLAP systems process dimensions one by one, often as the last step in the ETL module that populates the corresponding relational dimension table.

Your ETL system design needs to be aware of a few characteristics of OLAP dimensions. First, recall that OLAP systems are designed for ease of use and good query performance for queries that navigate up and down the strong dimensional hierarchy or hierarchies (such as Product to Brand to Category). With a classic relational data warehouse, you should ensure referential integrity between hierarchy levels (that is, that a product rolls up to one and only one brand, and so on). But with OLAP tools, you absolutely must ensure referential integrity. The OLAP server will insist on referential integrity between the levels of a strong hierarchy. If a violation is found, the dimension processing will either fail, or, at best, the OLAP server will make an assumption about how to proceed. You don't want either of these events to occur: Thoroughly clean your data before OLAP processing is launched.

Changes in Dimension Data

OLAP servers handle different kinds of dimension-data changes completely differently. Most OLAP servers handle new dimension rows, such as adding a new customer, as gracefully as the relational dimensional model does. Updating attribute values that do not participate in strong hierarchies and thus do not have permanent aggregations built on them is usually graceful as well. With changes in dimension attributes where attributes are part of the OLAP dimension hierarchy, the ETL designer needs to be very careful.

Let's get the easy case out of the way first. Type 2 slowly changing dimensions, where a new dimension row with a new surrogate key is added for the changed member, is consumed gracefully by the OLAP servers. From the OLAP server's point of view, this is simply a new customer, indistinguishable from a completely new customer.

A Type 1 slowly changing dimension that updates in place the strong hierarchical attributes is much harder for OLAP servers to manage. The OLAP servers' challenges are exactly analogous to those faced by relational data warehouses with aggregate tables built on the same dimension attributes. When the dimension hierarchy is restructured, the existing aggregations are invalidated. The OLAP servers handle this problem with a wide variety of gracefulness, from not noticing the change, to simply throwing away the aggregations and rebuilding them as a background process, to invalidating the dimension and all cubes that use that dimension, forcing a full OLAP database reprocessing. You should look to your OLAP server vendor for detailed information about how this problem is handled in your technology. Vendors are improving this area with each release, so it's important to use the latest software or validate the conditions for your version.

You should test before OLAP dimension processing to verify that no changes have been made that would put your OLAP database into an invalid or inconsistent state. Depending on the costs and system usage, you could decide to design your system to:

* Let the OLAP and relational data warehouse databases diverge: Defer any further OLAP processing until a convenient time (typically the next weekend).
* Keep the OLAP cubes in synch with the relational data warehouse by halting relational processing as well (typically accumulating changes in the staging area).
* Keep the OLAP cubes in synch with the relational data warehouse by accepting the expensive reprocessing operation during a nightly load. This option would be more palatable if the OLAP cubes were *mirrored* or otherwise remain available to business users during reprocessing.

In any case, the extraordinary event should be logged into the ETL error event table and the operator e-mailed or paged.

Processing Facts

Many people think of cubes as containing only aggregated data. This perception is becoming as old fashioned as the notion that the data warehouse contains only aggregated data. Server-based OLAP products are capable of managing very large volumes of data and are increasingly used to hold data at the same grain as the relational data warehouse. This distinction is important for the design of the cube-processing portion of the ETL system.

Most server-based OLAP products support some form of incremental processing; others support only full processing. Full cube processing is most appropriate for aggregate cubes or small detailed cubes. For good processing performance on a large volume of detailed data, it is vital to use incremental processing for fact data. Two types of cube incremental processing may be available in your OLAP technology: partition-based and incremental facts.

Loading into a partition is an appealing way to load a subset of the cube's data. If your OLAP technology supports partitions, and your cube is partitioned by time, usually weekly or monthly, you can easily load only that time period. If you process daily into weekly partitions, your Monday data will actually be dropped and reloaded seen times during the week until the Sunday load closes down the partition. Certainly, this technique doesn't maximize load efficiency, but it is perfectly acceptable for many applications. It is common to design OLAP cube partitions with the same periodicity as their corresponding relational partitions and to extend the script that manages the relational partitions to also manage the OLAP partitions.

Partitioning the OLAP cube can provide significant benefits for both query and processing performance. Queries against cubes partitioned by one or more dimensions can be executed only against the partitions included in the query rather than the whole cube. The advantages of processing are both to support a simple pseudo-incremental processing as described previously and also to support the processing of multiple partitions in parallel. If your cubes use a complex partition design, your ETL system should be designed to launch multiple partition processing jobs in parallel. If your OLAP server doesn't manage parallel processing on your behalf, you should design this part of your ETL system so you can process a configurable number of partitions in parallel. This is a parameter to optimize during the system-testing phase.

Some OLAP servers also support true incremental fact processing. You supply a way to identify the new data (usually Date Processed), and the OLAP server will add it to the cube or cube partition. If you have late-arriving facts, incremental processing will almost surely be a better approach for you than reprocessing the current partition.

An alternative to full or incremental processing is for the OLAP engine to monitor the source databases for new transactions and to automatically populate the cube with new data. This is a complex process that requires close integration between OLAP engine and relational engine. At the time of this writing, some rudimentary examples of such a feature are available; more functional features are under development.

Common Errors and Problems

One of the most common errors during fact processing is a referential integrity failure: A fact row being processed does not have a corresponding dimension member. If you follow the advice in this and other Toolkit books and use surrogate keys, you should not confront a fact referential integrity failure in the normal course of processing. Nonetheless, you should educate yourself about how your OLAP server will handle this event should an extraordinary event occur.

OLAP servers are not a natural fit with fact data that is updated in place. On the relational side, the preferred design is to use ledgered fact tables that have entries for fact *changes* as positive or negative transaction amounts. The ledger design enables auditing of fact changes and for that reason alone is preferred. An OLAP cube built on a ledgered fact table can accept changes gracefully, though the processing logic may need to be complex enough to support late-arriving fact data if the cube is partitioned by time.

By contrast, an OLAP cube built on a fact table that supports in-place updates is very likely to need full processing each time an update occurs. If the cube is small, this may not be a problem. If the cube is large, you should investigate whether it is acceptable to business users to group updates into weekly or monthly batches or else forgo incorporating the updateable subject area into the OLAP database. Note also that the OLAP processing, which is designed to consume new fact rows, will not identify that fact rows have been updated. The ETL system must trigger the extraordinary processing.

Occasional Full Processing

OLAP technology is not, at the time of this writing, as reliable as relational technology. We believe firmly that the relational dimensional data warehouse should be managed as the data warehouse system-of-record. OLAP cubes should be regarded as ephemeral. Many companies go six months, a year, or more without having to fully reprocess their cubes. However, all installations should develop and test procedures for fully reprocessing the OLAP database. A corollary to this stricture is that you should not design an OLAP cube to contain data that is not in the relational data warehouse or is not easily recoverable into the data warehouse. This includes *writeback* data (most common in budgeting applications), which should be populated directly or indirectly into a relational database.

Until OLAP servers are as reliable and open as their relational brethren, they should be considered secondary systems. OLAP vendors are focusing on reliability and recoverability in versions currently under development, so we hope this second-class status will soon be unnecessary.

Integrating OLAP Processing into the ETL System

If your data warehouse includes OLAP cubes, they should be as professionally managed as any other part of the system. This means that you should have service agreements with business users about data currency and system uptime. Although we generally prefer cubes to be published on the same schedule as relational data, it may be acceptable to refresh cubes on a slower schedule than the relational database. The most important thing is to negotiate an agreement with business users, stick to it, and notify them promptly when the inevitably unexpected occurs.

Although the OLAP server vendors haven't done a great job of providing tools to manage OLAP databases in a professional way, they have all at the very least provided a command-line tool in addition to the more familiar cube management wizard. If you can launch OLAP processing from a command line, you can integrate it, however weakly, into your ETL system. Technologies that include ETL and OLAP offerings from a single vendor provide more elegant integration.

Many systems can fully process the entire OLAP database on a regular schedule, usually weekly or monthly. In this case, the OLAP processing needs merely to verify that week-end or month-end processing of the relational data warehouse has completed successfully. The ETL system can check the system metadata for that successful condition and, if encountered, kick off the cube processing.

For larger systems, a common integration structure includes adding the OLAP dimension processing as the final step in each dimension table processing branch, module, or package in your ETL system. As described previously, you should test for dangerous data operations that might invalidate a cube before launching the OLAP dimension processing script or command. Similarly, the fact table processing module or branch should be extended to include the OLAP processing. The ultimate step of all processing should update metadata of timing and success (or failure) and post this information to the business community.

OLAP Wrap-up

If an OLAP database is part of your data warehouse system, it should be managed rigorously. The ETL team should be expert in the processing features and quirks of the corporate OLAP technology and ideally should have input into the choice of that technology. This is especially true if you are following the current trend of including most or all data warehouse data, including fine-grained data, in the OLAP cubes. To truly reap the benefits of fine-grained cubes, the data warehouse team must own and integrate OLAP processing with the more familiar relational-focused ETL system.

Summary

In this chapter, we defined the fact table as the vessel that holds all numeric measurements of the enterprise. All fact table records have two main parts: the keys that describe the context of the measurements and the measurements themselves which we call facts. We then described the essential role of the fact table provider, who publishes the fact table to the rest of the community.

We saw that referential integrity is hugely important to the proper functioning of a dimensional schema, and we proposed three places where referential integrity can be enforced.

We then showed how to build a surrogate key pipeline for data warehouses that accurately track the historical changes in their dimensional entities.

We described the structure of the three kinds of fact tables: transaction grain, periodic snapshot grain, and accumulating snapshot grain. In our experience, these three grains suffice to model all possible measurement conditions. This simple result is made possible by never mixing the grain of measurements in a single fact table. Adhering to this approach simplifies application development and makes it far less likely that the end user will make mistakes by not understanding the structure of the data.

We then proposed a number of specific techniques for handling graceful modifications to fact and dimension tables, multiple units of measurement, late-arriving fact data, and building aggregations.

We finished the chapter with a specialized section on loading OLAP cubes, which are legitimate first cousins of relational dimensional schemas.

With this chapter, we have described the four main steps of the ETL system in a data warehouse: extraction, quality assurance, conforming, and structuring data as a series of dimensional schemas ready to be consumed by end users. In the next [Chapters 7](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch07.html) and [8](https://www.safaribooksonline.com/library/view/the-data-warehouse/9780764567575/ch08.html), we'll dive into the software tools most commonly used in building ETL systems, and we'll figure out how to schedule the operations of such a complicated system.