20 ETL System Design and Development Process and Tasks

Developing the extract, transformation, and load (ETL) system is the hidden part of the iceberg for most DW/BI projects. So many challenges are buried in the data sources and systems that developing the ETL application invariably takes more time than expected. This chapter is structured as a 10-step plan for creating the data warehouse's ETL system. The concepts and approach described in this chapter, based on content from *The Data Warehouse Lifecycle Toolkit*, *Second Edition* (Wiley, 2008), apply to systems based on an ETL tool, as well as hand-coded systems.

Chapter 20 discusses the following concepts:

- ETL system planning and design consideration
- Recommendations for one-time historic data loads
- Development tasks for incremental load processing
- Real-time data warehousing considerations

ETL Process Overview

This chapter follows the flow of planning and implementing the ETL system. We implicitly discuss the 34 ETL subsystems presented in Chapter 19: ETL Subsystems and Techniques, broadly categorized as extracting data, cleaning and conforming, delivering for presentation, and managing the ETL environment.

Before beginning the ETL system design for a dimensional model, you should have completed the logical design, drafted your high-level architecture plan, and drafted the source-to-target mapping for all data elements.

The ETL system design process is critical. Gather all the relevant information, including the processing burden the extracts will be allowed to place on the operational source systems, and test some key alternatives. Does it make sense to host the transformation process on the source system, target system, or its own platform? What tools are available on each, and how effective are they?

Develop the ETL Plan

ETL development starts out with the high-level plan, which is independent of any specific technology or approach. However, it's a good idea to decide on an ETL tool before doing any detailed planning; this can avoid redesign and rework later in the process.

Step 1: Draw the High-Level Plan

We start the design process with a very simple schematic of the known pieces of the plan: sources and targets, as shown in Figure 20-1. This schematic is for a fictitious utility company's data warehouse, which is primarily sourced from a 30-year-old COBOL system. If most or all the data comes from a modern relational transaction processing system, the boxes often represent a logical grouping of tables in the transaction system model.

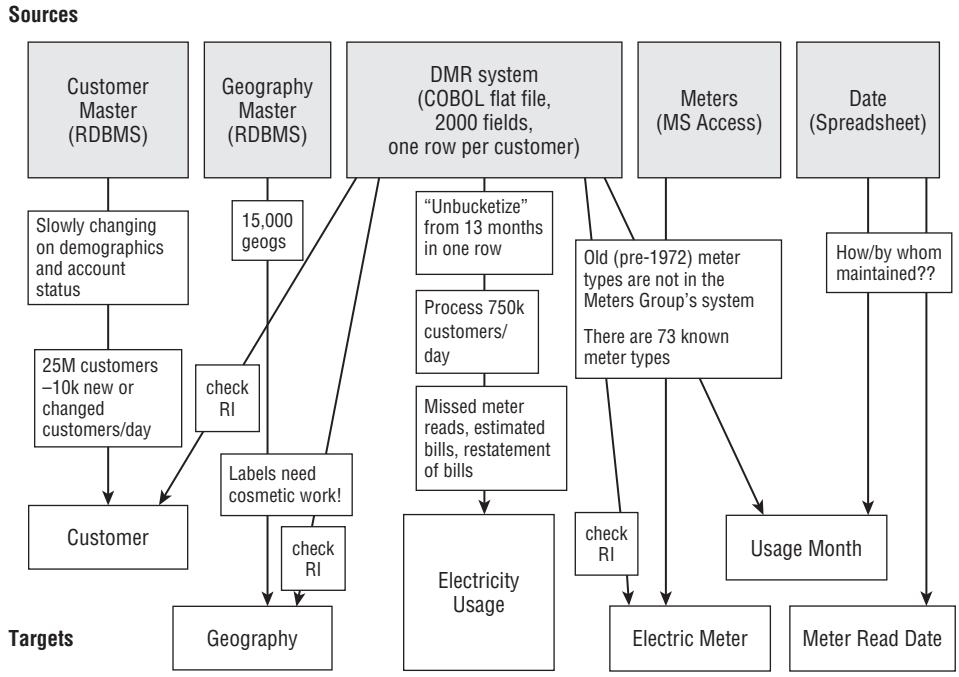

Figure 20-1: Example high-level data staging plan schematic.

As you develop the detailed ETL system specification, the high-level view requires additional details. Figure 20-1 deliberately highlights contemporary questions and unresolved issues; this plan should be frequently updated and released. You might sometimes keep two versions of the diagram: a simple one for communicating

with people outside the team and a detailed version for internal DW/BI team documentation.

Step 2: Choose an ETL Tool

There are a multitude of ETL tools available in the data warehouse marketplace. Most of the major database vendors offer an ETL tool, usually at additional licensing cost. There are also excellent ETL tools available from third-party vendors.

ETL tools read data from a range of sources, including flat files, ODBC, OLE DB, and native database drivers for most relational databases. The tools contain functionality for defining transformations on that data, including lookups and other kinds of joins. They can write data into a variety of target formats. And they all contain some functionality for managing the overall logic flow in the ETL system.

If the source systems are relational, the transformation requirements are straightforward, and good developers are on staff, the value of an ETL tool may not be immediately obvious. However, there are several reasons that using an ETL tool is an industry standard best practice:

- Self-documentation that comes from using a graphical tool. A hand-coded system is usually an impenetrable mess of staging tables, SQL scripts, stored procedures, and operating system scripts.
- Metadata foundation for all steps of the ETL process.
- Version control for multi-developer environments and for backing out and restoring consistent versions.
- Advanced transformation logic, such as fuzzy matching algorithms, integrated access to name and address deduplication routines, and data mining algorithms.
- Improved system performance at a lower level of expertise. Relatively few SQL developers are truly expert on how to use the relational database to manipulate extremely large data volumes with excellent performance.
- Sophisticated processing capabilities, including automatically parallelizing tasks, and automatic fail-over when a processing resource becomes unavailable.
- One-step conversion of virtualized data transformation modules into their physical equivalents.

Don't expect to recoup the investment in an ETL tool on the first phase of the DW/BI project. The learning curve is steep enough that developers sometimes feel the project could have been implemented faster by coding. The big advantages come with future phases, and particularly with future modifications to existing systems.

Step 3: Develop Default Strategies

With an overall idea of what needs to happen and what the ETL tool's infrastructure requires, you should develop a set of default strategies for the common activities in the ETL system. These activities include:

- **Extract from each major source system.** At this point in the design process, you can determine the default method for extracting data from each source system. Will you normally push from the source system to a flat file, extract in a stream, use a tool to read the database logs, or another method? This decision can be modified on a table-by-table basis. If using SQL to access source system data, make sure the native data extractors are used rather than ODBC, if that's an option.
- **Archive extracted and staged data.** Extracted or staged data, before it's been transformed, should be archived for at least a month. Some organizations permanently archive extracted and staged data.
- **Police data quality for dimensions and particularly facts.** Data quality must be monitored during the ETL process rather than waiting for business users to find data problems. Chapter 19 describes a comprehensive architecture for measuring and responding to data quality issues in ETL subsystems 4 through 8.
- **Manage changes to dimension attributes.** In Chapter 19, we described the logic required to manage dimension attribute changes in ETL subsystem 9.
- **Ensure the data warehouse and ETL system meet the system availability requirements.** The first step to meeting availability requirements is to document them. You should document when each data source becomes available and block out high-level job sequencing.
- **Design the data auditing subsystem.** Each row in the data warehouse tables should be tagged with auditing information that describes how the data entered the system.
- **Organize the ETL staging area.** Most ETL systems stage the data at least once or twice during the ETL process. By staging, we mean the data will be written to disk for a later ETL step and for system recovery and archiving.

Step 4: Drill Down by Target Table

After overall strategies for common ETL tasks have been developed, you should start drilling into the detailed transformations needed to populate each target table in the data warehouse. As you're finalizing the source-to-target mappings, you also perform more data profiling to thoroughly understand the necessary data transformations for each table and column.

Ensure Clean Hierarchies

It's particularly important to investigate whether hierarchical relationships in the dimension data are perfectly clean. Consider a product dimension that includes a hierarchical rollup from product stock keeping unit (SKU) to product category.

In our experience, the most reliable hierarchies are well managed in the source system. The best source systems normalize the hierarchical levels into multiple tables, with foreign key constraints between the levels. In this case, you can be confident the hierarchies are clean. If the source system is not normalized—especially if the source for the hierarchies is an Excel spreadsheet on a business user's desktop—then you must either clean it up or acknowledge that it is not a hierarchy.

Develop Detailed Table Schematics

Figure 20-2 illustrates the level of detail that's useful for the table-specific drilldown; it's for one of the tables in the utility company example previously illustrated.

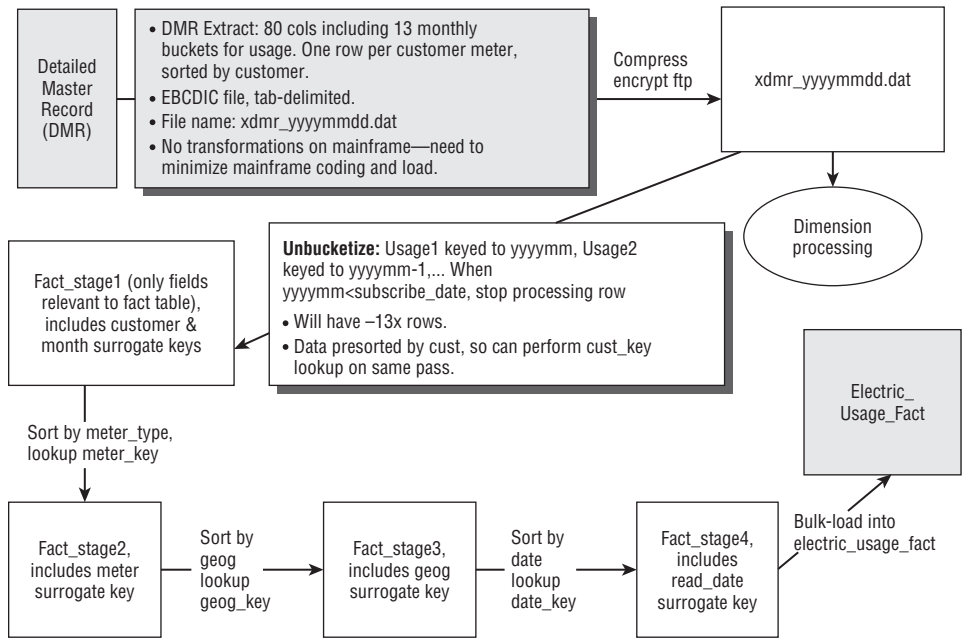

Figure 20-2: Example draft detailed load schematic for the fact table.

All the dimension tables must be processed before the key lookup steps for the fact table. The dimension tables are usually independent from each other, but sometimes they also have processing dependencies. It's important to clarify these dependencies, as they become fixed points around which the job control flows.

Develop the ETL Specification Document

We've walked through some general strategies for high-level planning and the physical design of the ETL system. Now it's time to pull everything together and develop a detailed specification for the entire ETL system.

All the documents developed so far—the source-to-target mappings, data profiling reports, physical design decisions—should be rolled into the first sections of the ETL specification. Then document all the decisions discussed in this chapter, including:

- Default strategy for extracting from each major source system
- Archiving strategy
- Data quality tracking and metadata
- Default strategy for managing changes to dimension attributes
- System availability requirements and strategy
- Design of the data auditing subsystem
- Locations of staging areas

The next section of the ETL specification describes the historic and incremental load strategies for each table. A good specification includes between two and 10 pages of detail for each table, and documents the following information and decisions:

- Table design (column names, data types, keys, and constraints)
- Historic data load parameters (number of months) and volumes (row counts)
- Incremental data volumes, measured as new and updated rows per load cycle
- Handling of late arriving data for facts and dimensions
- Load frequency
- Handling of slowly changing dimension (SCD) changes for each dimension attribute
- Table partitioning, such as monthly
- Overview of data sources, including a discussion of any unusual source characteristics, such as an unusually brief access window
- Detailed source-to-target mapping
- Source data profiling, including at least the minimum and maximum values for each numeric column, count of distinct values in each column, and incidence of NULLs
- Extract strategy for the source data (for example, source system APIs, direct query from database, or dump to flat files)
- Dependencies, including which other tables must be loaded before this table is processed
- Document the transformation logic. It's easiest to write this section as pseudo code or a diagram, rather than trying to craft complete sentences.

- Preconditions to avoid error conditions. For example, the ETL system must check for file or database space before proceeding.
- Cleanup steps, such as deleting working files
- An estimate of whether this portion of the ETL system will be easy, medium, or difficult to implement

NOTE Although most people would agree that all the items described in the ETL system specification document are necessary, it's a lot of work to pull this document together, and even more work to keep it current as changes occur. Realistically, if you pull together the "one-pager" high-level flow diagram, data model and source-to-target maps, and a five-page description of what you plan to do, you'll get a better start than most teams.

Develop a Sandbox Source System

During the ETL development process, the source system data needs to be investigated at great depth. If the source system is heavily loaded, and there isn't some kind of reporting instance for operational queries, the DBAs may be willing to set up a static snapshot of the database for the ETL development team. Early in the development process, it's convenient to poke around sandbox versions of the source systems without worrying about launching a kind of killer query.

It's easy to build a sandbox source system that simply copies the original; build a sandbox with a subset of data only if the data volumes are extremely large. On the plus side, this sandbox could become the basis of training materials and tutorials after the system is deployed into production.

Develop One-Time Historic Load Processing

After the ETL specification has been created, you typically focus on developing the ETL process for the one-time load of historic data. Occasionally, the same ETL code can perform both the initial historic load and ongoing incremental loads, but more often you build separate ETL processes for the historic and ongoing loads. The historic and incremental load processes have a lot in common, and depending on the ETL tool, significant functionality can be reused from one to the other.

Step 5: Populate Dimension Tables with Historic Data

In general, you start building the ETL system with the simplest dimension tables. After these dimension tables have been successfully built, you tackle the historic loads for dimensions with one or more columns managed as SCD type 2.

Populate Type 1 Dimension Tables

The easiest type of table to populate is a dimension table for which all attributes are managed as type 1 overwrites. With a type 1–only dimension, you extract the current value for each dimension attribute from the source system.

Dimension Transformations

Even the simplest dimension table may require substantial data cleanup and will certainly require surrogate key assignment.

Simple Data Transformations

The most common, and easiest, form of data transformation is data type conversion. All ETL tools have rich functions for data type conversion. This task can be tedious, but it is seldom onerous. We strongly recommend replacing NULL values with default values within dimension tables; as we have discussed previously, NULLs can cause problems when they are directly queried.

Combine from Separate Sources

Often dimensions are derived from several sources. Customer information may need to be merged from several lines of business and from outside sources. There is seldom a universal key pre-embedded in the various sources that makes this merge operation easy.

Most consolidation and deduplicating tools and processes work best if names and addresses are first parsed into their component pieces. Then you can use a set of passes with fuzzy logic that account for misspellings, typos, and alternative spellings such as I.B.M., IBM, and International Business Machines. In most organizations, there is a large one-time project to consolidate existing customers. This is a tremendously valuable role for master data management systems.

Decode Production Codes

A common merging task in data preparation is looking up text equivalents for production codes. In some cases, the text equivalents are sourced informally from a nonproduction source such as a spreadsheet. The code lookups are usually stored in a table in the staging database. Make sure the ETL system includes logic for creating a default decoded text equivalent for the case in which the production code is missing from the lookup table.

Validate Many-to-One and One-to-One Relationships

The most important dimensions probably have one or more rollup paths, such as products rolling up to product model, subcategory, and category, as illustrated in Figure 20-3. These hierarchical rollups need to be perfectly clean.

Product Dimension
Product Key (PK)
Product SKU
Product Name
Product Description
Product Model
Product Model Description
Subcategory Description
Category Description
Category Manager

Figure 20-3: Product dimension table with a hierarchical relationship.

Many-to-one relationships between attributes, such as a product to product model, can be verified by sorting on the "many" attribute and verifying that each value has a unique value on the "one" attribute. For example, this query returns the products that have more than one product model:

```
SELECT Product_SKU,
count[*] as Row_Count,
count(distinct Product_Model) as Model_Count
FROM StagingDatabase.Product
GROUP BY Product_SKU
HAVING count(distinct Product_Model) > 1 ;
```

Database administrators sometimes want to validate many-to-one relationships by loading data into a normalized snowflake version of the dimension table in the staging database, as illustrated in Figure 20-4. Note that the normalized version requires individual keys at each of the hierarchy levels. This is not a problem if the source system supplies the keys, but if you normalize the dimension in the ETL environment, you need to create them.

The snowflake structure has some value in the staging area: It prevents you from loading data that violates the many-to-one relationship. However, in general, the relationships should be pre-verified as just described, so that you never attempt to load bad data into the dimension table. After the data is pre-verified, it's not tremendously important whether you make the database engine reconfirm the relationship at the moment you load the table.

If the source system for a dimensional hierarchy is a normalized database, it's usually unnecessary to repeat the normalized structure in the ETL staging area. However, if the hierarchical information comes from an informal source such as a spreadsheet managed by the marketing department, you may benefit from normalizing the hierarchy in the ETL system.

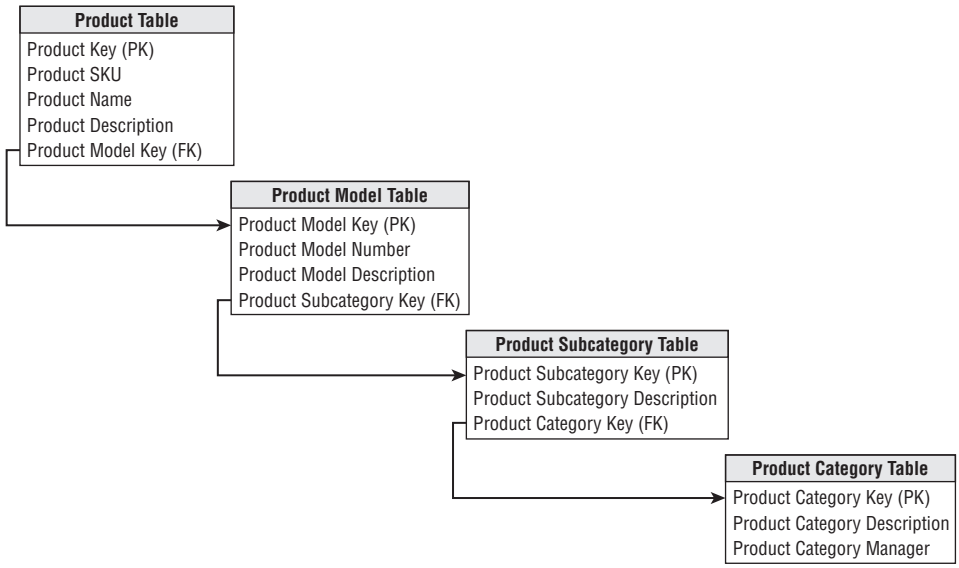

Figure 20-4: Snowflaked hierarchical relationship in the product dimension.

Dimension Surrogate Key Assignment

After you are confident you have dimension tables with one row for each true unique dimension member, the surrogate keys can be assigned. You maintain a table in the ETL staging database that matches production keys to surrogate keys; you can use this key map later during fact table processing.

Surrogate keys are typically assigned as integers, increasing by one for each new key. If the staging area is in an RDBMS, surrogate key assignment is elegantly accomplished by creating a sequence. Although syntax varies among the relational engines, the process is first to create a sequence and then to populate the key map table.

Here's the syntax for the one-time creation of the sequence:

```
create sequence dim1_seq cache=1000; – choose appropriate cache level
```

And then here's the syntax to populate the key map table:

```
insert into dim1_key_map (production_key_id, dim1_key)
select production_key_id, dim1_seq.NEXT
from dim1_extract_table;
```

Dimension Table Loading

After the dimension data is properly prepared, the load process into the target tables is fairly straightforward. Even though the first dimension table is usually small, use the database's bulk or fast-loading utility or interface. You should use fast-loading

techniques for most table inserts. Some databases have extended the SQL syntax to include a `BULK INSERT` statement. Others have published an API to load data into the table from a stream.

The bulk load utilities and APIs come with a range of parameters and transformation capabilities including the following:

- **Turn off logging**. Transaction logging adds significant overhead and is not valuable when loading data warehouse tables. The ETL system should be designed with one or more recoverability points where you can restart processing should something go wrong.
- **Bulk load in fast mode**. However, most of the database engines' bulk load utilities or APIs require several stringent conditions on the target table to bulk load in fast mode. If these conditions are not met, the load should not fail; it simply will not use the "fast" path.
- **Presort the file**. Sorting the file in the order of the primary index significantly speeds up indexing.
- **Transform with caution**. In some cases, the loader supports data conversions, calculations, and string and date/time manipulation. Use these features carefully and test performance. In some cases, these transformations cause the loader to switch out of high-speed mode into a line-by-line evaluation of the load file. We recommend using the ETL tool to perform most transformations.
- **Truncate table before full refresh**. The `TRUNCATE TABLE` statement is the most efficient way to delete all the rows in the table. It's commonly used to clean out a table from the staging database at the beginning of the day's ETL processing.

Load Type 2 Dimension Table History

Recall from Chapter 5: Procurement, that dimension attribute changes are typically managed as type 1 (overwrite) or type 2 (track history by adding new rows to the dimension table). Most dimension tables contain a mixture of type 1 and type 2 attributes. More advanced SCD techniques are described in Chapter 5.

During the historic load, you need to re-create history for dimension attributes that are managed as type 2. If business users have identified an attribute as important for tracking history, they want that history going back in time, not just from the date the data warehouse is implemented. It's usually difficult to re-create dimension attribute history, and sometimes it's completely impossible.

This process is not well suited for standard SQL processing. It's better to use a database cursor construct or, even better, a procedural language such as Visual Basic, C, or Java to perform this work. Most ETL tools enable script processing on the data as it flows through the ETL system.

When you've completely reconstructed history, make a final pass through the data to set the row end date column. It's important to ensure there are no gaps in the series. We prefer to set the row end date for the older version of the dimension member to the day before the row effective date for the new row if these row dates have a granularity of a full day. If the effective and end dates are actually precise date/time stamps accurate to the minute or second, then the end date/time must be set to exactly the begin date/time of the next row so that no gap exists between rows.

Populate Date and Other Static Dimensions

Every data warehouse database should have a date dimension, usually at the granularity of one row for each day. The date dimension should span the history of the data, starting with the oldest fact transaction in the data warehouse. It's easy to set up the date dimension for the historic data because you know the date range of the historic fact data being loaded. Most projects build the date dimension by hand, typically in a spreadsheet.

A handful of other dimensions will be created in a similar way. For example, you may create a budget scenario dimension that holds the values Actual and Budget. Business data governance representatives should sign off on all constructed dimension tables.

Step 6: Perform the Fact Table Historic Load

The one-time historic fact table load differs fairly significantly from the ongoing incremental processing. The biggest worry during the historic load is the sheer volume of data, sometimes thousands of times bigger than the daily incremental load. On the other hand, you have the luxury of loading into a table that's not in production. If it takes several days to load the historic data, that's usually tolerable.

Historic Fact Table Extracts

As you identify records that fall within the basic parameters of the extract, make sure these records are useful for the data warehouse. Many transaction systems keep operational information in the source system that may not be interesting from a business point of view.

It's also a good idea to accumulate audit statistics during this step. As the extract creates the results set, it is often possible to capture various subtotals, totals, and row counts.

Audit Statistics

During the planning phase for the ETL system, you identified various measures of data quality. These are usually calculations, such as counts and sums, that you compare between the data warehouse and source systems to cross-check the integrity

of the data. These numbers should tie backward to operational reports and forward to the results of the load process in the warehouse. The tie back to the operational system is important because it is what establishes the credibility of the warehouse.

NOTE There are scenarios in which it's difficult or impossible for the warehouse to tie back to the source system perfectly. In many cases, the data warehouse extract includes business rules that have not been applied to the source systems. Even more vexing are errors in the source system! Also, differences in timing make it even more difficult to cross-check the data. If it's not possible to tie the data back exactly, you need to explain the differences.

Fact Table Transformations

In most projects, the fact data is relatively clean. The ETL system developer spends a lot of time improving the dimension table content, but the facts usually require a fairly modest transformation. This makes sense because in most cases the facts come from transaction systems used to operate the organization.

The most common transformations to fact data include transformation of null values, pivoting or unpivoting the data, and precomputing derived calculations. All fact rows then enter the surrogate key pipeline to exchange the natural keys for the dimension surrogate keys managed in the ETL system.

Null Fact Values

All major database engines explicitly support a null value. In many source systems, however, the null value is represented by a special value of what should be a legitimate fact. Perhaps the special value of –1 is understood to represent null. For most fact table metrics, the "–1" in this scenario should be replaced with a true NULL. A null value for a numeric measure is reasonable and common in the fact table. Nulls do the "right thing" in calculations of sums and averages across fact table rows. It's only in the dimension tables that you should strive to replace null values with specially crafted default values. Finally, you should not allow any null values in the fact table columns that reference the dimension table keys. These foreign key columns should always be defined as NOT NULL.

Improve Fact Table Content

As we have stressed, all the facts in the final fact table row must be expressed in the same grain. This means there must be no facts representing totals for the year in a daily fact table or totals for some geography larger than the fact table's grain. If the extract includes an interleaving of facts at different grains, the transformation process must eliminate these aggregations, or move them into the appropriate aggregate tables.

The fact row may contain derived facts; although, in many cases it is more efficient to calculate derived facts in a view or an online analytical processing (OLAP) cube rather than in the physical table. For instance, a fact row that contains revenues and costs may want a fact representing net profit. It is very important that the net profit value be correctly calculated every time a user accesses it. If the data warehouse forces all users to access the data through a view, it would be fine to calculate the net profit in that view. If users are allowed to see the physical table, or if they often filter on net profit and thus you'd want to index it, precomputing it and storing it physically is preferable.

Similarly, if some facts need to be simultaneously presented with multiple units of measure, the same logic applies. If business users access the data through a view or OLAP database, then the various versions of the facts can efficiently be calculated at access time.

Pipeline the Dimension Surrogate Key Lookup

It is important that referential integrity (RI) is maintained between the fact table and dimension tables; you must never have a fact row that references a dimension member that doesn't exist. Therefore, you should not have a null value for any foreign key in the fact table nor should any fact row violate referential integrity to any dimension.

The surrogate key pipeline is the final operation before you load data into the target fact table. All other data cleaning, transformation, and processing should be complete. The incoming fact data should look just like the target fact table in the dimensional model, except it still contains the natural keys from the source system rather than the warehouse's surrogate keys. The surrogate key pipeline is the process that exchanges the natural keys for the surrogate keys and handles any referential integrity errors.

Dimension table processing must complete before the fact data enters the surrogate key pipeline. Any new dimension members or type 2 changes to existing dimension members must have already been processed, so their keys are available to the surrogate key pipeline.

First let's discuss the referential integrity problem. It's a simple matter to confirm that each natural key in the historic fact data is represented in the dimension tables. This is a manual step. The historic load is paused at this point, so you can investigate and fix any referential integrity problems before proceeding. The dimension table is either fixed, or the fact table extract is redesigned to filter out spurious rows, as appropriate.

Now that you're confident there will be no referential integrity violations, you can design the historic surrogate key pipeline, as shown in Figure 19-11 in the previous chapter. In this scenario, you need to include BETWEEN logic on any dimension

with type 2 changes to locate the dimension row that was in effect when the historical fact measurement occurred.

There are several approaches for designing the historic load's surrogate key pipeline for best performance; the design depends on the features available in your ETL tool, the data volumes you're processing, and your dimensional design. In theory, you could define a query that joins the fact staging table and each dimension table on the natural keys, returning the facts and surrogate keys from each dimension table. If the historic data volumes are not huge, this can actually work quite well, assuming you staged the fact data in the relational database and indexed the dimension tables to support this big query. This approach has several benefits:

- It leverages the power of the relational database.
- It performs the surrogate key lookups on all dimensions in parallel.
- It simplifies the problem of picking up the correct dimension key for type 2 dimensions. The join to type 2 dimensions must include a clause specifying that the transaction date falls between the row effective date and row end date for that image of the dimension member in the table.

No one would be eager to try this approach if the historic fact data volumes were large in the hundreds of gigabytes to terabyte range. The complex join to the type 2 dimension tables create the greatest demands on the system. Many dimensional designs include a fairly large number of (usually small) dimension tables that are fully type 1, and a smaller number of dimensions containing type 2 attributes. You could use this relational technique to perform the surrogate key lookups for all the type 1 dimensions in one pass and then separately handle the type 2 dimensions. You should ensure the effective date and end date columns are properly indexed.

An alternative to the database join technique described is to use the ETL tool's lookup operator.

When all the fact source keys have been replaced with surrogate keys, the fact row is ready to load. The keys in the fact table row have been chosen to be proper foreign keys to the respective dimension tables, and the fact table is guaranteed to have referential integrity with respect to the dimension tables.

Assign Audit Dimension Key

Fact tables often include an audit key on each fact row. The audit key points to an audit dimension that describes the characteristics of the load, including relatively static environment variables and measures of data quality. The audit dimension can be quite small. An initial design of the audit dimension might have just two environment variables (master ETL version number and profit allocation logic number), and only one quality indicator whose values are Quality Checks Passed and Quality Problems Encountered. Over time, these variables and diagnostic indicators can be

made more detailed and more sophisticated. The audit dimension key is added to the fact table either immediately after or immediately before the surrogate key pipeline.

Fact Table Loading

The main concern when loading the fact table is load performance. Some database technologies support fast loading with a specified batch size. Look at the documentation for the fast-loading technology to see how to set this parameter. You can experiment to find the ideal batch size for the size of the rows and the server's memory configuration. Most people don't bother to get so precise and simply choose a number like 10,000 or 100,000 or 1 million.

Aside from using the bulk loader and a reasonable batch size (if appropriate for the database engine), the best way to improve the performance of the historic load is to load into a partitioned table, ideally loading multiple partitions in parallel. The steps to loading into a partitioned table include:

1. Disable foreign key (referential integrity) constraints between the fact table and each dimension table before loading data.
2. Drop or disable indexes on the fact table.
3. Load the data using fast-loading techniques.
4. Create or enable fact table indexes.
5. If necessary, perform steps to stitch together the table's partitions.
6. Confirm each dimension table has a unique index on the surrogate key column.
7. Enable foreign key constraints between the fact table and dimension tables.

Develop Incremental ETL Processing

One of the biggest challenges with the incremental ETL process is identifying new, changed, and deleted rows. After you have a stream of inserts, modifications, and deletions, the ETL system can apply transformations following virtually identical business rules as for the historic data loads.

The historic load for dimensions and facts consisted largely or entirely of inserts. In incremental processing, you primarily perform inserts, but updates for dimensions and some kinds of fact tables are inevitable. Updates and deletes are expensive operations in the data warehouse environment, so we'll describe techniques to improve the performance of these tasks.

Step 7: Dimension Table Incremental Processing

As you might expect, the incremental ETL system development begins with the dimension tables. Dimension incremental processing is very similar to the historic processing previously described.

Dimension Table Extracts

In many cases, there is a customer master file or product master file that can serve as the single source for a dimension. In other cases, the raw source data is a mixture of dimensional and fact data.

Often it's easiest to pull the current snapshots of the dimension tables in their entirety and let the transformation step determine what has changed and how to handle it. If the dimension tables are large, you may need to use the fact table technique described in the section "Step 8: Fact Table Incremental Processing" for identifying the changed record set. It can take a long time to look up each entry in a large dimension table, even if it hasn't changed from the existing entry.

If possible, construct the extract to pull only rows that have changed. This is particularly easy and valuable if the source system maintains an indicator of the type of change.

Identify New and Changed Dimension Rows

The DW/BI team may not be successful in pushing the responsibility for identifying new, updated, and deleted rows to the source system owners. In this case, the ETL process needs to perform an expensive comparison operation to identify new and changed rows.

When the incoming data is clean, it's easy to find new dimension rows. The raw data has an operational natural key, which must be matched to the same column in the current dimension row. Remember, the natural key in the dimension table is an ordinary dimensional attribute and is not the dimension's surrogate primary key.

You can find new dimension members by performing a lookup from the incoming stream to the master dimension, comparing on the natural key. Any rows that fail the lookup are new dimension members and should be inserted into the dimension table.

If the dimension contains any type 2 attributes, set the row effective date column to the date the dimension member appeared in the system; this is usually yesterday if you are processing nightly. Set the row end date column to the default value for current rows. This should be the largest date, very far in the future, supported by the system. You should avoid using a null value in this second date column because relational databases may generate an error or return the special value Unknown if you attempt to compare a specific value to a NULL.

The next step is to determine if the incoming dimension row has changed. The simplest technique is to compare column by column between the incoming data and the current corresponding member stored in the master dimension table.

If the dimension is large, with more than a million rows, the simple technique of column-wise comparison may be too slow, especially if there are many columns

in the dimension table. A popular alternative method is to use a hash or checksum function to speed the comparison process. You can add two new housekeeping columns to the dimension table: hash type1 and hash type2. You should place a hash of a concatenation of the type 1 attributes in the hash type1 column and similarly for hash type2. Hashing algorithms convert a very long string into a much shorter string that is close to unique. The hashes are computed and stored in the dimension table. Then compute hashes on the incoming rowset in exactly the same way, and compare them to the stored values. The comparison on a single, relatively short string column is far more efficient than the pair-wise comparison on dozens of separate columns. Alternatively, the relational database engine may have syntax such as `EXCEPT` that enables a high-performance query to find the changed rows.

As a general rule, you do not delete dimension rows that have been deleted in the source system because these dimension members probably still have fact table data associated with them in the data warehouse.

Process Changes to Dimension Attributes

The ETL application contains business rules to determine how to handle an attribute value that has changed from the value already stored in the data warehouse. If the revised description is determined to be a legitimate and reliable update to previous information, then the techniques of slowly changing dimensions must be used.

The first step in preparing a dimension row is to decide if you already have that row. If all the incoming dimensional information matches the corresponding row in the dimension table, no further action is required. If the dimensional information has changed, then you can apply changes to the dimension, such as type 1 or type 2.

NOTE You may recall from Chapter 5 that there are three primary methods for tracking changes in attribute values, as well as a set of advanced hybrid techniques. Type 3 requires a change in the structure of the dimension table, creating a new set of columns to hold the "previous" versus "current" versions of the attributes. This type of structural change is seldom automated in the ETL system; it's more likely to be handled as a one-time change in the data model.

The lookup and key assignment logic for handling a changed dimension record during the extract process is shown in Figure 20-5. In this case, the logic flow does not assume the incoming data stream is limited only to new or changed rows.

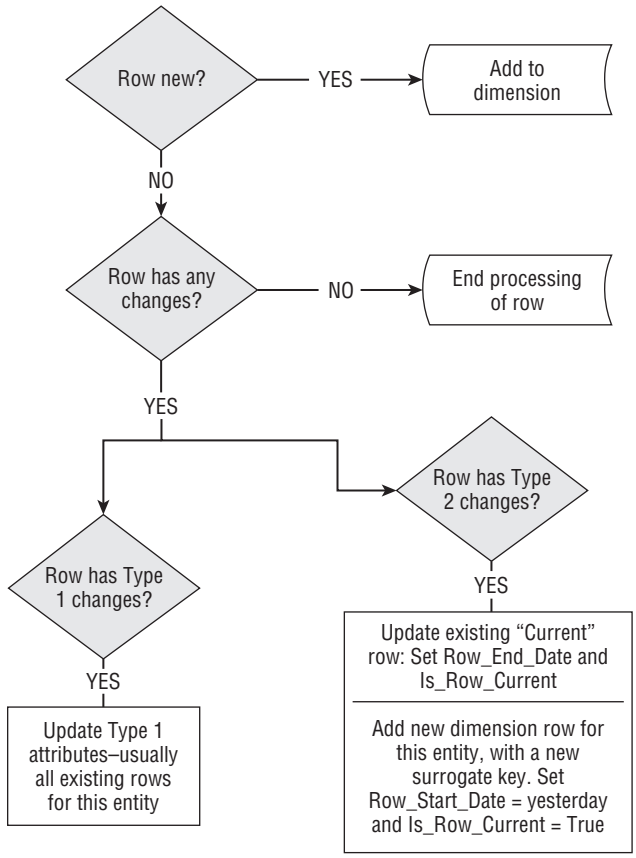

Figure 20-5: Logic flow for handling dimension updates.

Step 8: Fact Table Incremental Processing

Most data warehouse databases are too large to entirely replace the fact tables in a single load window. Instead, new and updated fact rows are incrementally processed.

NOTE It is much more efficient to incrementally load only the records that have been added or updated since the previous load. This is especially true in a journal-style system where history is never changed and only adjustments in the current period are allowed.

The ETL process for fact table incremental processing differs from the historic load. The historic ETL process doesn't need to be fully automated; you can stop the

process to examine the data and prepare for the next step. The incremental processing, by contrast, must be fully automated.

Fact Table Extract and Data Quality Checkpoint

As soon as the new and changed fact rows are extracted from the source system, a copy of the untransformed data should be written to the staging area. At the same time, measures of data quality on the raw extracted data are computed. The staged data serves three purposes:

- Archive for auditability
- Provide a starting point after data quality verification
- Provide a starting point for restarting the process

Fact Table Transformations and Surrogate Key Pipeline

The surrogate key pipeline for the incremental fact data is similar to that for the historic data. The key difference is that the error handling for referential integrity violations must be automated. There are several methods for handling referential integrity violations:

- **Halt the load**. This is seldom a useful solution; although, it's often the default in many ETL tools.
- **Throw away error rows**. There are situations in which a missing dimension value is a signal that the data is irrelevant to the business requirements underlying the data warehouse.
- **Write error rows to a file or table for later analysis**. Design a mechanism for moving corrected rows into a suspense file. This approach is not a good choice for a financial system, where it is vital that all rows be loaded.
- **Fix error rows by creating a dummy dimension row and returning its surrogate key to the pipeline**. The most attractive error handling for referential integrity violations in the incremental surrogate key pipeline is to create a dummy dimension row on-the-fly for the unknown natural key. The natural key is the only piece of information that you may have about the dimension member; all the other attributes must be set to default values. This dummy dimension row will be corrected with type 1 updates when the detailed information about that dimension member becomes available.
- **Fix error rows by mapping to a single unknown member in each dimension.** This approach is not recommended. The problem is that all error rows are mapped to the same dimension member, for any unknown natural key values in the fact table extract.

For most systems, you perform the surrogate key lookups against a query, view, or physical table that subsets the dimension table. The dimension table rows are filtered, so the lookup works against only the current version of each dimension member.

Late Arriving Facts and the Surrogate Key Pipeline

In most data warehouses, the incremental load process begins soon after midnight and processes all the transactions that occurred the previous day. However, there are scenarios in which some facts arrive late. This is most likely to happen when the data sources are distributed across multiple machines or even worldwide, and connectivity or latency problems prevent timely data collection.

If all the dimensions are managed completely as type 1 overwrites, late arriving facts present no special challenges. But most systems have a mixture of type 1 and type 2 attributes. The late arriving facts must be associated with the version of the dimension member that was in effect when the fact occurred. That requires a lookup in the dimension table using the row begin and end effective dates.

Incremental Fact Table Load

In the historic fact load, it's important that data loads use fast-load techniques. In most data warehouses, these fast-load techniques may not be available for the incremental load. The fast-load technologies often require stringent conditions on the target table (for example, empty or unindexed). For the incremental load, it's usually faster to use non-fast-load techniques than to fully populate or index the table. For small to medium systems, insert performance is usually adequate.

If your fact table is very large, you should already have partitioned the fact table for manageability reasons. If incremental data is always loading into an empty partition, you should use fast-load techniques. With daily loads, you would create 365 new fact table partitions each year. This is probably too many partitions for a fact table with long history, so consider implementing a process to consolidate daily partitions into weekly or monthly partitions.

Load Snapshot Fact Tables

The largest fact tables are usually transactional. Transaction fact tables are typically loaded only through inserts. Periodic snapshot fact tables are usually loaded at month end. Data for the current month is sometimes updated each day for current-month-to-date. In this scenario, monthly partitioning of the fact table makes it easy to reload the current month with excellent performance.

Accumulating snapshot fact tables monitor relatively short-lived processes, such as filling an order. The accumulating snapshot fact table is characterized by many

updates for each fact row over the life of the process. This table is expensive to maintain; although accumulating snapshots are almost always much smaller than the other two types of fact tables.

Speed Up the Load Cycle

Processing only data that has been changed is one way to speed up the ETL cycle. This section lists several additional techniques.

More Frequent Loading

Although it is a huge leap to move from a monthly or weekly process to a nightly one, it is an effective way to shorten the load window. Every nightly process involves 1/30 the data volume of a monthly one. Most data warehouses are on a nightly load cycle.

If nightly processing is too expensive, consider performing some preprocessing on the data throughout the day. During the day, data is moved into a staging database or operational data store where data cleansing tasks are performed. After midnight, you can consolidate multiple changes to dimension members, perform final data quality checks, assign surrogate keys, and move the data into the data warehouse.

Parallel Processing

Another way to shorten the load time is to parallelize the ETL process. This can happen in two ways: multiple steps running in parallel and a single step running in parallel.

- **Multiple load steps.** The ETL job stream is divided into several independent jobs submitted together. You need to think carefully about what goes into each job; the primary goal is to create independent jobs.
- **Parallel execution.** The database itself can also identify certain tasks it can execute in parallel. For example, creating an index can typically be parallelized across as many processors as are available on the machine.

NOTE There are good ways and bad ways to break processing into parallel steps. One simple way to parallelize is to extract all source data together, then load and transform the dimensions, and then simultaneously check referential integrity between the fact table and all dimensions. Unfortunately, this approach is likely to be no faster—and possibly much slower—than the even simpler sequential approach because each step launches parallel processes that compete for the same system resources such as network bandwidth, I/O, and memory. To structure parallel jobs well, you need to account not just for logically sequential steps but also for system resources.

Parallel Structures

You can set up a three-way mirror or clustered configuration on two servers to maintain a continuous load data warehouse, with one server managing the loads and the second handling the queries. The maintenance window is reduced to a few minutes daily to swap the disks attached to each server. This is a great way to provide high system availability.

Depending on the requirements and available budget, there are several similar techniques you can implement for tables, partitions, and databases. For example, you can load into an offline partition or table, and swap it into active duty with minimum downtime. Other systems have two versions of the data warehouse database, one for loading and one for querying. These are less effective, but less expensive, versions of the functionality provided by clustered servers.

Step 9: Aggregate Table and OLAP Loads

An aggregate table is logically easy to build. It's simply the results of a really big aggregate query stored as a table. The problem with building aggregate tables from a query on the fact table, of course, occurs when the fact table is just too big to process within the load window.

If the aggregate table includes an aggregation along the date dimension, perhaps to monthly grain, the aggregate maintenance process is more complex. The current month of data must be updated, or dropped and re-created, to incorporate the current day's data.

A similar problem occurs if the aggregate table is defined on a dimension attribute that is overwritten as a type 1. Any type 1 change in a dimension attribute affects all fact table aggregates and OLAP cubes that are defined on that attribute. An ETL process must "back out" the facts from the old aggregate level and move them to the new one.

It is extremely important that the aggregate management system keep aggregations in sync with the underlying fact data. You do not want to create a system that returns a different result set if the query is directed to the underlying detail facts or to a precomputed aggregation.

Step 10: ETL System Operation and Automation

The ideal ETL operation runs the regular load processes in a lights-out manner, without human intervention. Although this is a difficult outcome to attain, it is possible to get close.

Schedule Jobs

Scheduling jobs is usually straightforward. The ETL tool should contain functionality to schedule a job to kick off at a certain time. Most ETL tools also contain functionality to conditionally execute a second task if the first task successfully completed. It's common to set up an ETL job stream to launch at a certain time, and then query a database or filesystem to see if an event has occurred.

You can also write a script to perform this kind of job control. Every ETL tool has a way to invoke a job from the operating system command line. Many organizations are very comfortable using scripting languages, such as Perl, to manage their job schedules.

Automatically Handle Predictable Exceptions and Errors

Although it's easy enough to launch jobs, it's a harder task to make sure they run to completion, gracefully handling data errors and exceptions. Comprehensive error handling is something that needs to be built into the ETL jobs from the outset.

Gracefully Handle Unpredictable Errors

Some errors are predictable, such as receiving an early arriving fact or a NULL value in a column that's supposed to be populated. For these errors, you can generally design your ETL system to fix the data and continue processing. Other errors are completely unforeseen and range from receiving data that's garbled to experiencing a power outage during processing.

We look for ETL tool features and system design practices to help recover from the unexpected. We generally recommend outfitting fact tables with a single column surrogate key that is assigned sequentially to new records that are being loaded. If a large load job unexpectedly halts, the fact table surrogate key allows the load to resume from a reliable point, or back out the load by constraining on a contiguous range of the surrogate keys.

Real-Time Implications

Real-time processing is an increasingly common requirement in data warehousing. There is a strong possibility that your DW/BI system will have a real-time requirement. Some business users expect the data warehouse to be continuously updated throughout the day and grow impatient with stale data. Building a real-time DW/BI system requires gathering a very precise understanding of the true business requirements for real-time data and identifying an appropriate ETL architecture, incorporating a variety of technologies married with a solid platform.

Real-Time Triage

Asking business users if they want "real-time" delivery of data is a frustrating exercise for the DW/BI team. Faced with no constraints, most users will say, "That sounds good; go for it!" This kind of response is almost worthless.

To avoid this situation, we recommend dividing the real-time design challenge into three categories, called instantaneous, intra-day, and daily. We use these terms when we talk to business users about their needs and then design our data delivery pipelines differently for each option. Figure 20-6 summarizes the issues that arise as data is delivered faster.

Daily	Intra-Day	Instantaneous
Batch processing ETL	Micro-batch ETL	Streaming EII/ETL
Wait for file ready	Probe with queries or subscribe to message bus	Drive user presentation from source application
Conventional file table time partition	Daily hot fact table time partition	Separate from fact table
Reconciled	Provisional	Provisional
Complete transaction set	Individual transactions	Transaction fragments
Column screens	Column screens	Column screens
Structure screens	Structure screens	--
Business rule screens	--	--
Final results	Results updated, corrected nightly	Results updated, possibly repudiated nightly

Figure 20-6: Data quality trade-offs with low latency delivery.

Instantaneous means the data visible on the screen represents the true state of the source transaction system at every instant. When the source system status changes, the screen instantly and synchronously responds. An instantaneous real-time system is usually implemented as an enterprise information integration (EII) solution, where the source system itself is responsible for supporting the update of remote users' screens and servicing query requests. Obviously, such a system must limit the complexity of the query requests because all the processing is done on the source system. EII solutions typically involve no caching of data in the ETL pipeline because EII solutions by definition have no delays between the source systems and the users' screens. Some situations are plausible candidates for an instantaneous real-time solution. Inventory status tracking may be a good example, where the decision maker has the right to commit available inventory to a customer in real time.

Intra-day means the data visible on the screen is updated many times per day but is not guaranteed to be the absolute current truth. Most of us are familiar with stock market quote data that is current to within 15 minutes but is not instantaneous.

The technology for delivering frequent real-time data (as well as the slower daily data) is distinctly different from instantaneous real-time delivery. Frequently delivered data is usually processed as micro-batches in a conventional ETL architecture. This means the data undergoes the full gamut of change data capture, extract, staging to file storage in the ETL back room of the data warehouse, cleaning and error checking, conforming to enterprise data standards, assigning of surrogate keys, and possibly a host of other transformations to make the data ready to load into the presentation server. Almost all these steps must be omitted or drastically reduced in an EII solution. The big difference between intra-day and daily delivered data is in the first two steps: change data capture and extract. To capture data many times per day from the source system, the data warehouse usually must tap into a high bandwidth communications channel, such as message queue traffic between legacy applications, an accumulating transaction log file, or low level database triggers coming from the transaction system every time something happens.

Daily means the data visible on the screen is valid as of a batch file download or reconciliation from the source system at the end of the previous working day. There is a lot to recommend daily data. Quite often processes are run on the source system at the end of the working day that correct the raw data. When this reconciliation becomes available, that signals the ETL system to perform a reliable and stable download of the data. If you have this situation, you should explain to the business users what compromises they will experience if they demand instantaneous or intra-day updated data. Daily updated data usually involves reading a batch file prepared by the source system or performing an extract query when a source system readiness flag is set. This, of course, is the simplest extract scenario because you wait for the source system to be ready and available.

Real-Time Architecture Trade-Offs

Responding to real-time requirements means you need to change the DW/BI architecture to get data to the business users' screens faster. The architectural choices involve trade-offs that affect data quality and administration.

You can assume the overall goals for ETL system owners are not changed or compromised by moving to real-time delivery. You can remain just as committed to data quality, integration, security, compliance, backup, recovery, and archiving as you were before starting to design a real-time system. If you agree with this statement, then read the following very carefully! The following sections discuss the typical trade-offs that occur as you implement a more real-time architecture:

Replace Batch Files

Consider replacing a batch file extract with reading from a message queue or transaction log file. A batch file delivered from the source system may represent a clean

and consistent view of the source data. The batch file may contain only those records resulting from completed transactions. Foreign keys in the batch files are probably resolved, such as when the file contains an order from a new customer whose complete identity may be delivered with the batch file. Message queue and log file data, on the other hand, is raw instantaneous data that may not be subject to any corrective process or business rule enforcement in the source system. In the worst case, this raw data may 1) be incorrect or incomplete because additional transactions may arrive later; 2) contain unresolved foreign keys that the DW/BI system has not yet processed; and 3) require a parallel batch-oriented ETL data flow to correct or even replace the hot real-time data each 24 hours. And if the source system subsequently applies complex business rules to the input transactions first seen in the message queues or the log files, then you really don't want to recapitulate these business rules in the ETL system!

Limit Data Quality Screens

Consider restricting data quality screening only to column screens and simple decode lookups. As the time to process data moving through the ETL pipeline is reduced, it may be necessary to eliminate more costly data quality screening, especially structure screens and business rule screens. Remember that column screens involve single field tests and/or simple lookups to replace or expand known values. Even in the most aggressive real-time applications, most column screens should survive. But structure screens and business rule screens by definition require multiple fields, multiple records, and possibly multiple tables. You may not have time to pass an address block of fields to an address analyzer. You may not check referential integrity between tables. You may not be able to perform a remote credit check through a web service. All this may require informing the users of the provisional and potentially unreliable state of the raw real-time data and may require that you implement a parallel, batch-oriented ETL pipeline that overwrites the real-time data periodically with properly checked data.

Post Facts with Dimensions

You should allow early arriving facts to be posted with old copies of dimensions. In the real-time world, it is common to receive transaction events before the context (such as the identity of the customer) of those transactions is updated. In other words, the facts arrive before the dimensions. If the real-time system cannot wait for the dimensions to be resolved, then old copies of the dimensions must be used if they are available, or generic empty versions of the dimensions must be used otherwise. If and when revised versions of the dimensions are received, the data warehouse may decide to post those into the hot partition or delay updating the dimension until a batch process takes over, possibly at the end of the day. In any case, the users need

to understand there may be an ephemeral window of time where the dimensions don't exactly describe the facts.

Eliminate Data Staging

Some real-time architectures, especially EII systems, stream data directly from the production source system to the users' screens without writing the data to permanent storage in the ETL pipeline. If this kind of system is part of the DW/BI team's responsibility, then the team should have a serious talk with senior management about whether backup, recovery, archiving, and compliance responsibilities can be met, or whether those responsibilities are now the sole concern of the production source system.

Real-Time Partitions in the Presentation Server

To support real-time requirements, the data warehouse must seamlessly extend its existing historical time series right up to the current instant. If the customer has placed an order in the last hour, you need to see this order in the context of the entire customer relationship. Furthermore, you need to track the hourly status of this most current order as it changes during the day. Even though the gap between the production transaction processing systems and the DW/BI system has shrunk in most cases to 24 hours, the insatiable needs of your business users require the data warehouse to fill this gap with real-time data.

One design solution for responding to this crunch is building a real-time partition as an extension of the conventional, static data warehouse. To achieve real-time reporting, a special partition is built that is physically and administratively separated from the conventional data warehouse tables. Ideally, the real-time partition is a true database partition where the fact table in question is partitioned by activity date.

In either case, the real-time partition ideally should meet the following tough set of requirements:

- Contain all the activity that has occurred since the last update of the static data warehouse.
- Link as seamlessly as possible to the grain and content of the static data warehouse fact tables, ideally as a true physical partition of the fact table.
- Be indexed so lightly that incoming data can continuously be "dribbled in." Ideally, the real-time partition is completely unindexed; however, this may not be possible in certain RDBMSs where indexes have been built that are not logically aligned with the partitioning scheme.
- Support highly responsive queries even in the absence of indexes by pinning the real-time partition in memory.

The real-time partition can be used effectively with both transaction and periodic snapshot fact tables. We have not found this approach needed with accumulating snapshot fact tables.

Transaction Real-Time Partition

If the static data warehouse fact table has a transaction grain, it contains exactly one row for each individual transaction in the source system from the beginning of "recorded history." The real-time partition has exactly the same dimensional structure as its underlying static fact table. It contains only the transactions that have occurred since midnight when you last loaded the regular fact tables. The real-time partition may be completely unindexed, both because you need to maintain a continuously open window for loading and because there is no time series because only today's data is kept in this table.

In a relatively large retail environment experiencing 10 million transactions per day, the static fact table would be pretty big. Assuming each transaction grain row is 40 bytes wide (seven dimensions plus three facts, all packed into 4-byte columns), you accumulate 400 MB of data each day. Over a year, this would amount to approximately 150 GB of raw data. Such a fact table would be heavily indexed and supported by aggregates. But the daily real-time slice of 400 MB should be pinned in memory. The real-time partition can remain biased toward very fast-loading performance but at the same time provide speedy query performance.

Periodic Snapshot Real-Time Partition

If the static data warehouse fact table has a periodic grain (say, monthly), then the real-time partition can be viewed as the current hot rolling month. Suppose you are a big retail bank with 15 million accounts. The static fact table has the grain of account by month. A 36-month time series would result in 540 million fact table rows. Again, this table would be extensively indexed and supported by aggregates to provide query good performance. The real-time partition, on the other hand, is just an image of the current developing month, updated continuously as the month progresses. Semi-additive balances and fully additive facts are adjusted as frequently as they are reported. In a retail bank, the supertype fact table spanning all account types is likely to be quite narrow, with perhaps four dimensions and four facts, resulting in a real-time partition of 480 MB. The real-time partition again can be pinned in memory.

On the last day of the month, the periodic real-time partition can, with luck, just be merged onto the less volatile fact table as the most current month, and the process can start again with an empty real-time partition.

Summary

The previous chapter introduced 34 subsystems that are possible within a comprehensive ETL implementation. In this chapter, we provided detailed practical advice for actually building and deploying the ETL system. Perhaps the most interesting perspective is to separate the initial historical loads from the ongoing incremental loads. These processes are quite different.

In general we recommend using a commercial ETL tool as opposed to maintaining a library of scripts, even though the ETL tools can be expensive and have a significant learning curve. ETL systems, more than any other part of the DW/BI edifice, are legacy systems that need to be maintainable and scalable over long periods of time and over changes of personnel.

We concluded this chapter with some design perspectives for real-time (low latency) delivery of data. Not only are the real-time architectures different from conventional batch processing, but data quality is compromised as the latency is progressively lowered. Business users need to be thoughtful participants in this design trade-off.