EPAM Systems, RD Dep., RD Dep.

INTRODUCTION TO DWH AND ETL

Dimension & Fact Table Techniques

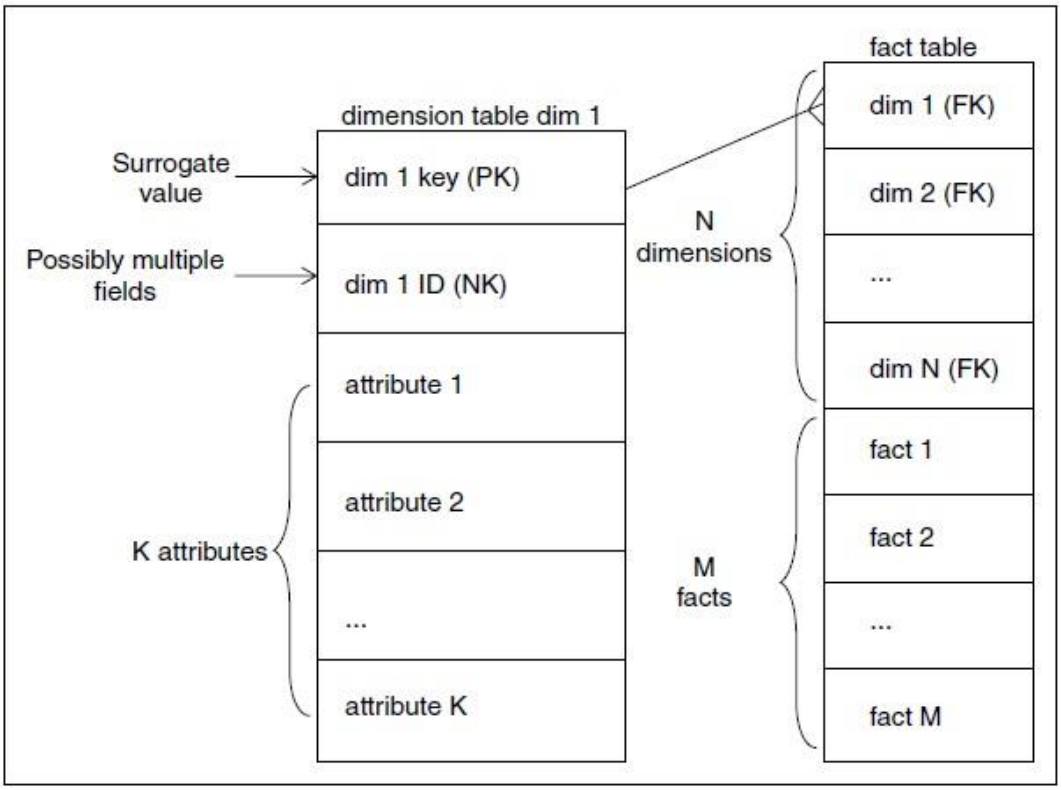
CONTENTS

1. 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.

1.1 THE BASIC STRUCTURE OF A DIMENSION

All dimensions should be physically built to have the minimal set of components shown in Figure below.



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. 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. 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.

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.

1.2 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.

1.3 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. 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.

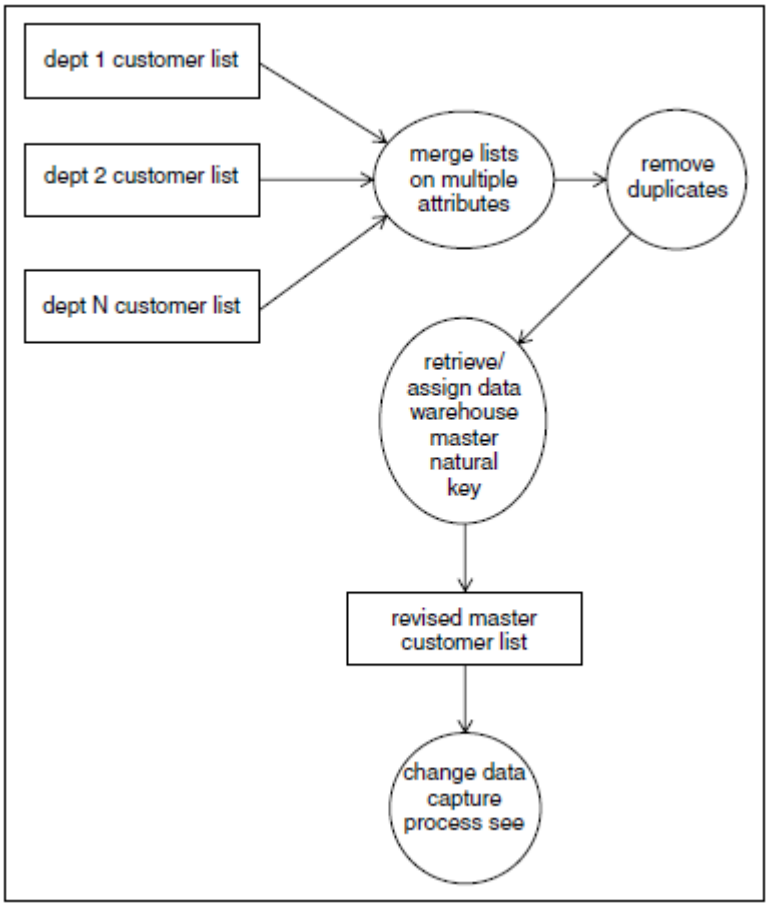
2. DIMENSION ENTITIES CLASSIFICATION

2.1 BIG DIMENSIONS

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.

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.



Merging and Deduplicating Different 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.

2.2 SMALL DIMENSIONS

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 transaction grained 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 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.

2.3 CALENDAR DATE DIMENSION

Calendar date dimensions are attached to virtually every fact table to allow navigation of the fact table through familiar dates, months, fiscal periods, and special days on the calendar. You would never want to compute Easter in SQL, but rather want to look it up in the calendar date dimension. The calendar date dimension typically has many attributes describing characteristics such as week number, month name, fiscal period, and national holiday indicator. To facilitate partitioning, the primary key of a date dimension can be more meaningful, such as an integer representing YYYYMMDD, instead of a sequentially-assigned surrogate key. However, the date dimension table needs a special row to represent unknown or to-be-determined dates. If a smart date key is used, filtering and grouping should be based on the dimension table's attributes, not the smart key.

When further precision is needed, a separate date/time stamp can be added to the fact table. The date/time stamp is not a foreign key to a dimension table, but rather is a standalone column. If business users constrain or group on time-of-day attributes, such as day part grouping or shift number, then you would add a separate time-of-day dimension foreign key to the fact table.

Here are some examples of columns that might be in a date dimension:

COLUMN	DATA TYPE	DESCRIPTION
DATE_KEY	INTEGER / DATE	primary key; date in a format YYYYMMDD. If you are using PostgreSQL there is no need to implement such a key and use DATE type instead, because of good internal representation of DATE in PostgreSQL.
DAY_OF_WEEK_NUMBER	INTEGER	number of a day in a week (from 1 to 7)
DAY_OF_WEEK_DESC	VARCHAR(25)	name of a day (Sunday, Monday, ... Saturday)
WEEKEND_FLAG	INTEGER	flag whether a day is a weekend day (1) or not (0)
ISO_WEEK_NUMBER	INTEGER	number of an ISO week in a year (from 1 to 52)
DAY_OF_MONTH_NUMBER	INTEGER	number of a day in a month (from 1 to 31)
MONTH_VALUE	VARCHAR(2)	2-digits string in a format MM, number of a month in a year (from 01 to 12)
MONTH_DESC	VARCHAR(25)	full name of a month (January, February, ... December)
QUARTER_VALUE	VARCHAR(1)	1-digit string in a format Q, number of a quarter in a year (from 1 to 4)
QUARTER_DESC	VARCHAR(2)	name of a quarter (Q1, Q2, Q3, Q4)
YEAR_VALUE	VARCHAR(4)	4-digits string in a format YYYY (2011, 2012, ...)

2.4 JUNK DIMENSION

Junk dimensions are particularly useful when dealing with low-cardinality attributes, which means attributes with a small number of distinct values.

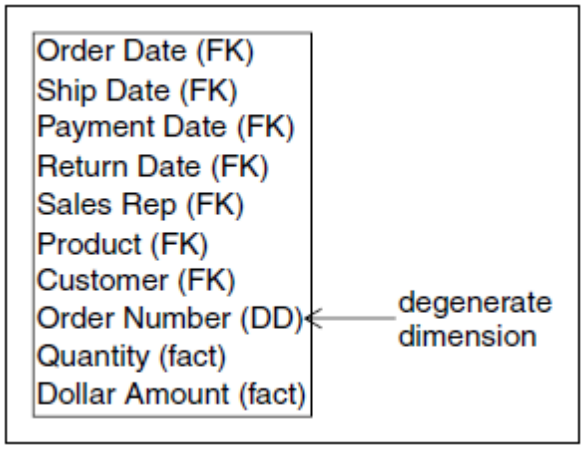
Examples include flags like 'Yes' or 'No', 'Male' or 'Female', 'High', 'Medium', 'Low', etc.

Instead of creating separate dimensions for each of these low-cardinality flags, a junk dimension consolidates them into a single table. The junk dimension table is denormalized and includes all possible combinations of the low-cardinality flags. Each row in the junk dimension corresponds to a unique combination of these flags. The junk dimension typically has a surrogate key that is used in the fact table to establish a relationship.

2.5 DEGENERATE DIMENSIONS

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. 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.



Fact Table with Degenerate Dimension.

2.6 MULTIVALUED DIMENSIONS AND BRIDGE TABLES

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.

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. 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. The admission and discharge flags are convenient attributes that allow the diagnosis profiles at the time of admission and discharge to be easily isolated.

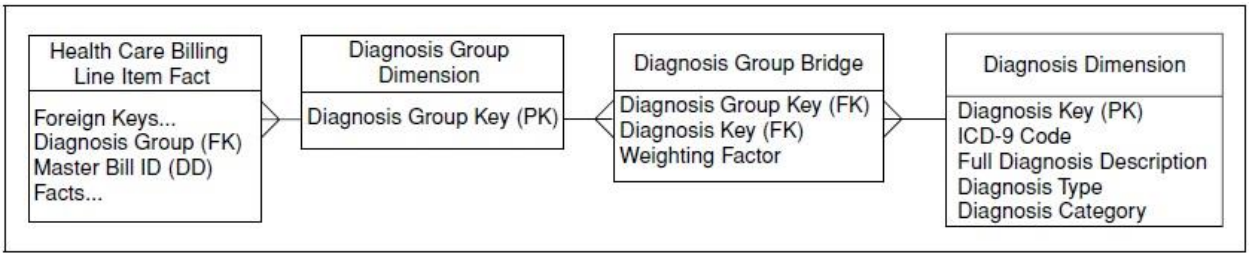
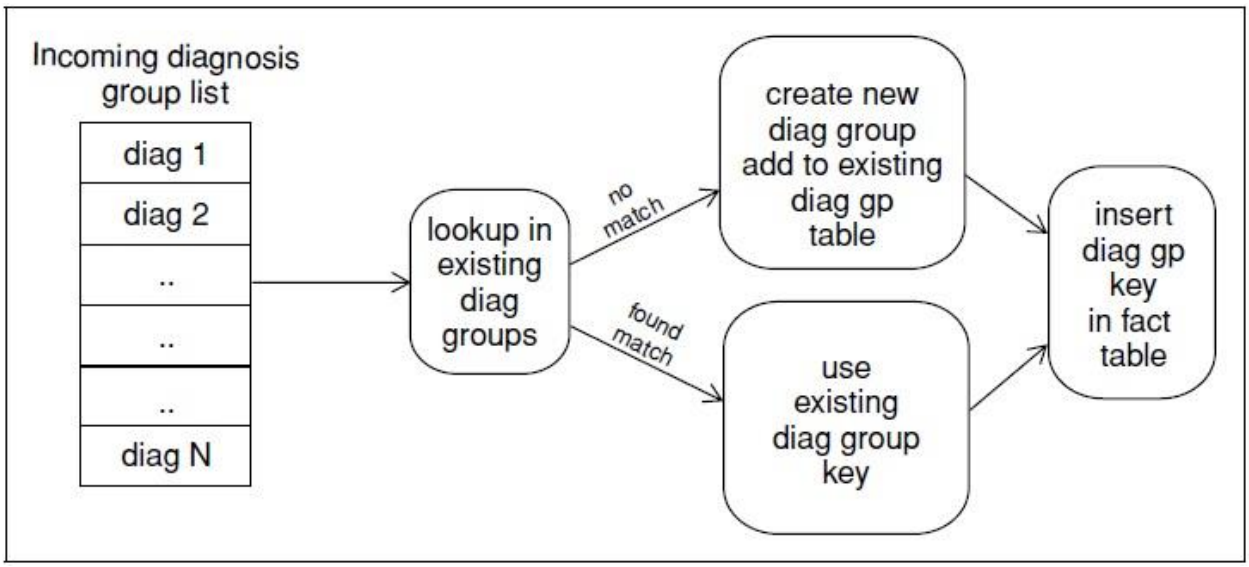


Figure 1 Using a bridge table to represent multiple diagnoses



Processing diagnosis groups in an outpatient setting

2.7 DIMENSION ENTITIES CLASSIFICATION – HISTORY CHANGED

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

2.7.1 Type 0: Retain Original

This technique hasn't been given a type number in the past, but it's been around since the beginning of SCDs. With type 0, the dimension attribute value never changes, so facts are always grouped by this original value. Type 0 is appropriate for any attribute labeled "original," such as customer original credit score. It also applies to most attributes in a date dimension.

The dimension table's primary key is a surrogate key rather than relying on the natural operational key. Although we demoted the natural key to being an ordinary dimension attribute, it still has special significance. Presuming it's durable, it would remain inviolate. Persistent durable keys are always type 0 attributes.

2.7.2 Type 1. Overwrite

The Type 1 SCD is a simple overwrite of one or more attributes in an existing dimension record. See Figure below. 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.

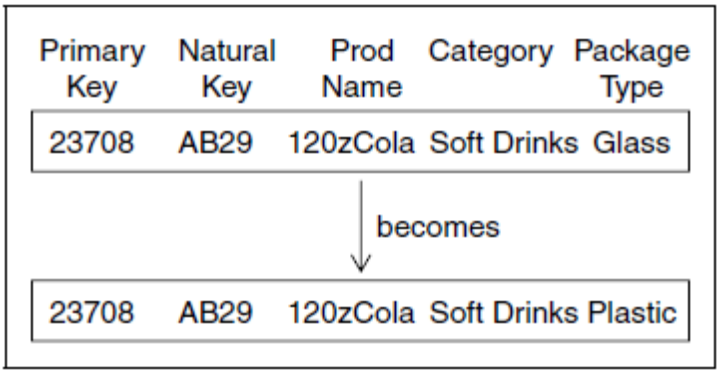


Figure 5 Processing Type-1 SCD.

Although inserting new records into a Type 1SCDrequires 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.

2.7.2.1 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.

2.7.3 Type 2. Add New Row

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.

Let say that the Type SCD 2 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.

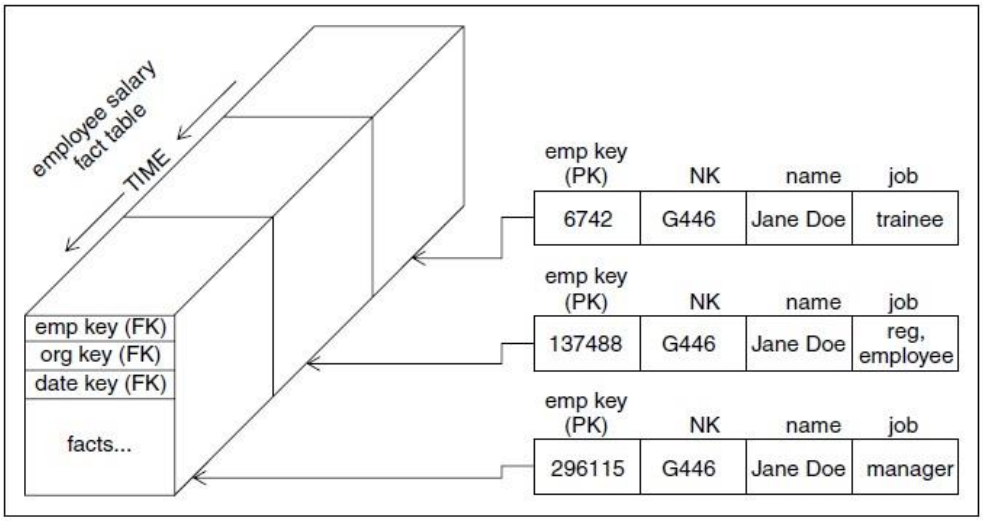


Figure 6 The Type 2 SCD

With type 2 changes, the fact table is again untouched; you don't go back to the historical fact table rows to modify the product key. In the fact table, rows for IntelliKidz (see Figure 7) prior to February 1, 2013, would reference product key 12345 when the product rolled up to the Education department. After February 1, new IntelliKidz fact rows would have product key 25984 to reflect the move to the Strategy department. This is why we say type 2 responses perfectly partition or segment history to account for the change. Reports summarizing pre-February 1 facts look identical whether the report is generated before or after the type 2 change.

Original row in Product dimension:

Product Key	SKU (NK)	Product Description	Department Name	...	Row Effective Date	Row Expiration Date	Current Row Indicator
12345	ABC922-Z	IntelliKidz	Education	...	2012-01-01	9999-12-31	Current

Rows in Product dimension following department reassignment:

Product Key	SKU (NK)	Product Description	Department Name	...	Row Effective Date	Row Expiration Date	Current Row Indicator
12345	ABC922-Z	IntelliKidz	Education	...	2012-01-01	2013-01-31	Expired
25984	ABC922-Z	IntelliKidz	Strategy	...	2013-02-01	9999-12-31	Current

Figure 7 Sample SCD 2 Rows

For a small dimension of a few thousand records and a dozen fields, such as a simple product file, the detection of changes 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.

***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.*

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.

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.

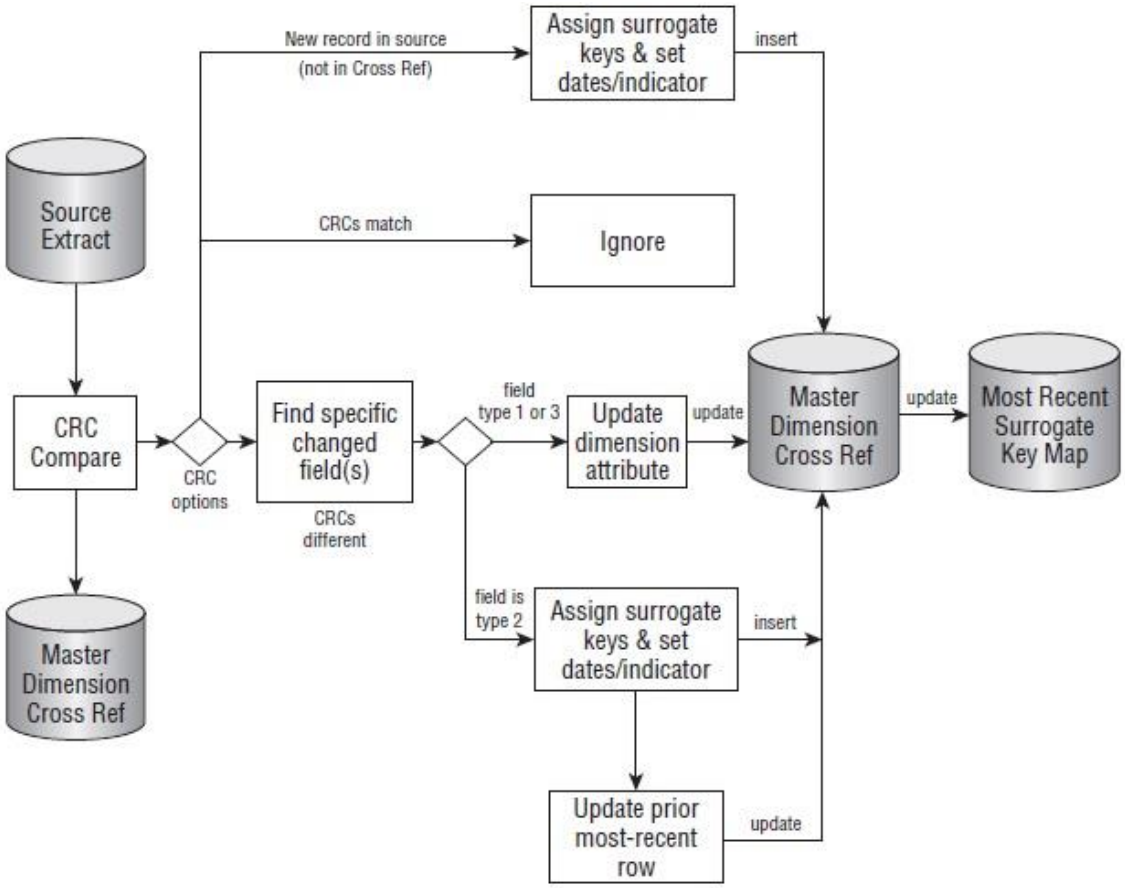


Figure 8 Processing flow for SCD surrogate key management.

2.7.3.1 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. Useful to 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)

2.7.4 Type 3.Add New Attribute

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 sales territory 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.

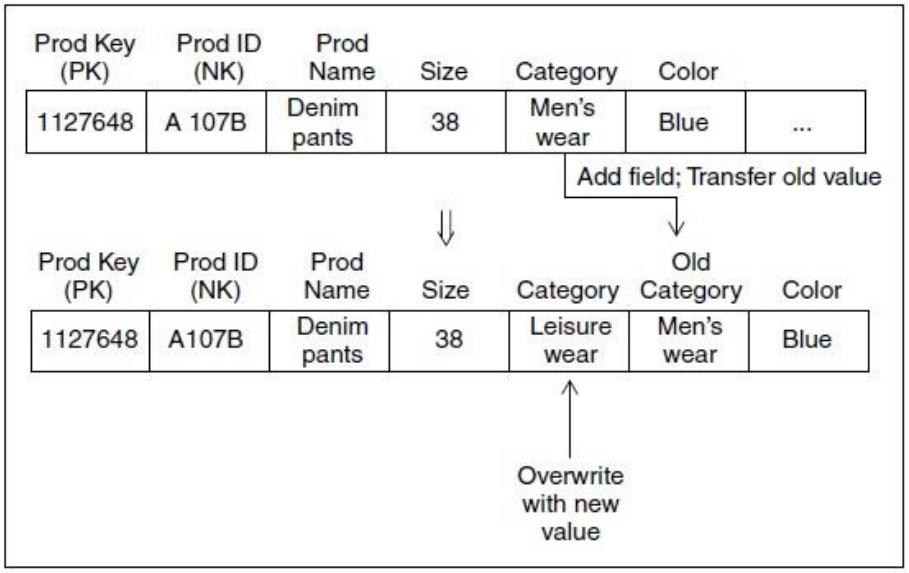


Figure 9 Type 3 SCD Implementation

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.

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.

2.7.5 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.

3 DIMENSION ENTITIES HIERARCHIES

Hierarchical many-to-one relationships are found in most dimension tables. Fortunately, most hierarchies are fixed depth, so they can be denormalized into columns on a flattened dimension table. Things get much more interesting (and complicated) when the hierarchical relationships are ragged with variable depths, as we explore in this section.

Dimensions are key to navigating the data warehouse, and hierarchies are the key to navigating dimensions. Often business users want to drill up or down into the data, they are implicitly referring to a dimension hierarchy. In order for those drill paths to work properly, those hierarchies must be correctly designed, cleaned, and maintained.

Hierarchies are important not just for usability. They play a huge role in query performance for a modern DW/BI system: Aggregations are often precomputed and stored for intermediate hierarchy levels and transparently used in queries. Precomputed aggregations are one of the most valuable tools to improve query performance, but in order for them to work, your hierarchies have to be clean.

3.1 START WITH THE DESIGN

The solution to the problem of maintaining hierarchies begins during the design phase. For every substantial dimension, spend time thinking through the hierarchical relationships. Business user input is absolutely imperative, as is time spent exploring the data.

The first question to resolve is what are the drilldown paths or hierarchies in each dimension? Most dimensions have a hierarchy, even if it's not coded in the transaction system. A core dimension such as customer, product, account, or even date may have many hierarchies. Date provides a good example that we all understand.

The date dimension often has three or more hierarchies. Novice dimensional modelers will try to create a single hierarchy that goes from day to week, month, quarter, and year. But that just doesn't work! Weeks do not roll up smoothly to months or even years. There is usually a separate fiscal calendar, and sometimes several others.

Display the hierarchies graphically to review them with the business users. Figure 10 shows clearly the different hierarchies and levels that will be available. Notice the attributes that apply at different levels. This picture is a graphical display suitable for communicating with users and among the DW/BI team; it does not represent the table's physical structure. Get user buy-in on the hierarchies, levels, and names. Equally important, test how much transformation you need to apply to the actual data in order to populate these hierarchical structures.

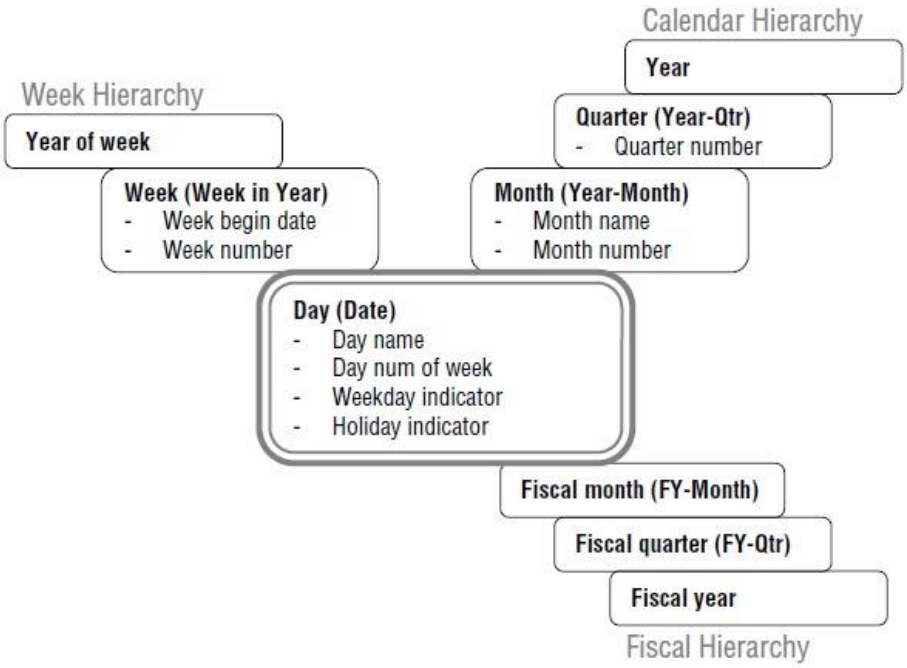


Figure 10 Graphical representation of multiple date hierarchies

The familiar date dimension contains lessons that are applicable to the administration of all dimensions:

- You can have multiple hierarchies. Most interesting dimensions have several alternative hierarchies. Work with business users to name columns and hierarchies so that the meaning of each is clear.
- You must have many-to-one referential integrity between each level: A day rolls up to one and only one month, month to quarter, and quarter to year.
- If the ETL system (as opposed to the original source) maintains referential integrity with explicit physical tables for each level, then a unique primary key must be identified at each level. If these keys are artificial surrogate keys, they should be hidden from the business users in the final single, flat denormalized dimension table in the presentation layer of the data warehouse. A common error is to think of the key for the month level as month name (January) or month number. The correct primary key is year and month. Likewise, in a location dimension, for example, city name alone is not an identifier column; it needs to be some combination of city, state, and perhaps country.
- Think carefully during the design phase about whether columns can be reused between hierarchies. You might think that the week hierarchy could share the year column with the calendar hierarchy, but what about the first and last weeks of the year? If our business rule is to have week 1 for a new year start on the first Monday of the year, week 1 of 2009 starts on January 5. January 1–4 will fall in 2008 for the week hierarchy. You need a separate year-of-week column. Sometimes you do want hierarchies to intersect, but you must be certain that the data will support that intersection.

3.2 LOAD NORMALIZED DATA

The date dimension hierarchies are easy to load and maintain. Nothing is more predictable than the calendar, and no user intervention is required.

If your source systems are imperfect, managing the hierarchies over time is painful. Optimally, hierarchies should be maintained before the data warehouse in the transaction system or a master data management (MDM) system. With good source data, the data warehouse will never see malformed data. In the real world, we're not always so lucky. Data warehouse teams have been managing master data for decades and in many organizations will continue to do so.

Consider a product dimension for a retail store, with a hierarchy that goes from product to brand, category, and department. In this example, the product hierarchy isn't officially part of the transaction systems, but instead is managed by business users in the marketing department. When we initially load the data warehouse, our incoming data is as illustrated in Figure 11.

SKU	Product Name	Brand Name	Category Name	Department Name	Dept Sq Foot
101	Baked Well Sourdough	Baked Well	Bread	Bakery	150
102	Fluffy Sliced Whole Wheat	Fluffy	Bread	Bakery	150
103	Fluffy Light Whole Wheat	Fluffy	Bread	Bakery	150
951	Fat Free Mini Cinnamon Rolls	Light	Sweeten Bread	Bakery	150
952	Diet Lovers Vanilla	Coldpack	Frozen Desserts	Frozen Foods	200
953	Strawberry Icy Creamy	Icy Creamy	Frozen Desserts	Frozen Foods	200
954	Icy Creamy Sandwiches	Icy Creamy	Frozen	Frozen Foods	200

Figure 11 Sample source data for a product dimension

The scenario described here is not ideal: This product dimension is not well maintained by the source systems. Most of it is fine, but notice the last two rows of data: We have a typo in the category, which breaks referential integrity. The “Icy Creamy” brand in one row rolls up to the Frozen Desserts category, and in another row to Frozen. This is forbidden.

You should find and fix problems like these early on before you even start building the ETL system. Your ETL system must implement checks to confirm that each category rolls to one department, and each brand to one category. But by the time you’re actually loading the historical data, you should have worked with the source systems and business users to fix the data errors.

The real challenge lies with ongoing updates of the dimension table. We don’t have time during nightly processing to have a person examine a suspect row and make an intelligent determination about what to do. If the data arriving at the ETL system’s door is suspect, the ETL system can’t distinguish between bad data and intentional changes. This is one of the hazards of developing a prototype or proof of concept. It’s easy to fix up the data on a one-time basis; keeping it clean over time is hard.

3.3 MAINTAIN TRUE HIERARCHIES

Clean source data is essential. True hierarchies are often maintained in normalized tables. Optimally, this maintenance occurs before the data warehouse proper, either in the source transaction system or a master data management system.

You can write an ETL process to move this nicely structured data into the dimension table; it’s a two-step process. Start at the top of the hierarchy (department), and perform inserts and updates into normalized tables in the staging area. Work down to the leaf level (product). Your staging tables will look similar to the structures in the sample product hierarchy table presented earlier. Once you’ve performed the extract step and have staged all the hierarchical data, write a query to join these tables together and perform standard dimension processing from the staging area into the data warehouse dimension.

The product dimension in the data warehouse should be denormalized into a single flattened dimension table. The normalization illustrated previously is the design pattern for the source system and ETL staging areas, not the actual dimension table that users query.

3.4 ADDRESS DIRTY SOURCES

Not everyone has a well-designed source system with normalized hierarchies as described in the preceding section. It's common in the DW/BI world for hierarchies to be managed by business users.

Transaction systems tend to have only enough information to do their job, and business users often have a legitimate need for alternative, richer rollups and attributes. What can you do?

- Modify the source systems. This is extraordinarily unlikely, unless your organization wrote those systems.
- Buy and implement an MDM system that manages the process of defining and maintaining hierarchies. This is the best solution, though MDM is expensive in terms of software license, and even more so in management commitment and attention.
- Write an applet to manage a specific user hierarchy. Keep your design simple, solving only the problem in front of you, such as the product hierarchy. If you get carried away, you'll find yourself developing what amounts to an MDM solution.

A true hierarchy has referential integrity between each of its levels. Remember that this is fundamentally a data quality issue that is enforced in the ETL back room or source systems; it's typically not carried into the presentation area as separate tables or snowflakes of tables. When a dimension has a true hierarchy, you gain two huge benefits:

- You will be able to define and maintain precomputed aggregations at intermediate levels of the hierarchy. In other words, you can precompute and store an aggregate at the month or brand level. Precomputed aggregations are one of the most important tools for improving query performance in the DW/BI system.
- You will be able to integrate data at different levels of granularity. Sometimes data naturally exists at an aggregate level. For example, our store might develop a long term sales forecast by month and category. We can create a subset dimension at the category level to associate with the forecast facts, and then join together actual and forecast sales, if and only if the product hierarchy is a true hierarchy.

3.5 MAKE IT PERFORM

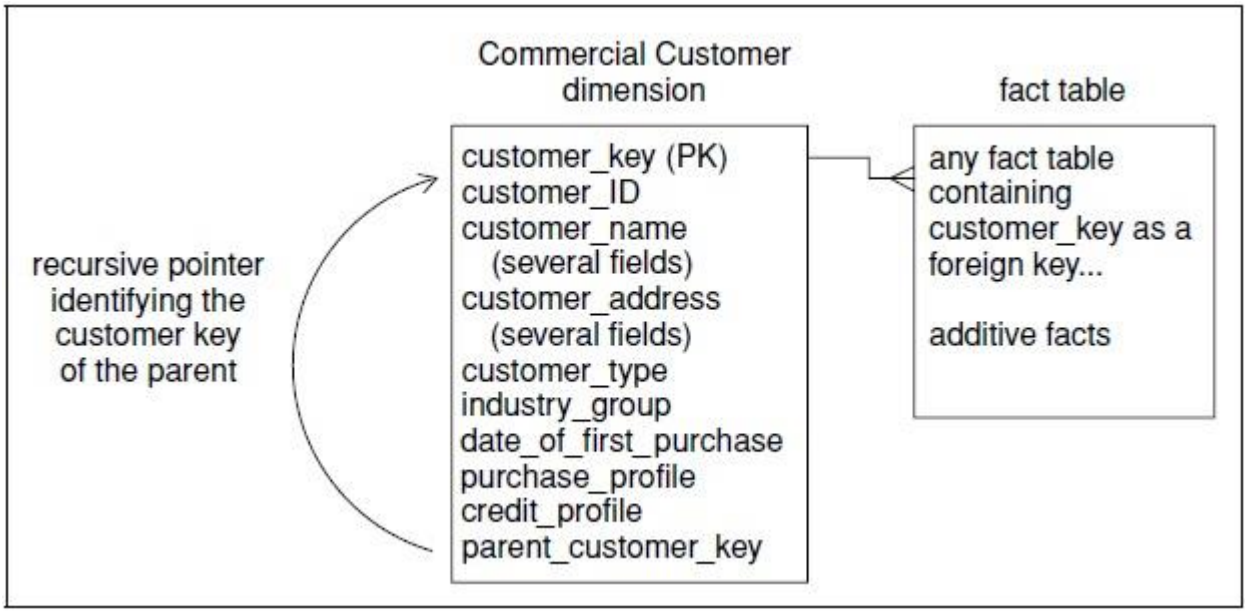
Those with large data warehouses, especially those with large dimensions, need to worry about dimension hierarchies. The performance benefits of precomputed aggregations are tremendous, and they will make or break the usability of the DW/BI system. To realize these benefits, you must implement procedures to maintain hierarchical information correctly in the source or master data management system.

In the meantime, users can benefit from navigation paths that look like hierarchies but really aren't. Business users have legitimate reasons for wanting to group information together, and it's our job to make that not just possible, but also easy and highly responsive.

3.6 RAGGED HIERARCHIES AND BRIDGE TABLES

Ragged hierarchies of indeterminate depth are an important topic in the data warehouse. Organization hierarchies 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 below.



A customer dimension with a recursive pointer

The recursive pointer approach:

(+) embeds the hierarchy relationships entirely in the customer dimension

(+) has simple administration for adding and moving portions of the hierarchy

(−) 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:

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

(+) 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

(−) 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.

(−) 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

4 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.

4.1 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, it 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. Some fact tables possess one or more special dimension like fields known as degenerate dimensions. Degenerate dimensions exist in the fact table, but they are not foreign keys, and they do not join to a real dimension. They are usually denoted with the notation DD.

4.2 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 1, 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

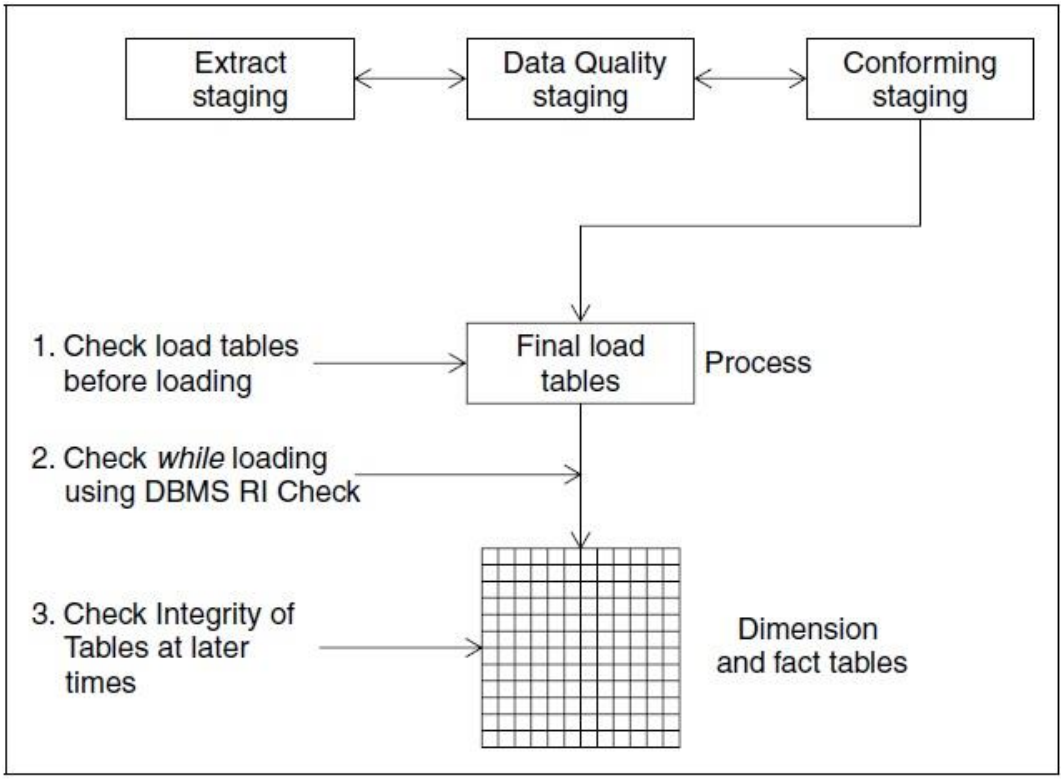


Figure 1 Choices for Enforcing Referential Integrity

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.

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.

4.3 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 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.

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.

Our task for processing the incoming fact table records is simple to understand. 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.

4.3.1 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 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.

5 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.

5.1 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.

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. 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.

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.

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. When the incoming transaction data arrives in a streaming fashion, rather than in discrete file-based loads, we have crossed the boundary into real-time data warehouses, which are discussed in Chapter 11.

5.2 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). 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.

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. These intermediate surrogate keys simply do not appear in the monthly periodic snapshot.

Periodic snapshot fact tables have completely predictable sparseness. As long as an account is active, an application can assume that the various dimensions will all be present in every query.

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. 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.

5.3 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.

Accumulating snapshot fact tables have several unusual characteristics. The most obvious difference 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 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.

The facts in the accumulating snapshot record are also revisited and overwritten as events unfold. Note that the actual width of an individual record depends on its contents, so accumulating snapshot records 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.

6 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.

6.1 MANAGING PARTITIONS

Partitions allow a table (and its indexes) to be physically divided into minitables 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 any database you receive the error.

So, 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.

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.

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.

6.2 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.

6.3 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.

6.4 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 misrepresent 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.

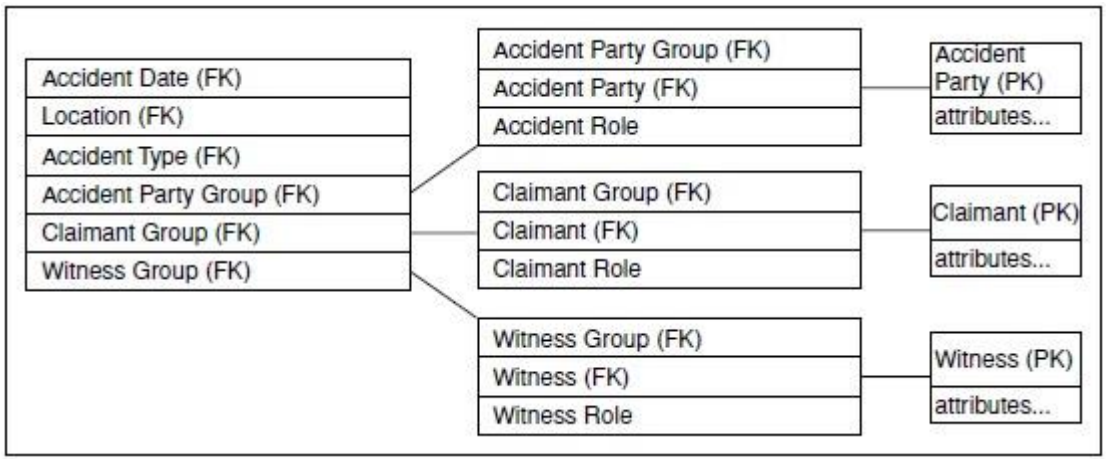
All three strategies result in a reflection of the actual occurrence—the sale of 1,000 20-ounce bottles of soda.

7 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. But after the dimensions are assembled, there may well be no measured fact. Event tracking frequently produces factless designs like this example.

Actually, this design 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.

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. 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.



Factless Fact Table

8 SOURCE BOOKS AND ARTICLES

1. Kimball R.,& Ross M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, Third Edition. Indianapolis: John Wiley & Sons, 2013.