|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.07 Dimension and Fact Basics |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Kiryl Bucha](mailto:Kiryl_Bucha@epam.com) | 12-JAN-2012 |  |  |
| 2.0 | Updated in accordance with renewed content | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |

Contents

[1. Dimension Entities Classification 3](#_Toc383290128)

[1.1. The Grain of a Dimension 4](#_Toc383290129)

[1.2. Date and Time Dimensions 4](#_Toc383290130)

[1.3. Big Dimensions 5](#_Toc383290131)

[1.4. Small Dimensions 6](#_Toc383290132)

[1.5. Degenerate Dimensions 7](#_Toc383290133)

[2. Dimension Entities Classification – History Changed 7](#_Toc383290134)

[2.1. Type 0: Retain Original 7](#_Toc383290135)

[2.2. Type 1. Overwrite 7](#_Toc383290136)

[2.2.1. Bulk Loading Type 1 Dimension Changes 8](#_Toc383290137)

[2.3. Type 2. Add New Row 8](#_Toc383290138)

[2.3.1. Precise Time Stamping of a Type 2 Slowly Changing Dimension 10](#_Toc383290139)

[2.4. Type 3.Add New Attribute 11](#_Toc383290140)

[2.5. Hybrid Slowly Changing Dimensions 11](#_Toc383290141)

[3. Dimension Entities Hierarchies 12](#_Toc383290142)

[3.1. Start with the Design 12](#_Toc383290143)

[3.2. Load Normalized Data 13](#_Toc383290144)

[3.3. Maintain True Hierarchies 14](#_Toc383290145)

[3.4. Address Dirty Sources 14](#_Toc383290146)

[3.5. Make It Perform 15](#_Toc383290147)

[4. Fact Entity 15](#_Toc383290148)

[4.1. The Basic Structure of a Fact Table 15](#_Toc383290149)

[4.2. Fundamental Grains 15](#_Toc383290150)

[4.2.1. Transaction Grain Fact Tables 16](#_Toc383290151)

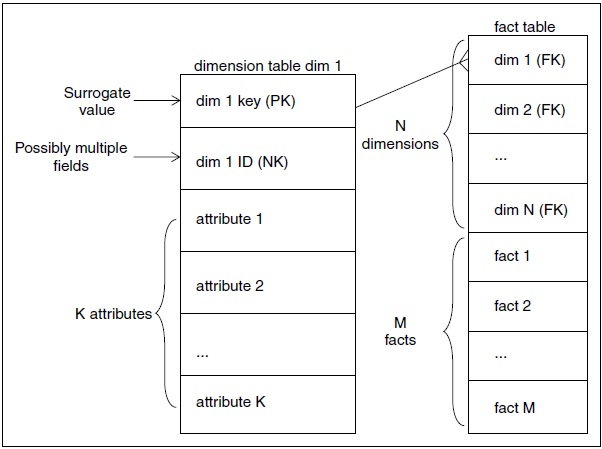
[4.2.2. Periodic Snapshot Fact Tables 16](#_Toc383290152)

[4.2.3. Accumulating Snapshot Fact Tables 16](#_Toc383290153)

[5. Source Books and Articles 17](#_Toc383290154)

# Dimension Entities Classification

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



**Figure 1 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.

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.

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.

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 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.

## Date and Time Dimensions

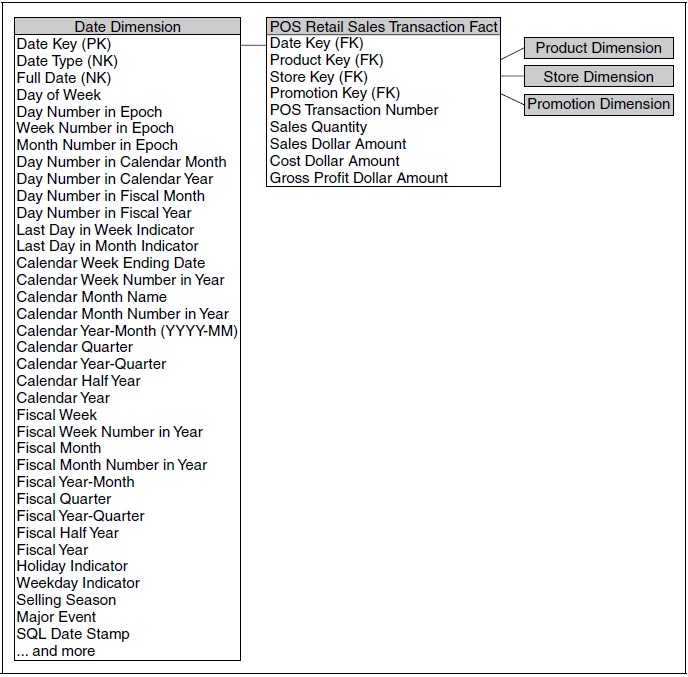
Virtually every fact table has one or more time-related dimension foreign keys. Measurements are defined at specific points and most measurements are repeated over time. The most common and useful time dimension is the calendar date dimension with the granularity of a single day. This dimension has surprisingly many attributes, as shown in Figure below on this chapter.

Only a few of these attributes (such as month name and year) can be generated directly from an SQL date-time expression. Holidays, work days, fiscal periods, week numbers, last day of month flags, and other navigational attributes must be embedded in the calendar date dimension and all date navigation should be implemented in applications by using the dimensional attributes. The calendar date dimension has some very unusual properties. It is one of the only dimensions completely specified at the beginning of the data warehouse project. It also doesn’t have a conventional source. The best way to generate the calendar date dimension is to spend an afternoon with a spreadsheet and build it by hand. Ten years worth of days is fewer than 4000 rows. Every calendar date dimension needs a date type attribute and a full date description attribute. These two fields compose the natural key of the table.

The date type attribute usually has the value date, but there must be at least one record that handles the special non applicable date situation where the recorded date is inapplicable, corrupted, or hasn’t happened yet. Foreign key references in fact tables referring to these special data conditions must point to a nondate date in the calendar date table! You need at least one of these special records in the calendar date table, but you may want to distinguish several of these unusual conditions. For the inapplicable date case, the value of the date type is inapplicable or NA.

The full date attribute is a full relational date stamp, and it takes on the legitimate value of null for the special cases described previously. Remember that the foreign key in a fact table can never be null, since by definition that violates referential integrity. The calendar date primary key ideally should be a meaningless surrogate key, but many ETL teams can’t resist the urge to make the key a readable quantity such as 20040718, meaning July 18, 2004. However, as with all smart keys, the few special records in the time dimension will make the designer play tricks with the smart key. For instance, the smart key for the inapplicable date would have to be some nonsensical value like 99999999, and applications that tried to interpret the date key directly without using the dimension table would always have to test against this value because it is not a valid date.

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



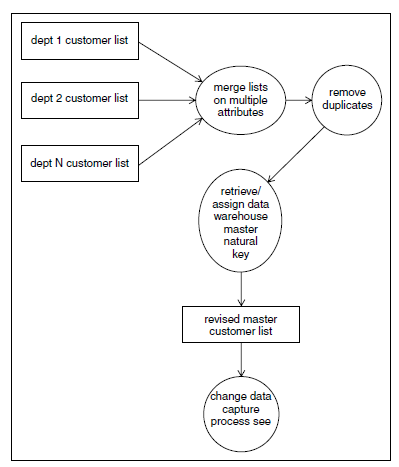
**Figure 2 Calendar Date Dimension.**

## 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.



**Figure 3 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 union into one big, wide dimension record.

## 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.

## 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.



**Figure 4 Fact Table with Degenerate Dimension.**

# 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).

## 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.

## 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.

### 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.

Example of bulk load Inserts:

create or replace procedure proc\_name(p\_n in number default 100)

as

type array is table of t2%rowtype;

l\_data array;

cursor c is select \* from all\_objects where rownum <= 5000;

begin

open c;

loop

fetch c bulk collect into l\_data LIMIT p\_n;

forall i in 1..l\_data.count

insert into t2 values l\_data(i);

exit when l\_data.COUNT <p\_n;

end loop;

close c;

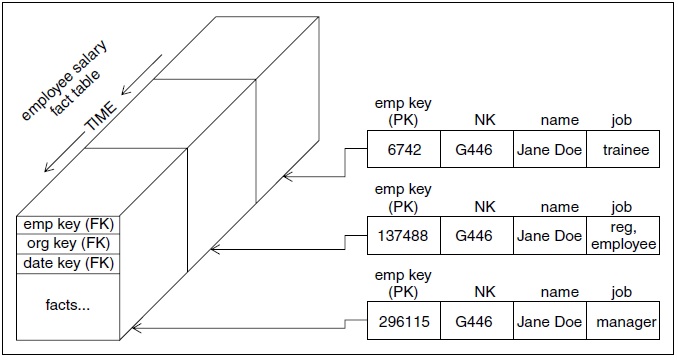
end;

/

## Type 2. Add New Row

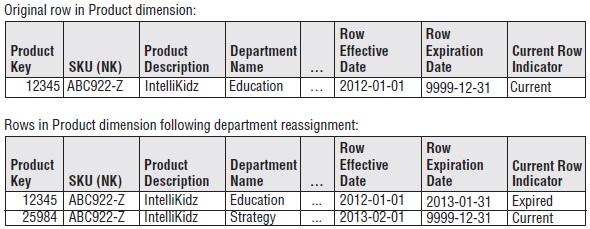
The Type 2 SCD is the standard basic technique for accurately tracking changes in dimensional entities and associating them correctly with fact tables. The basic idea is very simple. When the data warehouse is notified that an existing dimension record needs to be changed, rather than overwriting, the 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.



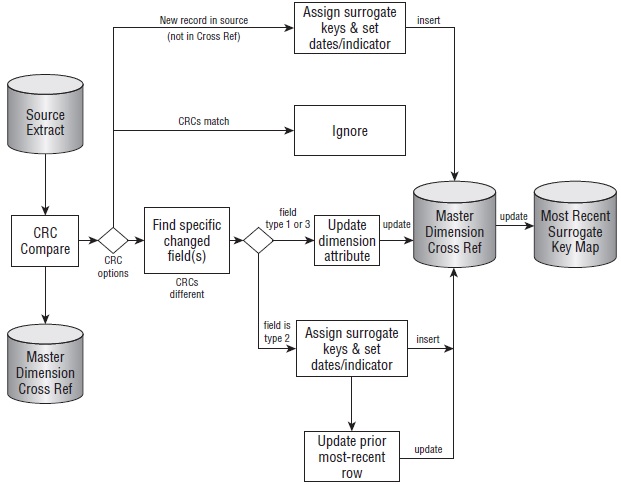
**Figure 7 Sample SCD 2 Rows**

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.**

### 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:

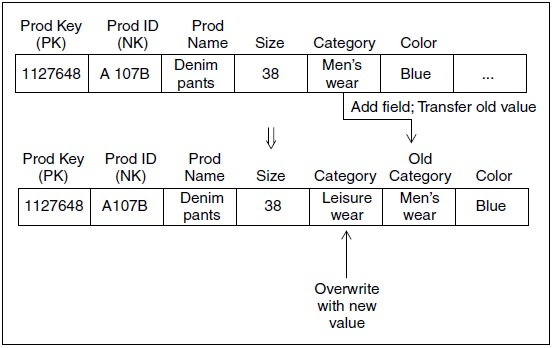
* Calendar Date foreign key (date of change)
* Row Effective DateTime (exact date-time of change)
* Row End DateTime (exact date-time of next change)
* Reason for Change (text field)
* Current Flag (current/expired)

## 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.

## 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.

# 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.

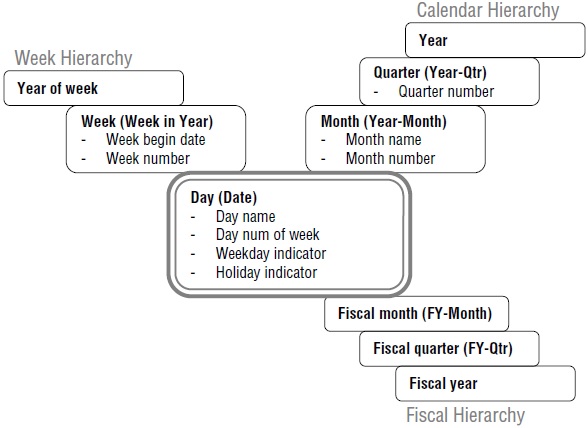
## Start with the Design

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:

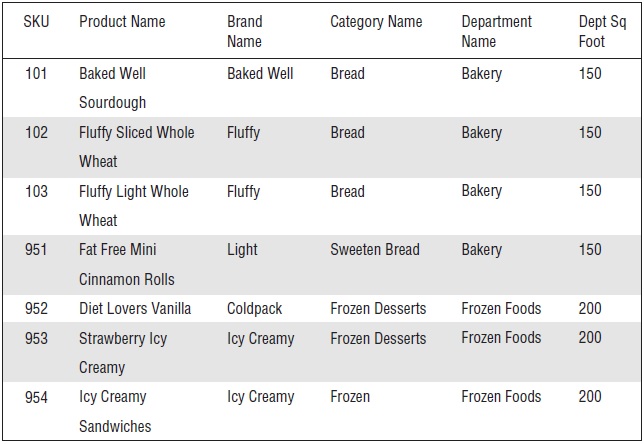
* You can have multiple hierarchies. Most interesting dimensions have several alternative hierarchies. Work with business users to name columns and hierarchies so that the meaning of each is clear.
* You must have many-to-one referential integrity between each level: A day rolls up to one and only one 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.

## 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.



**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.

## 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.

## 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.

## 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.

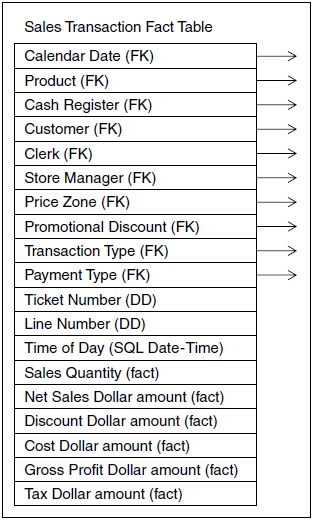
In the meantime, users can benefit from navigation paths that look like hierarchies but really aren’t. Business users have legitimate reasons for wanting to group information together, and it’s our job to make that not just possible, but also easy and highly responsive.

# Fact Entity

## 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. 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 calls facts.



**Figure 12 A sales transaction fact table at the lowest grain**

In practice, fact tables almost always have at least three dimensions, but most fact tables have more. As data warehousing and the surrounding hardware and software technology has matured over the last 20 years, fact tables have grown enormously because they are storing more and more granular data at the lowest levels of measurement.

## 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. Fact tables can always be reduced to just three fundamental types.

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.

### 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.

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.

### Periodic Snapshot Fact Tables

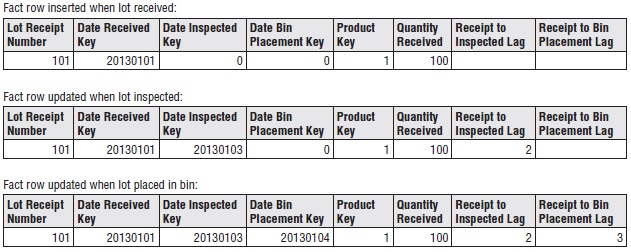
The periodic snapshot represents a span of time, regularly repeated. This style of table is well suited for tracking long-running processes such as bank accounts and other forms of financial reporting. The most common periodic snapshots in the finance world have a monthly grain. All the facts in a periodic snapshot must be true to the grain (that is, they must be measures of activity during the span).

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.

### 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.



**Figure 13 Evolution of an accumulating snapshot fact row**

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.

# Source Books and Articles

1. Powell G. Oracle Data Warehouse Tuning for 10g. Oxford: Elsevier Digital Press, 2005.
2. Kimball R., & Ross M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, Third Edition. Indianapolis: John Wiley & Sons, 2013.
3. Lane, P. Oracle Database Data Warehousing Guide, 11g Release 2 (11.2). Redwood City: Oracle, 2013.