|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.08 Fact Table Techniques |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Delivering Fact Tables 3](#_Toc384328539)

[1.1. The Basic Structure of a Fact Table 3](#_Toc384328540)

[1.2. Guaranteeing Referential Integrity 3](#_Toc384328541)

[1.3. Surrogate Key Pipeline 4](#_Toc384328542)

[1.3.1. Using the Dimension Instead of a Lookup Table 5](#_Toc384328543)

[2. Fundamental Grains 5](#_Toc384328544)

[2.1. Transaction Grain Fact Tables 5](#_Toc384328545)

[2.2. Periodic Snapshot Fact Tables 6](#_Toc384328546)

[2.3. Accumulating Snapshot Fact Tables 7](#_Toc384328547)

[3. Preparing for Loading Fact Tables 7](#_Toc384328548)

[3.1. Managing Partitions 8](#_Toc384328549)

[3.2. Incremental Loading 8](#_Toc384328550)

[3.3. Inserting Facts 9](#_Toc384328551)

[3.4. Updating and Correcting Facts 9](#_Toc384328552)

[4. Factless Fact Tables 9](#_Toc384328553)

[5. Source Books and Articles 10](#_Toc384328554)

# 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.

## 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 dimensionlike 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.

## Guaranteeing Referential Integrity

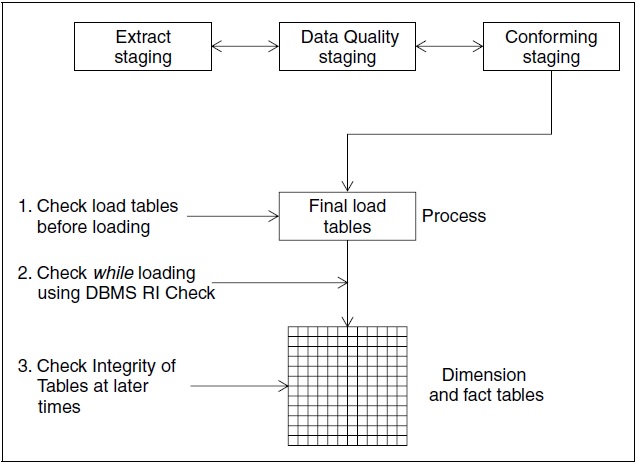
In dimensional modeling, referential integrity means that every fact table is filled with legitimate foreign keys. Or, to put it another way, no fact table record contains corrupt or unknown foreign key references. There are only two ways to violate referential integrity in a dimensional schema:

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.

## 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.

### 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.

# Fundamental Grains

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

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

## Transaction Grain Fact Tables

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

We have remarked that the tiny atomic measurements typical of transaction grain fact tables have a large number of dimensions.

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.

## 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.

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

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

# Preparing for Loading Fact Tables

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

## Managing 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 Oracle you receive the following error:

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

At this point, the ETL process has a choice:

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

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

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

compared with

select high\_value from all\_tab\_partitions

where table\_name = 'FACT\_TABLE'

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

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

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

ALTER TABLE fact\_table

ADD PARTITION year\_2005 VALUES LESS THAN (1828)

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

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

## Incremental Loading

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

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

## Inserting Facts

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

## 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.

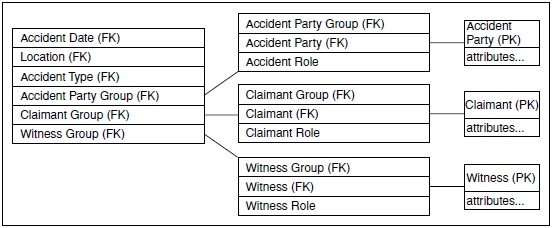
All three strategies result in a reflection of the actual occurrence—the sale of 1,000 20-ounce bottles of soda.

# 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.



**Figure 2 A Factless Fact Table**

# 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.