**Snowflake Schema**

Because of the **various levels of hierarchy**, data in a dimension table in Star schema contain duplicates or redundant values. Thus dimension tables are not **typically normalized**. There is no redundancy in the fact table - only in dimensions.

In contrast to relational databases, controlled redundancy is generally appropriate in multidimensional databases, if it increases the data's information value and query processing.
The **snowflake** schema is a variant of the **star schema**, where some dimension tables are normalized. The resulting schema forms a picture similar to the snowflake, and hence the name. The redundant attributes are removed from the flat, de-normalized dimension tables and placed in normalized secondary tables.

The major difference between the snowflake and star schemas is that the dimension tables in the snowflake schema may be kept in normalized form. This reduces redundancies and as such, the tables are easy to maintain. For example, if a department name is changed, it will require only to change in one place rather than in all occurrences of it in a single dimension table.

The normalized tables also save storage space, because a large dimension table can become enormous when the dimensional structure is included as columns. However, this saving of space is negligible as compared to the size of the fact table.

Figure below shows a partial snowflake schema through the expansion of **Product dimension** into multiple tables and their associated PK-FK relationships. Such an arrangement is normalized in the third normal form of an entity-relationship diagram.
A snowflake schema with multiple heavily snowflaked dimensions is illustrated in the following diagram.

**Family of Stars**

Sophisticated applications may require multiple fact tables to share dimension tables. This kind of schema can be viewed as a collection of stars, and often called as a *family of stars*, *galaxy schema* or *fact constellation*.
Advantages and Disadvantages of Snowflaking

By snowflaking, we may reduce redundancy in data values in the dimension tables and obtain a true normalized database. We may save some space in the dimension tables, but that is small compared to the overall size of the database which mainly comes from the large data volume of the fact table.

Advantages:
- Small savings in storage
- Normalized structures that are easier to update and maintain

Disadvantages:
- Schema less intuitive and end-users are put-off by the complicity
- Ability to browse through the contents difficult
- Degraded query performance because of additional joins

Star or Snowflake

Both star and snowflake schemas are dimensional models; the difference is in their physical implementations. Snowflake schemas support ease of dimension maintenance because they are more normalized. Star schemas are easier for direct user access and often support simpler and more efficient queries.

The decision to model a dimension as a star or snowflake depends on the nature of the dimension itself, such as how frequently it changes and which
of its elements change, and often involves evaluating tradeoffs between ease of use and ease of maintenance.

It is often easiest to maintain a complex dimension by snowflaking the dimension. By pulling hierarchical levels into separate tables, referential integrity between the levels of the hierarchy is guaranteed. OLAP services reads from a snowflaked dimension as well as, or better than, from a star dimension.

However, it is important to present a simple and appealing user interface (such as OLAP) to business users who are developing ad hoc queries on the dimensional database. It may be better to create a star version of the snowflaked dimension for presentation to the users.
Large Dimensions

A large dimension can be very deep containing a very large number of rows, or it may contain a large number of attributes. In either case, populating a dimension table should be done in a special way. In case of the large number of attributes, we may want to separate a large dimension into multiple smaller dimensions.

Large dimensions usually tend to have multiple hierarchies in their attributes. For example, a product dimension of a grocery store may form one hierarchy for the marketing department and another hierarchy for the finance department. OLAP tools can be used to represent different hierarchies of the same dimension.

Date and Time Dimensions

A date dimension with one record per day will suffice if users do not need time granularity finer than a single day. A date by day dimension table will contain 365 records per year (366 in leap years).

A separate time dimension table should be constructed if a fine time granularity, such as minute or second, is needed. See above. A time dimension table of one-minute granularity will contain 24X60 = 1,440 rows for a day, and a table of seconds will contain 24X60X60 = 86,400 rows for a day. If exact event time is needed, it should be stored in the fact table.

When a separate time dimension is used, the fact table contains one foreign key for the date dimension and another for the time dimension.
**Conforming Dimensions**

A dimension table may be used in multiple places if the data warehouse contains multiple fact tables, or contributes data to data marts. For example, a product dimension may be used with a sales fact table and an inventory fact table in the data warehouse, and also in one or more departmental data marts.

A dimension such as customer, time, or product that is used in multiple schemas is called a *conforming dimension* if all copies of the dimension are the same. Summarization data and reports will not correspond if different schemas use different versions of a dimension table.

**Use of Confirming Dimensions in Multiple Facts:** Multiple fact tables are used in data warehouses that address multiple business functions, such as sales, inventory, and finance. Each business function will typically have its own schema that contains a fact table, several conforming dimension tables, and some dimension tables unique to the specific business function.

![Dimensional Modeling Diagram](image)

**Data Sparsity in the Fact Table**

Sparsity of data typically occurs in the lowest level of granularity of a fact table, due to non-availability of a measure for a combination of the dimension keys.

A simple example might be a high volume retail store. A single store may carry 70,000 unique products, but in a given day a typical store sells only ten percent of those products. In a single week, the store may sell 15,000 unique products.
If we calculate the number of rows in the fact table for a chain with 100 stores, selling 7000 products a day, for 365 days, we will have:

\[ 100 \times 7000 \times 365 = 255,500,000 \] rows.

If we create an aggregate of product sales by store by week, we would expect that the number of rows in the aggregate table would be reduced by seven, or \( \frac{255,500,000}{7} = 36,500,000 \). This will not be the case due to the sparsity of data, because all stores do not sell the same products on the same day. The number of rows will be \( (100 \times 15000 \times 52) = 78,000,000 \), or double than expected.

**Measures in the Fact Tables**

The values that quantify facts are usually numeric, and are often referred to as *measures*. Measures are typically additive along all dimensions, such as *Quantity* in a sales fact table. A sum of Quantity by customer, product, time, or any combination of these dimensions results in a meaningful value.

**Additive and Non-additive Measures**: Some measures are not additive along one or more dimensions, such as *quantity-on-hand* in an inventory system or *price* in a sales system. Some measures can be added along dimensions other than the time dimension. These measures are sometimes referred to as semi-additive. For example, *quantity-on-hand* can be added along the Warehouse dimension to achieve the *total-quantity-on-hand*.

Measures that cannot be added along any dimension are truly non-additive. Non-additive measures can often be combined with additive measures to create new additive measures. For example, *Sale Price = Quantity* *Price*.

**Calculated Measures**: A calculated measure is a measure that results from applying a function to one or more measures, for example, the computed Extended Price value resulting from multiplying Quantity times Price. Other calculated measures may be more complex, such as profit, contribution to margin, allocation of sales tax, and so forth.

Calculated measures may be pre-computed during the load process and stored in the fact table, or they may be computed on the fly as they are used. Determination of which measures should be pre-computed is a design consideration.
Updates to the Dimension Tables

As businesses perform more and more transactions, more and more rows get added to the fact table. Thus fact table grows over time. Very rarely the rows in the fact table get updated. What about the dimension tables?

A characteristic of dimensions is that dimension data is relatively stable – data may be added as new products are released or customers are acquired, but data, such as the names of existing products and customers, changes infrequently.

However, business events do occur that cause dimension attributes to change, and the effects of these changes on the data warehouse must be managed (in particular, the potential effect of a change to a dimension attribute on how historical data is tracked and summarized).

**Slowly Changing Dimensions**

The "Slowly Changing Dimension" problem is a common one, particular to data warehousing. It applies to cases where the attribute for a record varies over time. Here is an example:

Christina is a customer with ABC Inc. She first lived in Chicago, Illinois. So, the original entry in the customer lookup table has the following record:

<table>
<thead>
<tr>
<th>Customer Key</th>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Christina</td>
<td>Illinois</td>
</tr>
</tbody>
</table>

At a later date, she moved to Los Angeles, California on January, 2003. How should this change be reflected in the customer table?

There are in general three ways to solve this type of problem, and they are categorized as follows:

- Type 1: Overwrite the dimension record
- Type 2: Add a new dimension record
- Type 3: Create new fields in the dimension record
**Type 1: Overwrite the Dimension Record**

In Type 1 Slowly Changing Dimension, the new information simply overwrites the original information. In other words, no history is kept. In the above example, after Christina moved to California, the new information replaces the new record, and we have the following table:

<table>
<thead>
<tr>
<th>Customer Key</th>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Christina</td>
<td>California</td>
</tr>
</tbody>
</table>

Advantages: This is the easiest way to handle the Slowly Changing Dimension problem, since there is no need to keep track of the old information.

Disadvantages: All history is lost. By applying this methodology, it is not possible to trace back in history. For example, in this case, the company would not be able to know that Christina lived in Illinois before.

When to use: Type 1 slowly changing dimension should be used when it is not necessary for the data warehouse to keep track of historical changes.

**Type 2: Add a Dimension Record**

In Type 2 Slowly Changing Dimension, a new record is added to the table to represent the new information. Therefore, both the original and the new record will be present. The new record gets its own primary key. Thus, in the previous example, we get the following table.

<table>
<thead>
<tr>
<th>Customer Key</th>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Christina</td>
<td>Illinois</td>
</tr>
<tr>
<td>1005</td>
<td>Christina</td>
<td>California</td>
</tr>
</tbody>
</table>

Advantages: This allows us to accurately keep all historical information.

Disadvantages: This will cause the size of the table to grow fast. In cases where the number of rows for the table is very high to start with, storage and performance can become a concern.

When to use: Type 2 slowly changing dimension should be used when it is necessary for the data warehouse to track historical changes.
Type 3: Create new fields

In Type 3 Slowly Changing Dimension, there will be two columns to indicate the particular attribute of interest, one indicating the original value, and one indicating the current value. There will also be a column that indicates when the current value becomes active. In our previous example, we will now have the following columns:

- Customer Key
- Name
- Original State
- Current State
- Effective Date

After Christina moved from Illinois to California, the original information gets updated, and we have the following table (assuming the effective date of change is January 15, 2003):

<table>
<thead>
<tr>
<th>Customer Key</th>
<th>Name</th>
<th>Original State</th>
<th>Current State</th>
<th>Effective Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Christina</td>
<td>Illinois</td>
<td>California</td>
<td>15-JAN-2003</td>
</tr>
</tbody>
</table>

Advantages: This does not increase the size of the table, since new information is updated. This allows us to keep some part of history.

Disadvantages: Type 3 will not be able to keep all history where an attribute is changed more than once. For example, if Christina later moves to Texas on December 15, 2003, the California information will be lost.

When to use: Type 3 slowly changing dimension should only be used when it is necessary for the data warehouse to track historical changes, and when such changes will only occur for a finite number of times.
**Rapidly Changing Dimensions**

A dimension is considered to be a rapidly changing if one or more of its attributes changes frequently in many rows. For a rapidly changing dimension, the dimension table can grow very large from the application of numerous type 2 changes.

The terms rapid and large are relative, of course. For example, a customer table with 50,000 rows and an average of 10 changes per customer per year will grow to about five million rows in 10 years, assuming the number of customers does not grow. This may be an acceptable growth rate. On the other hand, only one or two changes per customer per year for a ten million-row customer table will cause it to grow to hundreds of millions of rows in ten years.

Often, the correct solution for a dimension with rapidly changing attributes is to break the offending attributes out of the dimension and create one or more new dimensions. For example, an important attribute for customers might be their account status (good, late, very late, in arrears, suspended), and the history of their account status.

Over time many customers will move from one of these states to another. If this attribute is kept in the customer dimension table and a type 2 change is made each time a customer's status changes, an entire row is added only to track this one attribute. The solution is to create a separate `account_status` dimension with five members to represent the account states.
Fact Table Size

Fact tables are deep in rows with few attributes. To understand how deep a fact table can be, let us take a simple example of a grocery chain that has 300 stores and sells 40,000 products. In the lowest level of granularity, the data warehouse contains sales data by product, for each day, and for each store. It also keep data for product promotion. For the sake of simplicity, we consider the following limitations on the dimensions.

- Time dimension: 5 years = 5 X 365 = 1825
- Store dimension: 300 stores
- Product dimension: 4000 (out of 40,000) products are sold in each store daily
- Promotion dimension: a sold item may be in only one promotion in a store in a day

Thus, the total number of records = 1825X300X4000X1 = 2 billion.

In an actual data warehouse, the number of records in a fact table will be much more than this number.

Aggregate Fact Tables

Aggregates are pre-calculated summaries derived from the most granular fact table. There can be many summaries across different dimensional hierarchies. The summaries typically form a set of separate aggregate fact tables. It is important to note that unless data are kept in the most granular form, summations can not be performed over desired hierarchies.

The summary data are obtained through executing queries in the data warehouse database. These queries typically require selection of multiple records from the dimension tables and then summation and manipulation of hundreds and thousands of values from the base fact table.

Aggregate tables have fewer rows than the basic tables. Therefore, when most of the queries are run against the aggregate fact tables instead of the base fact table, a tremendous boost in performance is obtained in the data warehouse query performance.
Approaches to Aggregation

There are three approaches to aggregation: no aggregation, selective aggregation, or exhaustive aggregation. In some cases, the volume of data in the fact table will be small enough that performance is acceptable without aggregates; however this is not common in a data warehouse database.

The opposite extreme is exhaustive aggregation. This approach will produce optimal query results because a query can read the minimum number of rows required to return an answer. However, this approach is not normally practical due to the processing required to produce all possible aggregates and the storage required to store them.

In a simple sales example where the dimensions are product, sales geography, customer, and time, the number of possible aggregates is the number of levels in each hierarchy of each dimension multiplied together.

<table>
<thead>
<tr>
<th>Customer</th>
<th>Sales Geography</th>
<th>Product</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>All customers</td>
<td>Worldwide sales</td>
<td>All products</td>
<td>Year</td>
</tr>
<tr>
<td>Market</td>
<td>Country</td>
<td>Product line</td>
<td>Quarter</td>
</tr>
<tr>
<td>Customer</td>
<td>Region</td>
<td>Product</td>
<td>Month</td>
</tr>
<tr>
<td></td>
<td>District</td>
<td></td>
<td>Week</td>
</tr>
<tr>
<td></td>
<td>Office</td>
<td></td>
<td>Day</td>
</tr>
<tr>
<td></td>
<td>Salesperson</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total number of aggregates: \((3 \times 6 \times 3 \times 5) = 270\)
Examples of Aggregations

Depending on the user needs, there can be various ways of aggregations. Consider an example of a retail store consisting of three dimensions: store, product, and time. Each dimension has several hierarchies.

One-Way Aggregations: When we aggregate in the hierarchy of one dimension while keeping the other dimensions in the lowest level, a one-way aggregation is created. Examples of one-way aggregations on product dimension for the above retail store are:

- Sale of product by category by store by date
- Sale of product by department by store by date
- Sale of all products by store by date
Two-Way Aggregations: When we rise to the higher levels in the hierarchies of two dimensions while keeping the other dimensions in the lowest level, a two-way aggregation is created. Examples of two-way aggregations on product and store dimensions for the simple retail store are:

- Sale of product category by territory by date
- Sale of product department by region by date
- Sale of all products in all stores by date

Three-Way Aggregations: When we rise to the higher levels in the hierarchies of three dimensions while keeping the other dimensions in the lowest level, a three-way aggregation is created. Examples of three-way aggregations on product, store, and time dimensions for the simple retail store are:

- Sale of product by category by territory by month
- Sale of product by department by region by quarter
- Sale of all products in all stores by year

Each of these aggregates forms an aggregate fact table. These derived fact tables are joined to one or more derived dimension tables. See figure below for an example of one-way aggregation on the product dimension. There can be many possible aggregates just for three-dimensional hierarchies. In the real world, there are many more dimensions and many more hierarchies in a dimension, providing hundreds of aggregates.
Choosing Aggregates

Usage and Analysis Patterns: There are two basic pieces of information which are required to select the appropriate aggregates. Probably the most important item is the expected usage patterns of the data. Based on this information from the users, it is possible to determine the most frequently examined levels and they will be the good candidates for aggregation.

Base Table Row Reduction: The second piece of information to consider is the data volumes and distributions in the fact table. Queries can be run to get an idea of the number of rows at various levels in the dimension hierarchies. This will tell us where there are significant decreases in the volume of data along a given hierarchy. Some of the best candidates for aggregation will be those where the row counts decrease the most from one level in a hierarchy to the next.

The decrease of rows in a dimension hierarchy is not a hard rule due to the distribution of data along multiple dimensions. When you combine the fact rows to create an aggregate at a higher level, the reduction in the number of rows may not be as much as was expected. This may be due to the sparsity of data in the fact table, as discussed before.

Since we are trying to reduce the number of rows a query must process, one of the key procedures is finding aggregates where the intersection of dimensions has a significant decrease in the number of rows. Figure below shows the row counts for all possible aggregates of product by store by day using one year of data for a 200 store retail grocer.
Looking at the chart, it is apparent that creating aggregates at some of the lowest levels (such as product-district, store-subcategory) will provide minimal performance improvement. Depending on the frequency of usage, there are several likely candidates. Any of the subcategory level aggregates provide a significant reduction in volume and would be good starting points for exploration.

The subcategory-by-store aggregate provides a very significant drop over the detail data, and will probably be small enough that all higher level product and geography queries may be satisfied by this aggregate.

One thing to keep in mind is that it is appealing to decide based on what you can see in the chart, but there are still tens of millions of rows in some of the higher levels.

**Aggregate Storage**

Once you have made an initial decision about which aggregates to create, the next question is, how to create and store those aggregates.

*Storing Aggregate Fact Rows*

There are three basic options for storing the aggregated data which are diagrammed in the figure below: (i) create a combined fact and aggregate table which will contain both the base level fact rows and the aggregate rows, (ii) create a single aggregate table which holds all aggregate data for a single fact table, and lastly, (iii) create a separate table for each aggregate created.
The combined fact and aggregate table approach is appealing, but it usually results in a very large and unmanageable table. The single aggregate table is almost as unmanageable. Both approaches suffer from contention problems during query and update, issues with data storage for columns which are not valid at higher levels of aggregation, and the possibility of incorrectly summarizing data in a query.

Thus, the third approach is most appropriate, that is to create a separate table for each aggregate. There is also an advantage of having independent aggregation tables, that is they can be easily removed and created as necessary. They also simplify keying of the aggregates and provide easier management.

**Storing Aggregate Dimension Rows**

A big issue encountered when storing aggregates is how the dimensions will be managed. Normally the dimensions contain one row for each discrete item. For example, a product dimension has a single row for each product manufactured by the company. The question arises, "how do you store information about hierarchies so the fact and aggregate tables are appropriately keyed and queried?"

No matter how the dimensions and aggregates are handled, the aggregate rows will require new keys. This is because the levels in a dimension hierarchy are not actually elements of the dimension. They are constructs above the detail level within the dimension. This is easily seen if we look at the company geography dimension described below.

<table>
<thead>
<tr>
<th>Key</th>
<th>Level</th>
<th>Store</th>
<th>District</th>
<th>Region</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Store</td>
<td>123</td>
<td>A</td>
<td>NW</td>
<td>...</td>
</tr>
<tr>
<td>145</td>
<td>Store</td>
<td>145</td>
<td>B</td>
<td>NW</td>
<td>...</td>
</tr>
<tr>
<td>168</td>
<td>Store</td>
<td>168</td>
<td>C</td>
<td>NE</td>
<td>...</td>
</tr>
<tr>
<td>1001</td>
<td>District</td>
<td>###</td>
<td>A</td>
<td>NW</td>
<td>...</td>
</tr>
<tr>
<td>1002</td>
<td>District</td>
<td>###</td>
<td>B</td>
<td>NW</td>
<td>...</td>
</tr>
<tr>
<td>9001</td>
<td>Region</td>
<td>###</td>
<td>###</td>
<td>NW</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The granularity of the fact table is product by store by day. This means the base level in the geography dimension is the store level. All fact rows will have as part of their key the store key from a row in this dimension. The hierarchy in the dimension is: store, district, region, all stores. There is no
row available in the dimension table describing a district or region. We must create these rows and provide keys for them. These keys can't duplicate any of the base level keys already present in the dimension table.

This can be done in several ways. The preferred method is to store all of the aggregate dimension records together in a single table. This makes it simple to view dimension records when looking for particular information prior to querying from the fact or aggregate tables.

There is one issue with the column values if all the rows are stored in a single table like this. When adding an aggregate dimension record there will be some columns for which no values apply. For example, a district level row in the geography dimension will not have a store number. This is shown below.

When you wish to create a pick list of values for a level in the hierarchy you can issue a SELECT DISTINCT on the column for that level. An alternative to this method is to include a level column which contains a single value for each level in the hierarchy. Then queries for a set of values for a particular level need only select where the level column is the level required.

Other methods for storing the aggregate dimension rows include using a separate table for each level in the dimension, normalizing the dimension, or using one table for the base dimension rows and a separate table for the hierarchy information. The last approach is most appropriate, even though it doubles the original number of dimension tables. The aggregate dimension rows can use a new key structure since they are no longer under the column constraints imposed by the base level dimension.