Week 2 - Designing, Modeling and Implementing Data Warehouses

Tipo	Anotaciones
Posición	
Etiquetas	
Bibliografía	
Fecha de creación	@June 22, 2022 9:43 PM
Property	

▼ Designing, Modeling and Implementing Data Warehouses

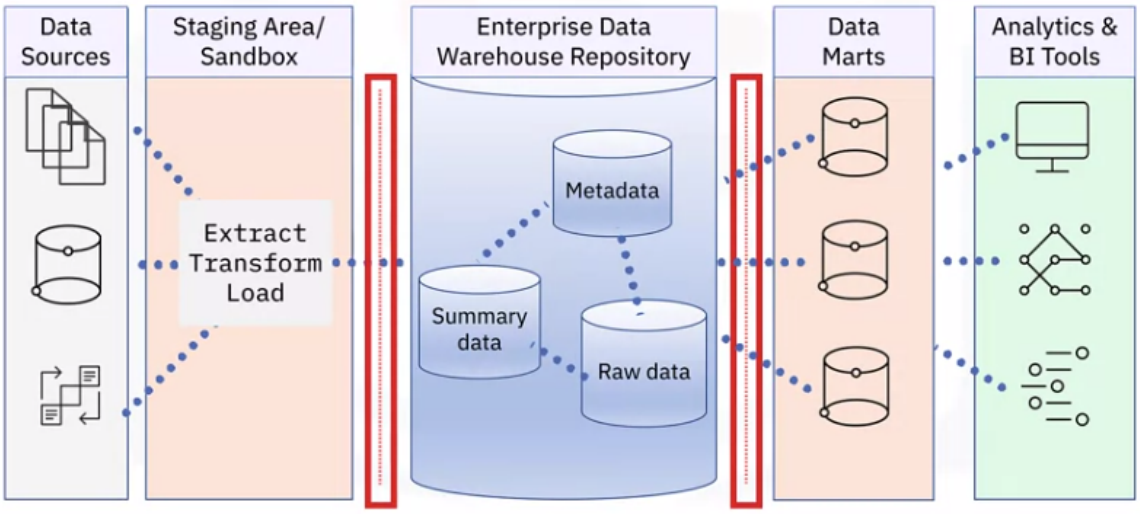
▼ Overview of Data Warehouse Architectures

Data warehouse architecture

Data warehouse architecture details depend on use cases:

- Report generation and dashboarding
- Exploratory data analysis
- Automation and machine learning
- Self-serve analytics

General EDM architecture

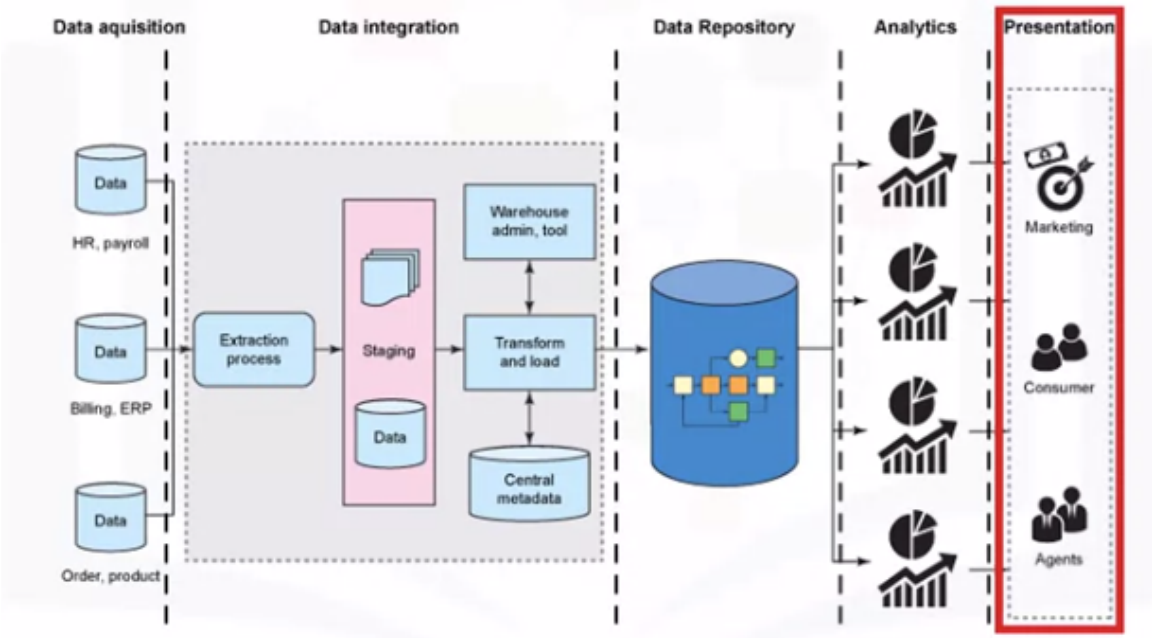


- **Data sources**, such as flat files, databases, and existing operational systems,.
- **ETL layer** for extracting, transforming, and loading data, optional staging and sandbox areas for holding data and developing workflows.
- **Enterprise Data Warehouse repository**.
- **Data Marts** (sometimes), which are known as a “hub and spoke” architecture when multiple data marts are involved.
- **Analytics layer** and business intelligence tools.
- Data warehouses also enforce **security** for incoming data and data passing through to further stages and users throughout the network.

EDW reference architecture

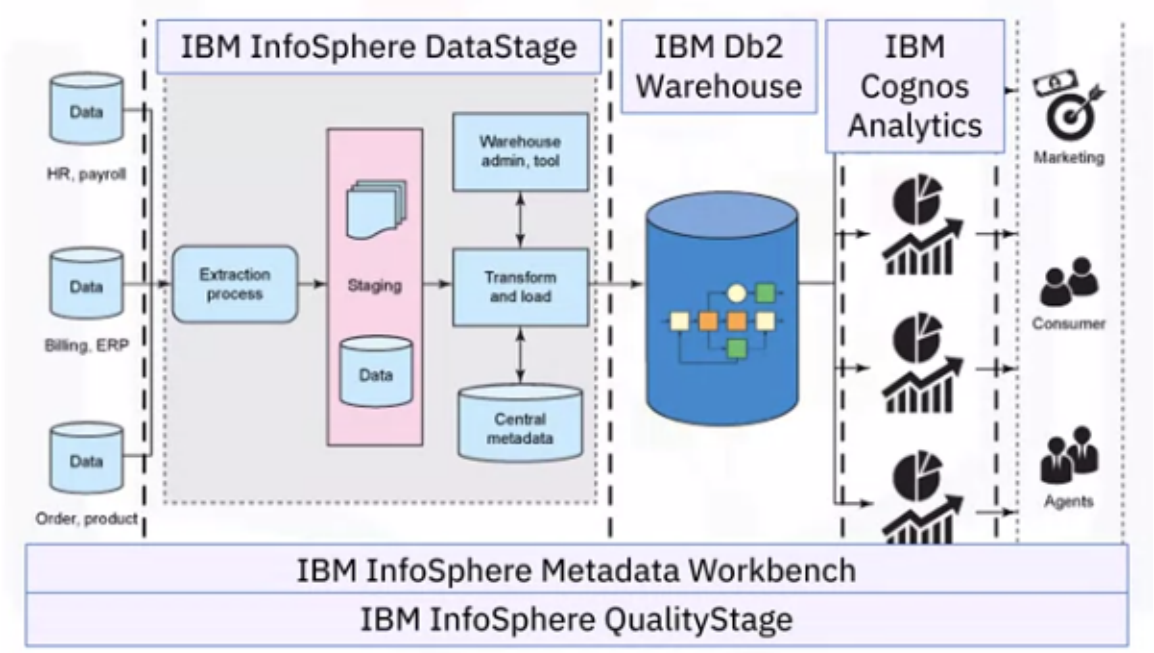
Enterprise data warehouse vendors often create proprietary reference architecture and implement template data warehousing solutions that are variations on this general architectural model. A data warehousing platform is a complex environment with lots of moving parts. Thus, interoperability among components is vital. Vendor-specific reference architecture typically incorporates tools and products from the vendor’s ecosystem that work well together.

IBM reference architecture



- The **data acquisition layer** consists of components to acquire raw data from source systems, such as human resources, finance, and billing departments.
- The **data integration layer**, essentially a staging area, has components for extracting the data, transforming it, and loading it into the data repository layer. It also houses administration tools and central metadata.
- The **data repository layer** stores the integrated data, typically employing a relational model.
- The **analytics layer** often stores data in a cube format to make it easier for users to analyze it.
- **Presentation layer** incorporates applications that provide access for different sets of users, such as marketing analysts, users, and agents.

Applications consume the data through web pages and portals defined in the reporting tool or through web services. IBM reference architecture is supported and extended using several products from the IBM InfoSphere suite.

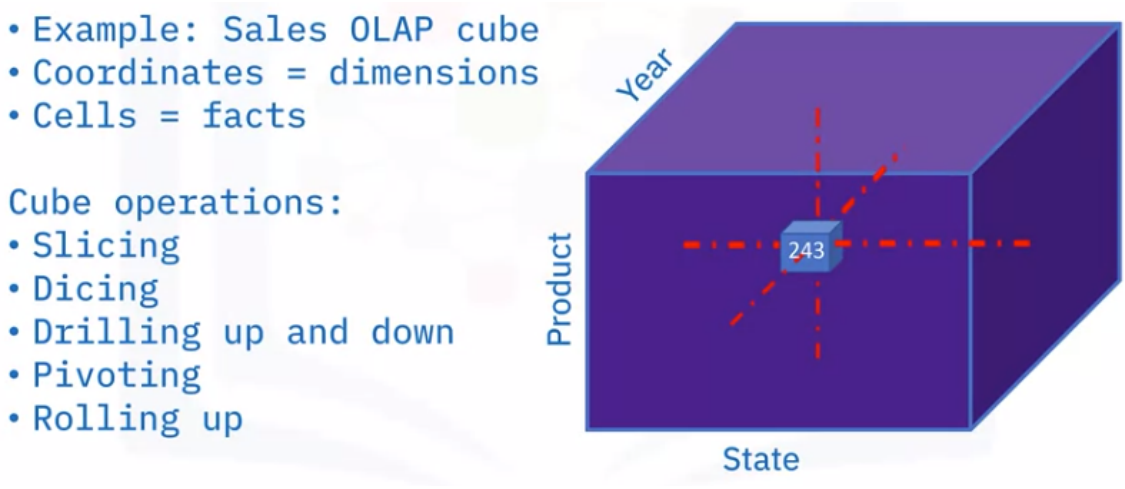


- **IBM InfoSphere DataStage** is a scalable ETL platform that delivers near real-time integration of all data types, on-premises, and in cloud environments.
- **IBM InfoSphere MetaData Workbench** provides end-to-end data flow reporting and impacts analysis of information assets in an environment that allows organizations to share easily, locate, and retrieve information from these systems. Use the built-in data flow reporting capabilities to monitor how IBM InfoSphere DataStage moves and transforms your data.
- **IBM InfoSphere QualityStage**, designed to support your data quality and information governance initiatives, enables you to investigate, cleanse, and manage your data. This solution helps you create and maintain consistent views of key entities, including customers, vendors, locations, and products.
- **IBM Db2 Warehouse** is a family of highly performant, scalable, and reliable data management products that manage both structured and unstructured data across on-premises and cloud environments.
- **IBM Cognos Analytics** is an advanced business intelligence platform that generates reports, scoreboards, and dashboards, performs exploratory data analysis, and even curates and joins your data using multiple sources.

▼ Cubes, Rollups, and Materialized Views and Tables

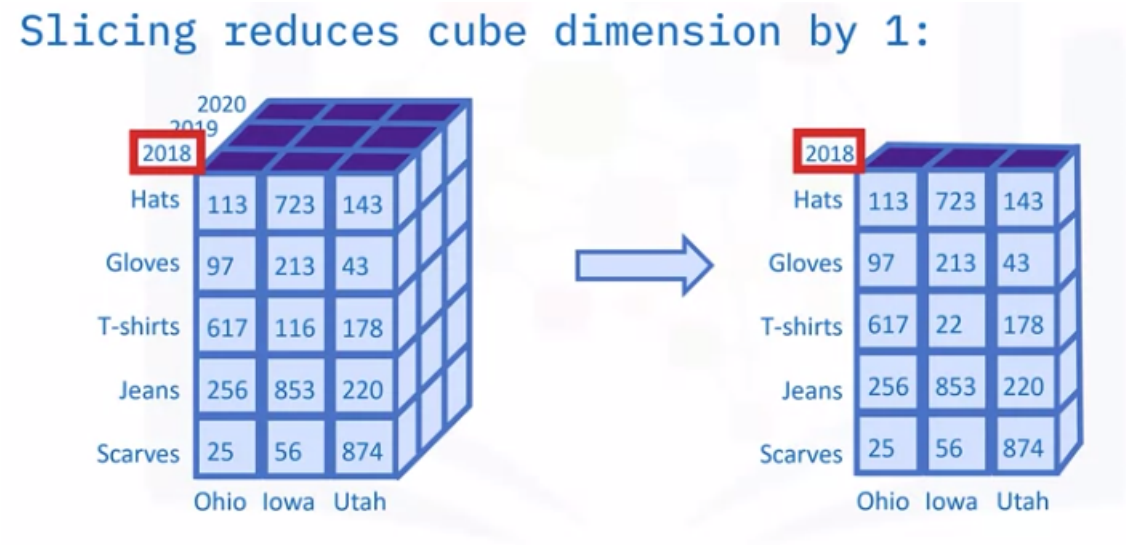
What is a data cube?

Here is a cube generated from an imaginary star schema for a Sales OLAP (online analytical processing system). The coordinates of the cube are defined by a set of dimensions, which are selected from the star schema.

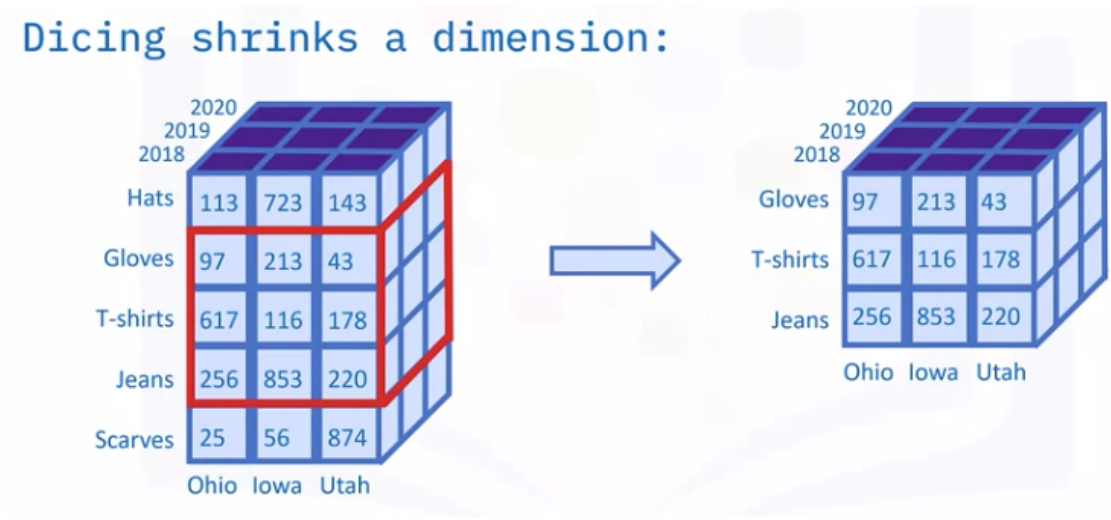


The cells of the cube are defined by a fact of interest from the schema, which could be something like “total sales in thousands of dollars.” Here the “243” indicates “243 thousand dollars” for some given Product, State, and Year combination.

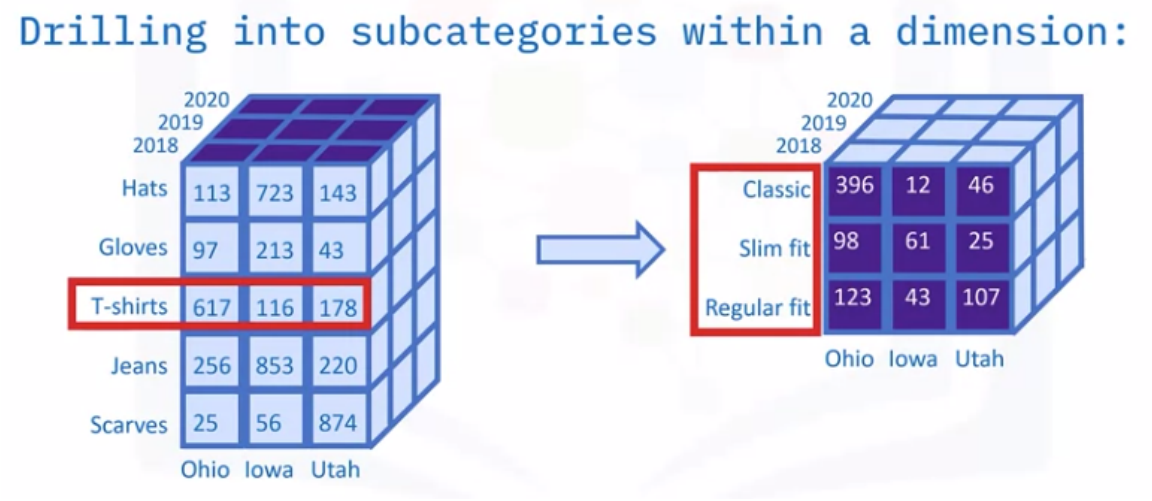
Slicing data cubes



Dicing data cubes

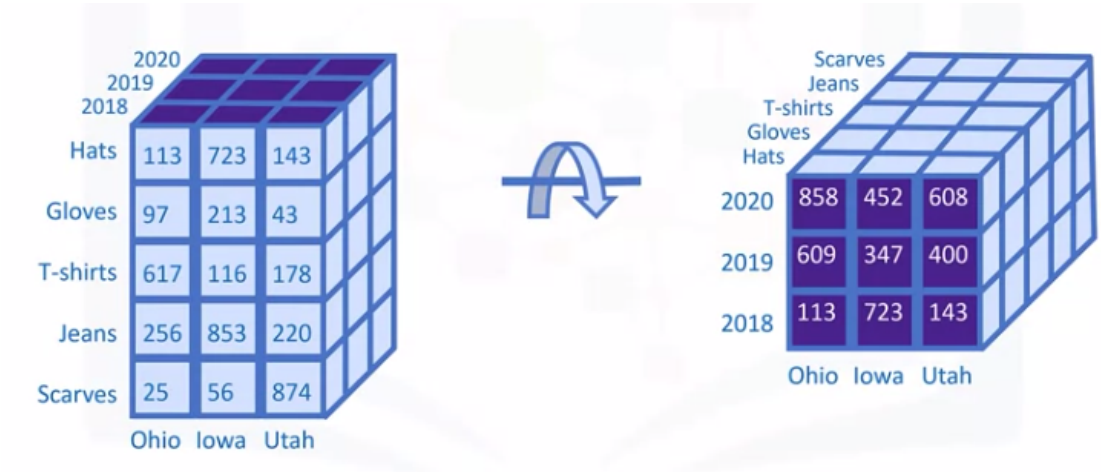


Drilling up or down in data cubes



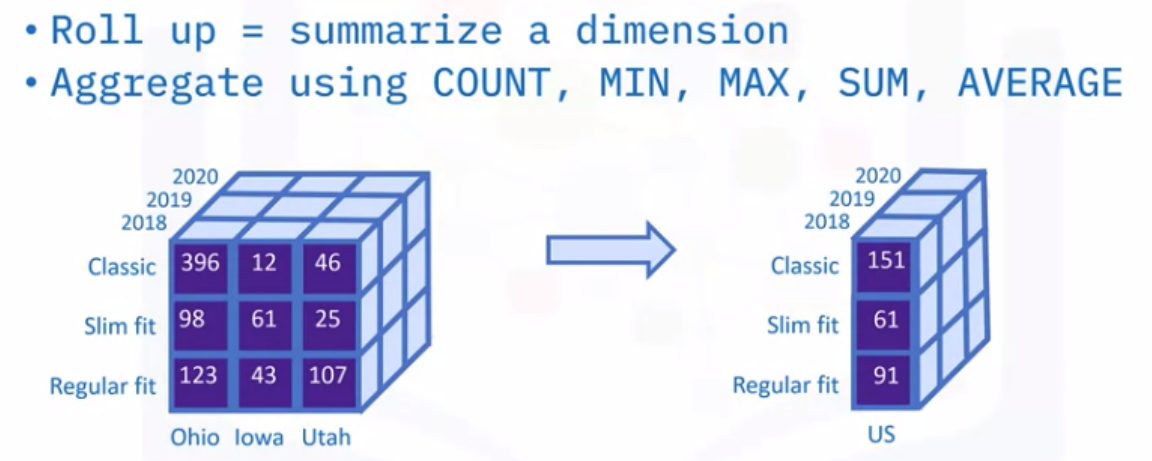
Drilling up is just the reverse process, which would take you back to the original data cube.

Pivoting data cubes



Pivoting doesn't change its information content; it just changes the point of view you may choose to analyze it from.

Rolling up in data cubes



Materialized views

- A “materialized view” is essentially a local, read-only copy, or snapshot, of the results of a query.
- They can be used to replicate data, for example to be used in a staging database as part of an ETL process, or to precompute and cache expensive queries, such as joins or aggregations, for use in data analytics environments.
- Materialized views also have options for automatically refreshing the data, thus keeping your query up-to-date.
- Because materialized views can be queried, you can safely work with them without worrying about affecting the source database.

Materialized Views can be set up to have different refresh options, such as:

- Never: they are only populated when created, which is useful if the data seldom changes.
- Upon request: manually refresh, for example, after changes to the data have been made, or scheduled refresh, for example, after daily data loads.
- Immediately: automatically refresh after every statement.

Materialized view in Oracle

```
CREATE MATERIALIZED VIEW MY_MAT_VIEW
REFRESH FAST START WITH SYSDATE
```

```
NEXT SYSDATE + 1
AS SELECT * FROM <my_table_name>;
```

Materialized view in PostgreSQL

```
CREATE MATERIALIZED VIEW MY_MAT_VIEW
[ WITH (storage_parameter [= value] [, ... ]) ]
 [ TABLESPACE tablespace_name ]
  AS SELECT * FROM <table_name>;

-- In PostgreSQL you can only refresh materialized views manually
REFRESH MATERIALIZED VIEW MY_MAT_VIEW;
```

Materalized view in Db2

In Db2, materialized views are called MQTs, which stands for "materialized query tables."

```
create table emp as (select e.empno, e.firstnme, e.lastname, e.phoneno, d.deptno,
  substr(d.deptname, 1, 12) as department, d.mgrno from employee e, department d
    where e.workdept = d.deptno)
      data initially deferred refresh immediate

immediate checked not incremental
```

The table, which is named “emp,” is based on the underlying tables: “Employee” and “Department” from the “Sample” database. The table will be created according to the query formed by these SQL statements, which selects columns from both tables. The “data initially deferred” clause means that data will not be inserted into the table as part of the “create table” statement, while the “refresh immediate” clause specifies that the query should refresh automatically. The “immediate checked” clause specifies that the data is to be checked against the MQT’s defining query and refreshed. Lastly, the “not incremental” clause specifies that integrity checking is to be done on the whole table.

A query executed against the “emp” materialized query table shows that it is fully populated with data.

```
select * from emp

EMPNO  FIRSTNME     LASTNAME        PHONENO DEPTNO DEPARTMENT   MGRNO
------ ------------ --------------- ------- ------ ------------ ------
000010 CHRISTINE    HAAS            3978    A00    SPIFFY COMPU 000010
000020 MICHAEL      THOMPSON        3476    B01    PLANNING     000020
000030 SALLY        KWAN            4738    C01    INFORMATION  000030
000050 JOHN         GEYER           6789    E01    SUPPORT SERV 000050
000060 IRVING       STERN           6423    D11    MANUFACTURIN 000060

5 record(s) selected. connect reset
```

▼ Reading: Grouping Sets in SQL

The GROUPING SETS clause is used in conjunction with the GROUP BY clause to allow you to easily summarize data by aggregating a fact over as many dimensions as you like.

SQL GROUP BY clause

Recall that the SQL GROUP BY clause allows you to summarize an aggregation such as SUM or AVG over the distinct members, or groups, of a categorical variable or dimension.

You can extend the functionality of the GROUP BY clause using SQL clauses such as CUBE and ROLLUP to select multiple dimensions and create multi-dimensional summaries. These two clauses also generate grand totals, like a report you might see in a spreadsheet application or an accounting style sheet. Just like CUBE and ROLLUP, the SQL GROUPING SETS clause allows you to aggregate data over multiple dimensions but does not generate grand totals.

Examples

Let’s start with an example of a regular GROUP BY aggregation and then compare the result to that of using the GROUPING SETS clause. We’ll use data from a fictional company called Shiny Auto Sales. The schema for the company’s warehouse is displayed in the entity-relationship diagram in Figure 1.

Fig. 1. Entity-relationship diagram for a "sales" star schema based on the fictional "Shiny Auto Sales" company.

We'll work with a convenient materialized view of a completely denormalized fact table from the sales star schema, called DNsales, which looks like the following:

This DNsales table was created by joining all the dimension tables to the central fact table and selecting only the columns which are displayed. Each record in DNsales contains details for an individual sales transaction.

Example 1

Consider the following SQL code which invokes GROUP BY on the auto class dimension to summarize total sales of new autos by auto class.

The result looks like this:

Example 2

Now suppose you want to generate a similar view, but you also want to include the total sales by salesperson. You can use the GROUPING SETS clause to access both the auto class and salesperson dimensions in the same query. Here is the SQL code you can use to summarize total sales of new autos, both by auto class and by salesperson, all in one expression:

Here is the query result. Notice that the first four rows are identical to the result of Example 1, while the next 5 rows are what you would get by substituting salespersonname for autoclassname in Example 1.

Essentially, applying GROUPING SETS to the two dimensions, salespersonname and autoclassname, provides the same result that you would get by appending the two individual results of applying GROUP BY to each dimension separately as in Example 1.

▼ Facts and Dimensional Modeling

Facts and dimensions

- Data can be categorized as facts and dimensions
- Facts are usually measured quantities, such as temperature, number of sales, or mm of rainfall
- Facts can also be qualitative
- Dimensions are attributes relating to facts
- Dimensions provide context to facts:
    - “24°C” is not useful information by itself

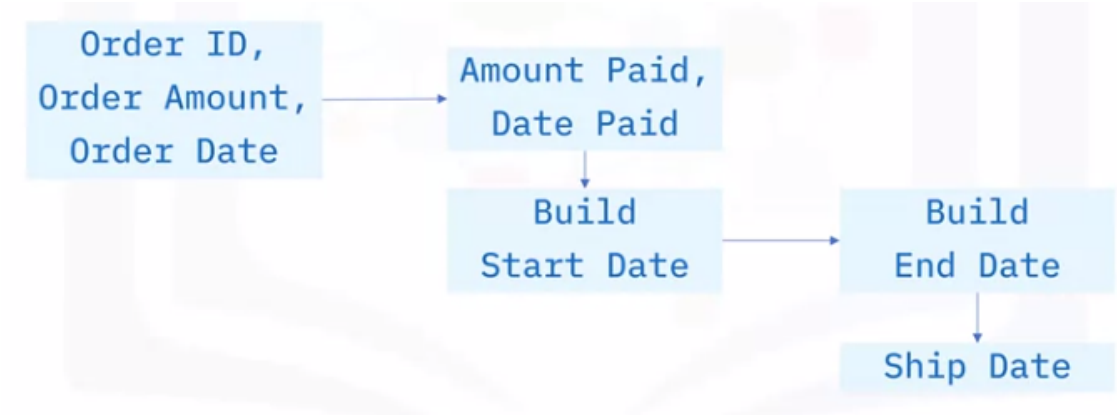
Fact tables

A fact table typically consists of the facts of a business process, and it also contains foreign keys which establish well-defined links to dimension tables. Usually, facts are additive measures or metrics, such as dollar amounts for individual sales transactions. A fact table can contain detail level facts, such as individual sales transactions, or facts that have been aggregated, such as daily or weekly sales totals. Fact tables that contain aggregated facts are called summary tables. For example, you could summarize sales transactions by summing overall sales for each quarter of the year. An example of a foreign key might be ‘store ID.’

Accumulating snapshot fact tables

“Accumulating snapshot” fact tables are used to record events that take place during a well-defined business process. For example, suppose you have finished configuring a custom computer for yourself online, and you have just placed your order. The order date and the order amount are recorded in a snapshot table by the manufacturer. A unique “order ID” is also assigned. Once your order is

verified, your payment is processed. The “amount paid” and the “date paid” are then recorded. After payment verification, the computer specifications are sent to the manufacturing department. Once the computer is in production, the “build start-date” is entered, and once the computer is built, the “build end-date” is recorded. Finally, once your computer is ready to be sent, the “ship date” is recorded. All these fields are stored in a single row of the table, which is uniquely identified by the “order ID.”



Dimensions

A dimension is a variable that categorizes facts, Dimensions are called “**categorical variables**” by statisticians and machine learning engineers. Dimensions enable users to answer business questions. The main uses for dimensions in analytics include filtering, grouping, and labeling operations. For example, commonly used dimensions include names of people, products, and places, and date or time stamps. A **dimension table** thus stores the dimensions of a fact and is joined to the fact table via a foreign key.

Dimension table examples

Product tables:

- Make, model, color, size

Employee tables:

- Name, title, department

Temporal tables:

- Date/time at granularity of recorded events

Geography tables:

- Country, state, city, region

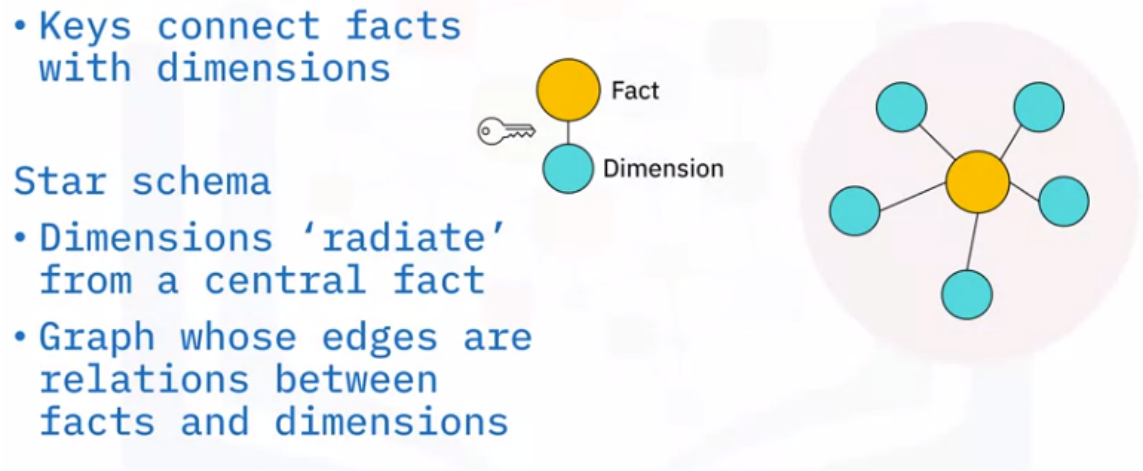
Example schema with fact & dimension tables

We could have a **fact** table for recording sales at a car dealership This table would store facts about each car sale such as the Sale date, and Sale amount as well as a primary key “Sale ID.” For each sales transaction, we also need to record its **dimensions** like the Vehicle sold and the Salesperson who sold it. The attributes of these dimensions, such as the vehicle Make and Model or the Salesperson's First and Last name are stored in separate Vehicle and Salesperson tables However, we link them with the Sales table by recording the “Vehicle ID” and “Salesperson ID” as foreign keys in the Fact table. This way we typically end up with multiple dimension tables for each fact table.

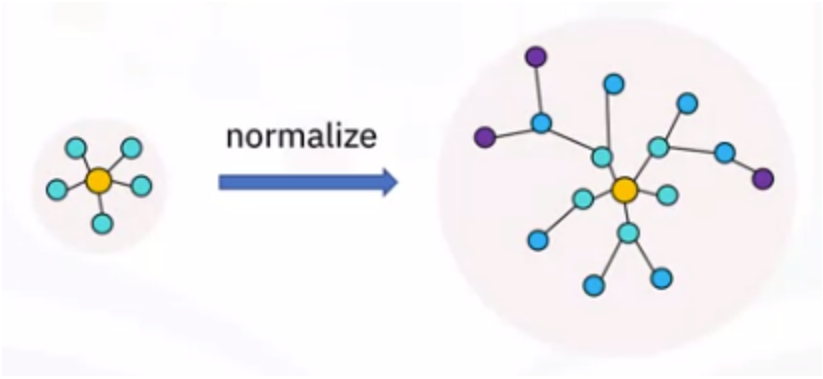
▼ Data Modeling using Star and Snowflake Schemas

Star schemas

The idea of a star schema is based on the way a set of dimension tables can be visualized, or modeled, as radiating from a central fact table, linked by these keys. A star schema is thus a graph, whose nodes are fact and dimension tables, and whose edges are the relations between those tables. Star schemas are commonly used to develop specialized data warehouses called “data marts.”



Snowflake schemas



Snowflake schemas are a generalization of star schemas and can be seen as normalized star schemas. Normalization means separating the levels or hierarchies of a dimension table into separate child tables. A schema need not be fully normalized to be considered a snowflake, so long as at least one of its dimensions has its levels separated.

Modeling with a star schema

General principles you need to consider when designing a data model for a star schema.

- Select a business process as the basis for what you want to model. You might be interested in processes such as sales, manufacturing, or supply chain logistics.

- Choose a granularity, which is the level of detail that you need to capture. Are you interested in coarse-grained information such as annual regional sales numbers? Or, maybe you want to drill down into monthly sales performance by salesperson.
- Identify the dimensions. These may include attributes such as the date and time, and names of people, places, and things.
- Identify the facts. These are the things being measured in the business process.

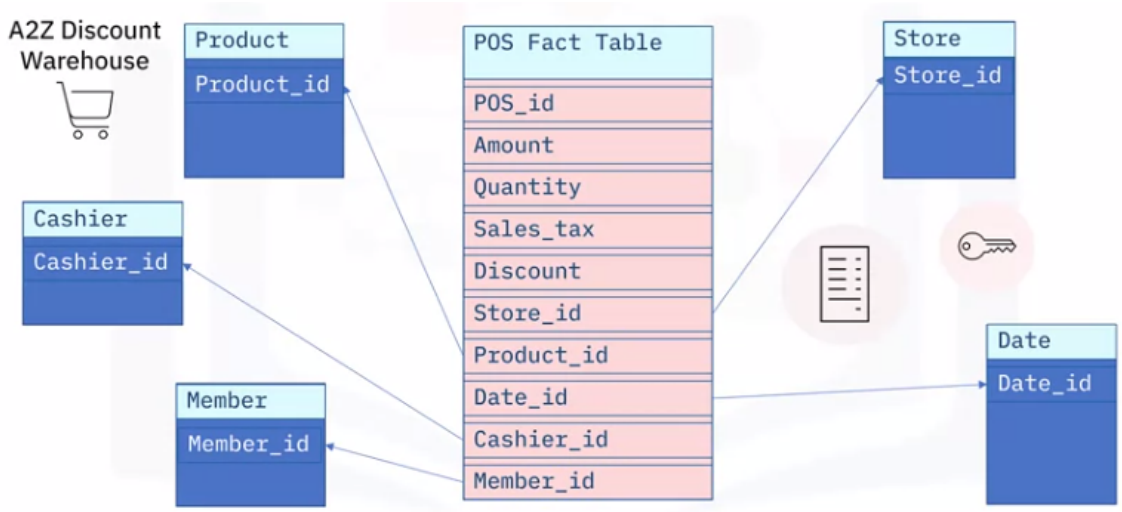
Data Warehouse Architecture

Example: that you are a data engineer helping to lay out the data ops for a new store called “A to Z Discount Warehouse.”

They would like you to develop a data plan to capture everyday POS, or point-of-sales transactions that happen at the till, where customers have their items scanned and pay for them. Thus, “point-of-sale transactions” is the business process that you want to model. The finest granularity you can expect to capture from POS transactions comes from the individual line items, which is included in the detailed information you can see on a typical store receipt. This is precisely what “A to Z” is interested in capturing. The next step in the process is to identify the dimensions. These include attributes such as the date and time of the purchase, the store name, the products purchased, and the cashier who processed the items. You might add other dimensions, like “payment method,”

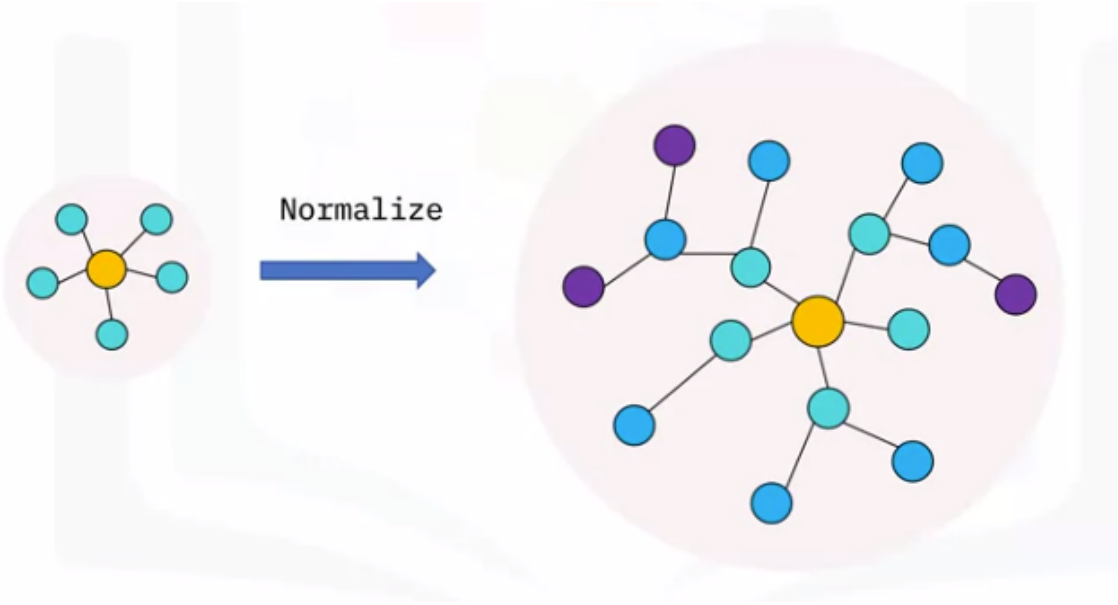
whether the line item is a return or a purchase, and perhaps a “customer membership number.” Now it’s time to consider the facts. Thus, you identify facts such as the amount for each item’s price, the quantity of each product sold, any discounts applied to the sale, and the sales tax applied. Other facts to consider include environmental fees, or deposit fees for returnable containers.

Point-of-sale-star



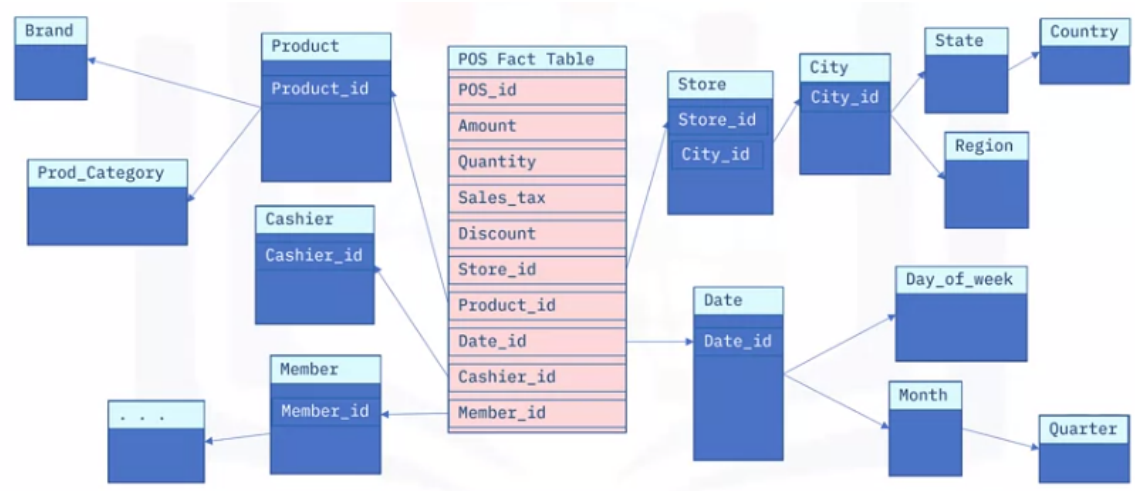
From star to snowflake

You can use normalization to extend your star schema to a snowflake schema.



Point-of-sale snowflake

Starting with your star schema, you can extract some of the details of the dimension tables into their own separate dimension tables, creating a hierarchy of tables.



Much like how pointers are used to point to memory locations in computing, normalization reduces the memory footprint of the data.

▼ Data Warehousing with Star and Snowflake schemas

Why do we use these schemas, and how do they differ?

Star schemas are optimized for reads and are widely used for designing data marts, whereas snowflake schemas are optimized for writes and are widely used for transactional data warehousing. A star schema is a special case of a snowflake schema in which all hierarchical dimensions have been denormalized, or flattened.

Attribute	Aa Star schema	Snowflake schema
Read speed	Fast	Moderate

Attribute	Star schema	Snowflake schema
Write speed	Moderate	Fast
Storage space	Moderate to high	Low to moderate
Data integrity risk	Low to moderate	Low
Query complexity	Simple to moderate	Moderate to complex
Schema complexity	Simple to moderate	Moderate to complex
Dimension hierarchies	Denormalized single tables	Normalized over multiple tables
Joins per dimension hierarchy	One	One per level
Ideal use	OLAP systems, Data Marts	OLTP systems

Table 1. A comparison of star and snowflake schema attributes.

Normalization reduces redundancy

Both star and snowflake schemas benefit from the application of normalization. “Normalization reduces redundancy” is an idiom that points to a key advantage leveraged by both schemas.

Normalizing a table means to create, for each dimension:

1. A surrogate key to replace the natural key, that is, the unique values of the given column, and
2. A lookup table to store the surrogate and natural key pairs.

Each surrogate key’s values are repeated exactly as many times within the normalized table as the natural key was before moving the natural key to its new lookup table. Thus, you did nothing to reduce the redundancy of the original table.

However, dimensions typically contain groups of items that appear frequently, such as a “city name” or “product category”. Since you only need one instance from each group to build your lookup table, your lookup table will have many fewer rows than your fact table. If there are child dimensions involved, then the lookup table may still have some redundancy in the child dimension columns. In

other words, if you have a hierarchical dimension, such as “Country”, “State”, and “City”, you can repeat the process on each level to further reduce the redundancy.

Notice that further normalizing your hierarchical dimensions has no effect on the size or content of your fact table - star and snowflake schema data models share identical fact tables.

Normalization reduces data size

When you normalize a table, you typically reduce its data size, because in the process you likely replace expensive data types, such as strings, with much smaller integer types. But to preserve the information content, you also need to create a new lookup table that contains the original objects.

The question is, does this new table use less storage than the savings you just gained in the normalized table?

For small data, this question is probably not worth considering, but for big data, or just data that is growing rapidly, the answer is yes, it is inevitable. Indeed, your fact table will grow much more quickly than your dimension tables, so normalizing your fact table, at least to the minimum degree of a star schema is likely warranted. Now the question is about which is better – star or snowflake?

Comparing benefits: snowflake vs. star data warehouses

The snowflake, being completely normalized, offers the least redundancy and the smallest storage footprint. If the data ever changes, this minimal redundancy means the snowflaked data needs to be changed in fewer places than would be required for a star schema. In other words, writes are faster, and changes are easier to implement.

However, due to the additional joins required in querying the data, the snowflake design can have an adverse impact on read speeds. By denormalizing to a star schema, you can boost your query efficiency.

You can also choose a middle path in designing your data warehouse. You could opt for a partially normalized schema. You could deploy a snowflake schema as your basis and create views or even materialized views of denormalized data. You could for example simulate a star schema on top of a snowflake schema. At the cost of some additional complexity, you can select from the best of both worlds to craft an optimal solution to meet your requirements.

Practical differences

Most queries you apply to the dataset, regardless of your schema choice, go through the fact table. Your fact table serves as a portal to your dimension tables.

The main practical difference between star and snowflake schema from the perspective of an analyst has to do with querying the data. You need more joins for a snowflake schema to gain access to the deeper levels of the hierarchical dimensions, which can reduce query performance over a star schema. Thus, data analysts and data scientists tend to prefer the simpler star schema.

Snowflake schemas are generally good for designing data warehouses and in particular, transaction processing systems, while star schemas are better for serving data marts, or data warehouses that have simple fact-dimension relationships. For example, suppose you have point-of-sale records accumulating in an Online Transaction Processing System (OLTP) which are copied as a daily batch ETL process to one or more Online Analytics Processing (OLAP) systems where subsequent analysis of large volumes of historical data is carried out. The OLTP source might use a snowflake schema to optimize performance for frequent writes, while the OLAP system uses a star schema to optimize for frequent reads. The ETL pipeline that moves the data between systems includes a denormalization step which collapses each hierarchy of dimension tables into a unified parent dimension table.

Too much of a good thing?

There is always a tradeoff between storage and compute that should factor into your data warehouse design choices. For example, do your end-users or applications need to have precomputed, stored dimensions such as ‘day of

week', 'month of year', or 'quarter' of the year? Columns or tables which are rarely required are occupying otherwise usable disk space. It might be better to compute such dimensions within your SQL statements only when they are needed. For example, given a star schema with a date dimension table, you could apply the SQL 'MONTH' function as MONTH(dim_date.date_column) on demand instead of joining the precomputed month column from the MONTH table in a snowflake schema.

Scenario

Suppose you are handed a small sample of data from a very large dataset in the form of a table by your client who would like you to take a look at the data and consider potential schemas for a data warehouse based on the sample. Putting aside gathering specific requirements for the moment, you start by exploring the table and find that there are exactly two types of columns in the dataset - facts and dimensions. There are no foreign keys although there is an index. You think of this table as being a completely denormalized, or flattened dataset.

You also notice that amongst the dimensions are columns with relatively expensive data types in terms of storage size, such as strings for names of people and places.

At this stage you already know you could equally well apply either a star or snowflake schema to the dataset, thereby normalizing to the degree you wish. Whether you choose star or snowflake, the total data size of the central fact table will be dramatically reduced. This is because instead of using dimensions directly in the main fact table, you use surrogate keys, which are typically integers; and you move the natural dimensions to their own tables or hierarchy of tables which are referenced by the surrogate keys. Even a 32-bit integer is small compared to say a 10-character string (8 X 10 = 80 bits).

Now it's a matter of gathering requirements and finding some optimal normalization scheme for your schema.

▼ Staging Areas for Data Warehouses

Data warehouse staging areas

A staging area is:

- Intermediate storage area that is used for ETL processing. Thus, staging areas act as a bridge between data sources and the target data warehouses, data marts, or other data repos.
- They are often transient, meaning that they are erased after successfully running ETL workflows. However, many architectures hold data for archival or troubleshooting purposes.
- They are also useful for monitoring and optimizing your ETL workflows.

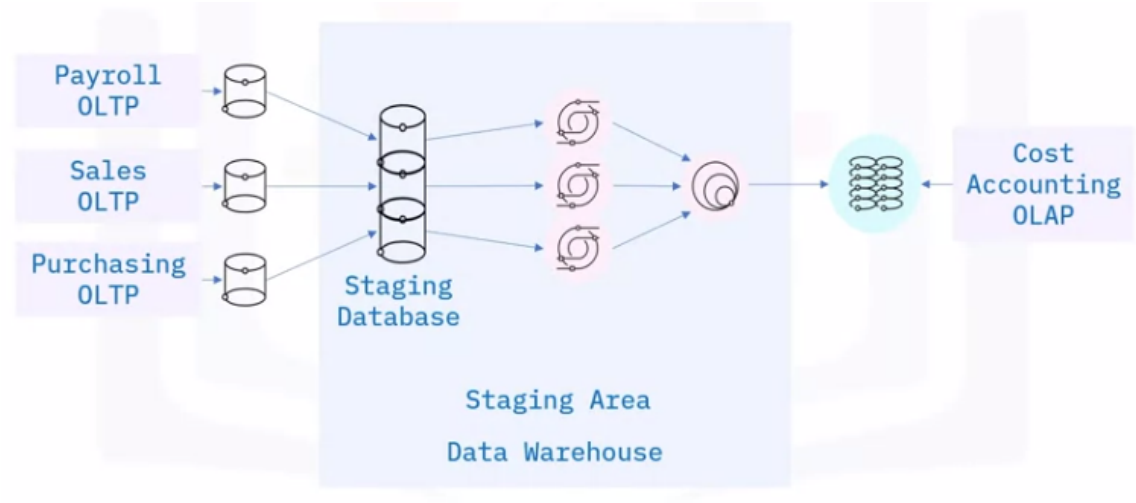
Staging areas can be implemented in many ways:

- Simple flat files, such as csv files, stored in a directory, and managed using tools such as Bash or Python.
- Set of SQL tables in a relational database such as Db2
- Self-contained database instance within a data warehousing or business intelligence platform such as Cognos Analytics.

DW staging area example

Imagine the enterprise would like to create a dedicated “Cost Accounting” Online Analytical Processing system. The required data is managed in separate Online Transaction Processing Systems within the enterprise, from the Payroll, Sales, and Purchasing departments. From these siloed systems, the data is extracted to individual Staging Tables, which are created in the Staging Database. Data from these tables is then transformed in the Staging Area using SQL to conform it to

the requirements of the Cost Accounting system. The conformed tables can now be integrated, or joined, into a single table. The final phase is the loading phase, where the data is loaded into the target cost-accounting system.



Functions of a staging area

- **Integration:** Indeed, one of the primary functions performed by a staging area is consolidation of data from multiple source systems.
- **Change detection:** Staging areas can be set up to manage extraction of new and modified data as needed.
- **Scheduling:** Individual tasks within an ETL workflow can be scheduled to run in a specific sequence, concurrently, and at certain times.
- **Data cleansing and validation.** You can handle missing values and duplicated records.
- **Aggregating data:** You can use the staging area to summarize data. For example, daily sales data can be aggregated into weekly, monthly, or annual averages, prior to loading into a reporting system.
- **Normalizing data:** To enforce consistency of data types, or names of categories such as country and state codes in place of mixed naming conventions such as “Mont,” “MA,” or “Montana.”

Why use a staging area?

- A staging area is a **separate location**, where data from source systems is extracted to.
- The extraction step therefore **decouples operations** such as validation, cleansing and other processes from the source environment. This helps to **minimize any risk** of corrupting source-data systems.
- **Simplifies ETL workflow** construction, operation, and maintenance.
- If any of the extracted data becomes corrupted somehow, you can **easily recover**.

▼ Verify Data Quality

What is data quality verification?

Data verification includes checking data for:

- Accuracy
- Completeness
- Consistency
- Currency

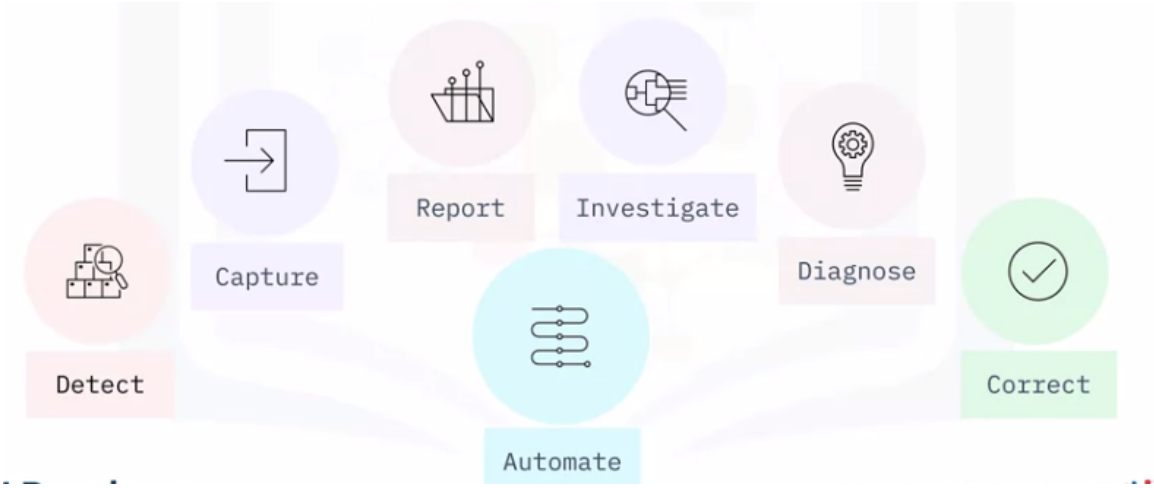
Data verification is about managing data quality and enhancing its reliability. High-quality data enables successful integration of related data and its complex relationships. Data verification also provides you with a complete and connected view of your organization, data that is ready for advanced analysis, statistical modeling and machine learning, and ultimately, more confidence in your insights and decision-making.

Data quality concerns

- Accuracy
    - Source and destination records match
    - Duplicated records
    - Typos
        - Out-of-range values

        - Spelling errors
    - Mass misalignment
        - Misinterpretation of a comma as a separator
- Completeness
    - Locating missing values
        - Void or null fields
        - Placeholders like “999” or “-1”
    - Locationg missing data records due to system failures
- Consistency
    - Non-conformance to standard terms
    - Date formatting
        - YMD and MDY
    - Inconsistent data entry
        - “Mr. Josn Doe” and “John Doe”
    - Inconsistent units
        - Metric and imperial
        - Dollars and thousands of dollars
- Currency
    - Avoiding outdated information
        - Update addresses: Check against a ‘change of address’ database
        - Manage name changes

Managing data quality



Managing data quality - an action plan (example)

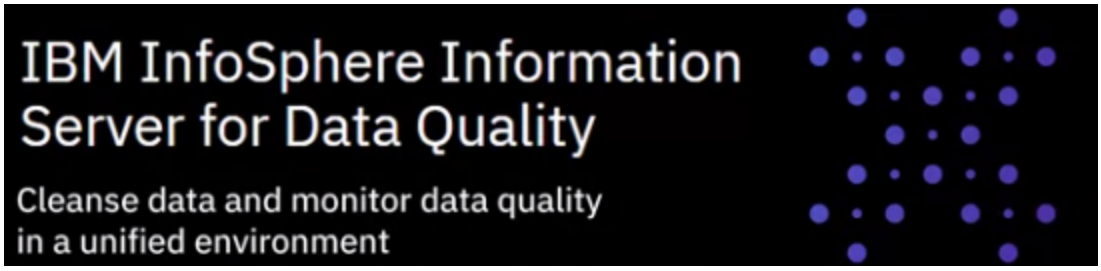
You need to validate the quality of data in the staging area before loading the data into a data warehouse for analytics. You determine that data from certain data sources consistently has data quality issues including: Missing data, Duplicate values, Out-of-range values, and Invalid values. Here's how an organization might manage and resolve these issues:

1. Write SQL queries to detect these issues and test for them
2. Address some of the quality issues that you've repeatedly identified by creating rules for treating them, such as removing rows that have out-of-range values.
3. Create a script that runs queries to detect data quality issues that happen during the nightly loads to the data warehouse. This script applies corrective measures and transformations for some of these known issues.
4. Create a second script that automates the script you created in step 3. After the data is extracted from the various data sources, this script automatically runs the prior script's SQL data validation queries every night in the staging area.
5. The script you created in step 3 generates a report of any remaining issues that could not be automatically resolved. The administrator can review this report and address the unresolved issues.

Data quality solutions

- IBM InfoSphere Server for Data Quality
- Informatica Data Quality
- SAP Data Quality Management
- SAS Data Quality
- Talend Open Studio for Data Quality
- Precisely Spectrum Quality
- Microsoft Data Quality Services
- Oracle Enterprise Data Quality
- OpenRefine (open-source)

IBM InfoSphere



Enables you:

- Continuously monitor data quality
- Clean data on an ongoing basis
- Turn your data into trusted information

End-to-end data quality tools:

- Undertand your data and its relationships
- Monitor and analyze data quality continuously
- Clean, standardize, and match data
- Maintain data lineage

▼ Populating a Data Warehouse

Loading frequency

Populating the warehouse is an ongoing process:

- Initial load + periodic incremental loads
- Full refreshes due to schema changes or catastrophic failure are rare
- Fact tables need more frequent updating than dimension tables
- City and store lists change slowly, unlike sales transactions

Typical ways of loading data

- Db2 have a Load utility that is faster than inserting a row at a time
- Loading your Warehouse can also be a part of your ETL data pipeline that is automated using tools like Apache Airflow and Apache Kafka
- You can also write your own scripts, combining lower-level tools like Bash, Python, and SQL, to build your data pipeline and schedule it with cron
- InfoSphere DataStage allows you to compile and run jobs to load your data

Populating your data warehouse

Preparation:

- Your schema has been modeled
- Your data has been staged
- You have verified the data quality

Setup and initial load:

- Instantiate the data warehouse and its schema
- Create production tables
- Establish relationships between tables
- Load your transformed and cleaned data

Onoging loads:

- You can automate subsequent incremental loads using a script as part of your ETL data pipeline

- You will also need to include some logic to determine what data is new or updated in your staging area

Change detection

Normally, you detect changes in the source system itself. Many relational database management systems have mechanisms for identifying any new, changed, or deleted records since a given date. You might also have access to timestamps that identify both when the data was first written and the date it might have been modified. Some systems might be less accommodating and you might need to load the entire source to your ETL pipeline for subsequent brute-force comparison to the target, which is fine if the source data isn't too large.

Periodic maintenance

Data warehouses need periodic maintenance, usually monthly or yearly, to archive data that is not likely to be used. You can script both the deletion of older data and its archiving to slower, less costly storage.

Create a dimension table

```
CREATE TABLE sales.DimSalesPerson (
  SalesPersonID SERIAL primary key,
  SalesPersonAltID varchar(10) not null,
  SalesPersonName varchar(50)
);
```

Populate the dimension table

```
INSERT INTO sales.DimSalesPerson(SalesPersonAltID,
                                 SalesPersonName)
values
( 617, 'Gocart Joe'),
( 642, 'Jake Salesbouroughs'),
( 680, 'Cadillac Jack');
```

View the salesperson table

```
SELECT * FROM sales.DimSalesPerson LIMIT 5;
```

Create the sales fact table

```
CREATE TABLE sales.FactAutoSales(
  TransactionID bigserial primary key,
  SalesID int not ull,
  SalesDateKey int,
  AutoClassID int not null,
  SalesPersonID int not null,
  Amount money
);
```

Set up table relationships

```
ALTER TABLE sales.FactAutoSales
ADD CONSTRAINT
KV_AutoClassID FOREIGN KEY (AutoClassID)
REFERENCES
sales.DimAutoCategory(AutoClassID);
```

Populate the sales fact table

```
INSERT INTO sales.FactAutoSales(
  SalesID,
  Amount,
  SalesPersonID,
  AutoClassID,
  SalesDateKey
) values

(1629, 42000.00, 2, 1, 4),
(1630, 17680.00, 1, 2, 4),
(1631, 37100.00, 2, 2, 5),
```

View the sales fact table

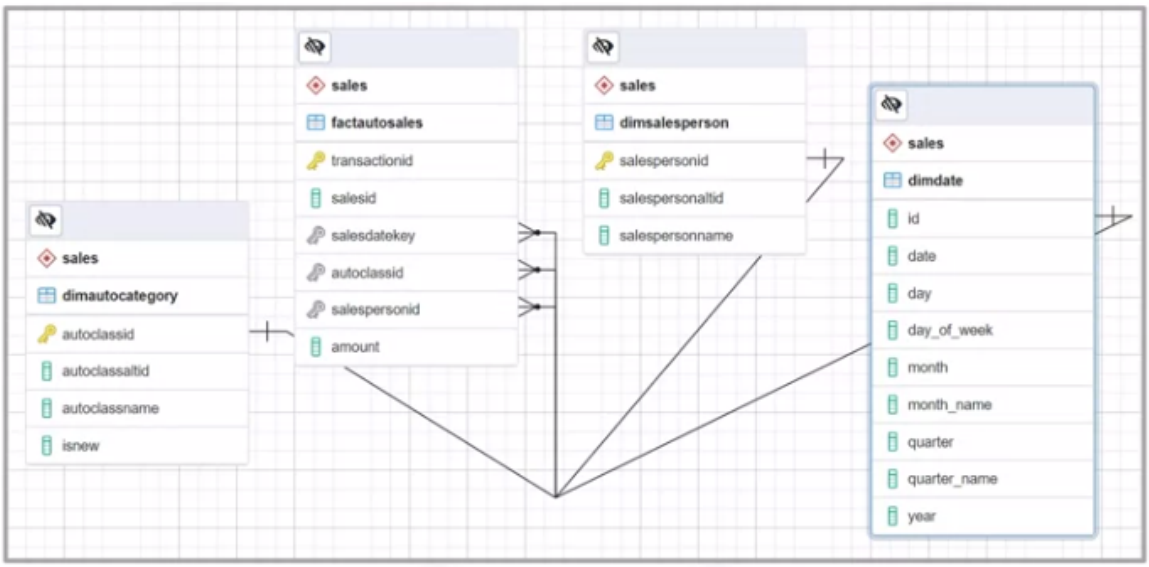
```
SELECT * sales.FactAutoSales LIMIT 5;
```

▼ Querying the Data

CUBE, ROLLUP, materialized views

- CUBE and ROLLUP provides summary reports
- Easier to implement than the multiple SQL queries that are otherwise required
- Materialized views conveniently enable you to create a stored table you so can refresh on a schedule or on-demand. When the a view is complex, requested frequently, or is run on large data sets, consider materializing the view to help reduce the load on the database
- Because the data is precomputed, querying materialized views can be much faster than querying the underlying tables
- Combining cubes or rollups with materialized views can enhance performance. You can even follow up by materializing the cube or rollup

ShinyAutoSales - sales ERD



View the sales fact table

```
sasDW=# SELECT * FROM sales.factautosales LIMIT 10;
 transactionid | salesid | salesdatekey | autoclassid | salespersonid |   amount
---------------+---------+--------------+-------------+---------------+------------
            52 |    1629 |            4 |           1 |             2 | $42,000.00
            53 |    1630 |            4 |           2 |             1 | $17,680.00
            54 |    1631 |            5 |           2 |             2 | $37,100.00
            55 |    1632 |            5 |           3 |             3 | $26,500.00
            56 |    1633 |            5 |           4 |             4 |  $8,200.00
            57 |    1634 |            5 |           5 |             2 | $42,099.00
            58 |    1635 |            6 |           6 |             5 | $12,099.00
            59 |    1636 |            6 |           5 |             6 | $51,999.00
            60 |    1637 |            7 |           2 |             1 | $42,099.00
            61 |    1638 |            7 |           3 |             1 | $32,099.00
(10 rows)
```

View the auto category table

```
sasDW=# SELECT * FROM sales.dimautocategory LIMIT 10;
 autoclassid | autoclassaltid | autoclassname | isnew
-------------+----------------+---------------+-------
           1 | 30             | 4 Door Sedan  | t
           2 | 70             | Truck         | t
           3 | 60             | Midsize SUV   | t
           4 | 71             | Truck         | f
           5 | 40             | Compact SUV   | t
           6 | 61             | Midsize SUV   | f
           7 | 41             | Compact SUV   | f
           8 | 31             | 4 Door Sedan  | f
(8 rows)
```

View the salesperson table

```
sasDW=# SELECT * FROM sales.dimsalesperson LIMIT 10;
 salespersonid | salespersonaltid |    salespersonname
---------------+------------------+-----------------------
             1 | 617              | Gocart Joe
             2 | 642              | Jake Salesbouroughs
             3 | 680              | Cadillac Jack
             4 | 707              | Jane Honda
             5 | 720              | Kayla Kaycar
             6 | 607              | William Jeepman
             7 | 609              | Happy Dollarmaker
             8 | 711              | Sally Caraway
(8 rows)
```

View the date table

```
sasDW=# SELECT * FROM sales.dimdate LIMIT 8;
 id |    date    | day | day_of_week | month | month_name | quarter | quarter_name | year
----+------------+-----+-------------+-------+------------+---------+--------------+------
  1 | 2021-01-01 |   1 | Fri         |     1 | Jan        |       1 | Q1           | 2021
  2 | 2021-01-02 |   2 | Sat         |     1 | Jan        |       1 | Q1           | 2021
  3 | 2021-01-03 |   3 | Sun         |     1 | Jan        |       1 | Q1           | 2021
  4 | 2021-01-04 |   4 | Mon         |     1 | Jan        |       1 | Q1           | 2021
  5 | 2021-01-05 |   5 | Tue         |     1 | Jan        |       1 | Q1           | 2021
  6 | 2021-01-06 |   6 | Wed         |     1 | Jan        |       1 | Q1           | 2021
  7 | 2021-01-07 |   7 | Thu         |     1 | Jan        |       1 | Q1           | 2021
  8 | 2021-01-08 |   8 | Fri         |     1 | Jan        |       1 | Q1           | 2021
(8 rows)
```

Denormalized view

```
sasDW-# SELECT
sasDW-#     D.date,
sasDW-#     C.autoclassname,
sasDW-#     C.isnew,
sasDW-#     SP.salespersonname,
sasDW-#     F.amount
sasDW-#
sasDW-# FROM sales.factautosales F
sasDW-#
sasDW-# INNER JOIN sales.dimdate D          ON (F.salesdatekey = D.id)
sasDW-# INNER JOIN sales.dimautocategory C  ON (F.autoclassid = C.autoclassid)
sasDW-# INNER JOIN sales.dimsalesperson SP  ON (F.salespersonid = SP.salespersonid)
sasDW-# ;
SELECT 17
```

Denormalized, materialized view

```
sasDW=# CREATE MATERIALIZED VIEW  DNsales AS
sasDW-#
sasDW-# SELECT
sasDW-#     D.date,
sasDW-#     C.autoclassname,
sasDW-#     C.isnew,
sasDW-#     SP.salespersonname,
sasDW-#     F.amount
sasDW-#
sasDW-# FROM sales.factautosales F
sasDW-#
sasDW-# INNER JOIN sales.dimdate D          ON (F.salesdatekey = D.id)
sasDW-# INNER JOIN sales.dimautocategory C  ON (F.autoclassid = C.autoclassid)
sasDW-# INNER JOIN sales.dimsalesperson SP  ON (F.salespersonid = SP.salespersonid)
sasDW-# ;
SELECT 17
```

DNsales materialized view

```
sasDW=# SELECT * FROM DNsales LIMIT 10;
    date    | autoclassname | isnew |   salespersonname    |   amount
------------+---------------+-------+----------------------+------------
 2021-01-04 | 4 Door Sedan  | t     | Jake Salesbouroughs  | $42,000.00
 2021-01-04 | Truck         | t     | Gocart Joe           | $17,680.00
 2021-01-05 | Truck         | t     | Jake Salesbouroughs  | $37,100.00
 2021-01-05 | Midsize SUV   | t     | Cadillac Jack        | $26,500.00
 2021-01-05 | Truck         | f     | Jane Honda           |  $8,200.00
 2021-01-05 | Compact SUV   | t     | Jake Salesbouroughs  | $42,099.00
 2021-01-06 | Midsize SUV   | f     | Kayla Kaycar         | $12,099.00
 2021-01-06 | Compact SUV   | t     | William Jeepman      | $51,999.00
 2021-01-07 | Truck         | t     | Gocart Joe           | $42,099.00
 2021-01-07 | Midsize SUV   | t     | Gocart Joe           | $32,099.00
(10 rows)
```

Applying CUBE to your materialized view

```
sasDW=# SELECT
sasDW-#     autoclassname,
sasDW-#     salespersonname,
sasDW-#     SUM(amount)
sasDW-# FROM
sasDW-#     DNsales
sasDW-# WHERE
sasDW-#     isNew=True
sasDW-# GROUP BY
sasDW-#     CUBE (
sasDW(#         autoclassname,
sasDW(#         salespersonname
sasDW(#     );
```

```
 autoclassname |   salespersonname    |     sum
---------------+----------------------+-------------
               |                      | $366,076.00
 4 Door Sedan  | Jake Salesbouroughs  |  $42,000.00
 Truck         | Gocart Joe           |  $59,779.00
 Truck         | Jake Salesbouroughs  |  $37,100.00
 Compact SUV   | Jake Salesbouroughs  |  $42,099.00
 Midsize SUV   | Gocart Joe           |  $32,099.00
 Compact SUV   | William Jeepman      |  $51,999.00
 Compact SUV   | Happy Dollarmaker    |  $74,500.00
 Midsize SUV   | Cadillac Jack        |  $26,500.00
 Compact SUV   |                      | $168,598.00
 Truck         |                      |  $96,879.00
 Midsize SUV   |                      |  $58,599.00
 4 Door Sedan  |                      |  $42,000.00
               | Happy Dollarmaker    |  $74,500.00
               | Jake Salesbouroughs  | $121,199.00
               | Cadillac Jack        |  $26,500.00
               | William Jeepman      |  $51,999.00
               | Gocart Joe           |  $91,878.00
(18 rows)
```

Applying ROLLUP to your materialized view

```
sasDW=# SELECT
sasDW-#     autoclassname,
sasDW-#     salespersonname,
sasDW-#     SUM(amount)
sasDW-# FROM
sasDW-#     DNsales
sasDW-# WHERE
sasDW-#     isNew=True
sasDW-# GROUP BY
sasDW-#     ROLLUP (
sasDW(#         autoclassnam
sasDW(#         salespersonn
sasDW(#     );
```

autoclassname	salespersonname	sum
		$366,076.00
4 Door Sedan	Jake Salesbouroughs	$42,000.00
Truck	Gocart Joe	$59,779.00
Truck	Jake Salesbouroughs	$37,100.00
Compact SUV	Jake Salesbouroughs	$42,099.00
Midsize SUV	Gocart Joe	$32,099.00
Compact SUV	William Jeepman	$51,999.00
Compact SUV	Happy Dollarmaker	$74,500.00
Midsize SUV	Cadillac Jack	$26,500.00
Compact SUV		$168,598.00
Truck		$96,879.00
Midsize SUV		$58,599.00
4 Door Sedan		$42,000.00

(13 rows)

Hands-on Lab: Working with Facts and Dimension Tables
Hands-on Lab: Settign up a Staging Area
Hands-on Lab: Verifying Data Quality for a Data Warehouse
Hands-on Lab: Populating a Data Warehouse
(Optional) Hands-on Lab: Populating a Data Warehouse
Hands-on Lab: Querying the Data Warehouse (Cubes, Rollups, Grouping Sets and Materialized Views)
(Optional) Hands-on Lab: Querying the Data Warehouse (Cubes, Rollups, Grouping Sets and Materialized Views)