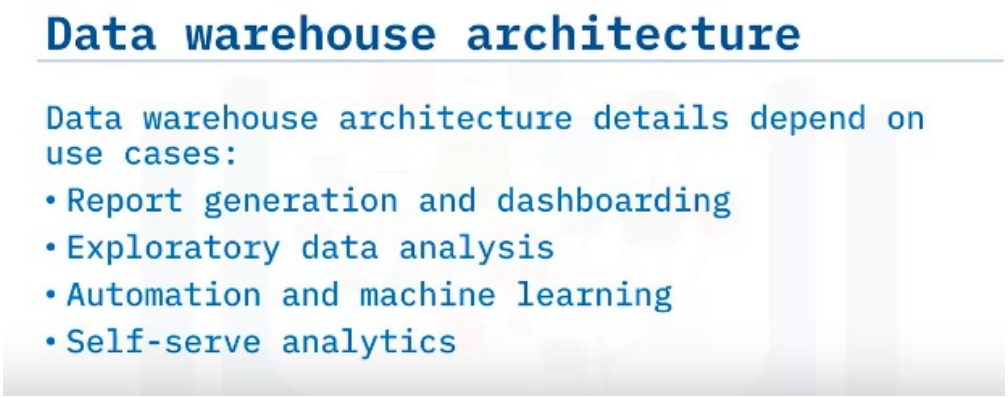
Designing, Modeling and Implementing Data Warehouses

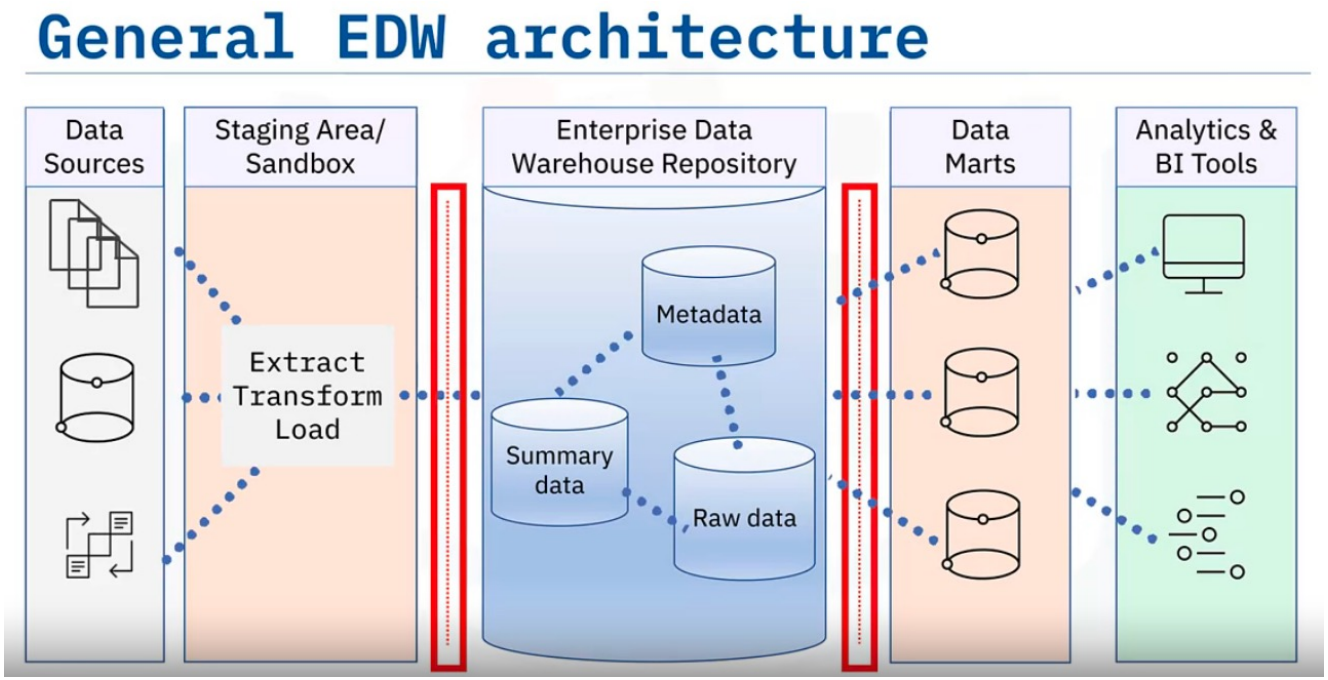
C9-Week-2

- Describe a general data warehousing architecture and its component layers and distinguish between general and reference enterprise data warehouse architecture.
- Describe the relationship of a data cube to a star schema and describe the actions of slice, dice, drill up or down, roll up, and pivot as they relate to data cubes.
- Describe materialized views and recall two use cases for them.
- Summarize how to aggregate data over multiple dimensions in SQL.
- Define facts, a fact table, dimensions, and dimension tables, and describe the use of facts and dimensions.
- *Design fact and dimension tables.
- Create a star schema using fact and dimension tables.
- *Desacribe star-schema modeling facts and dimensions and explain how to use normalization to create the snowflake schema as an extension of a star schema.
- Summarize key considerations for using star and snowflake schemes in data warehousing.
- Define a data warehouse staging area and describe why staging areas are used as a first step for integrating data sources.
- *Build a data warehouse staging area.
- Define data verification, explain why organizations verify data, and outline a process for handling bad data.
- *Verify data quality for a data warehouse.
- List the main steps for populating a data warehouse, describe methods for change detection and incremental loading, and manually create and populate tables for a sales star schema.
- *Populate a data warehouse.
- Interpret an entity-relationship diagram for a star schema.
- Write queries using grouping sets, rollups, cubes.
- Create a materialized query table (MQT).

Overview of Data Warehouse Architectures

- List use cases that drive data warehouse design considerations.
- Describe a general data warehousing architecture and list its component layers.
- Distinguish between general and reference enterprise data warehouse architecture and
- Describe reference architectures for two enterprise data warehouse platforms.

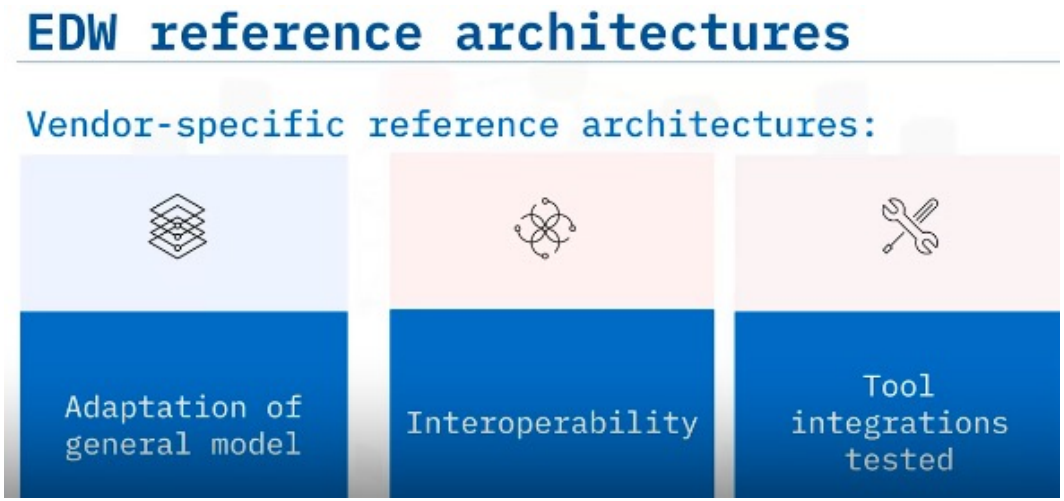




- A general architectural model for an Enterprise Data Warehouse, or EDW, platform, which companies can adapt for their analytics requirements.

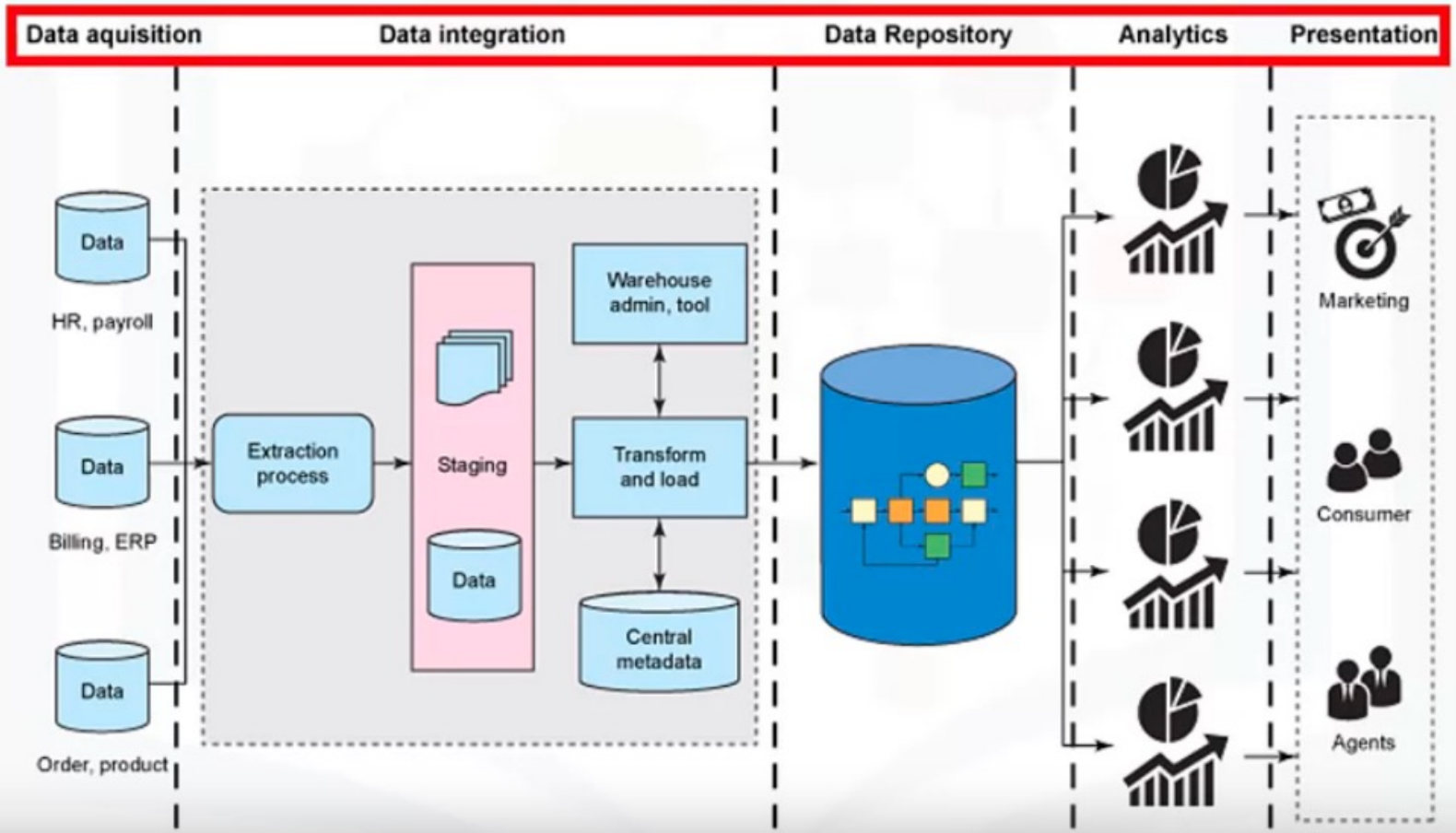
In this architecture, you can have various layers or components, including:

1. **Data sources**, such as flat files, databases, and existing operational systems,
2. **an ETL layer** for extracting, transforming, and loading data,
3. optional **staging and sandbox areas** for holding data and developing workflows,
4. an **enterprise data warehouse repository**,
5. sometimes, **data marts,** which are known as a "hub and spoke" architecture when multiple data marts are involved,
6. and an **analytics layer and BI tools**.
7. Data warehouses also enforce security for incoming data and data passing through to further stages and users throughout the network.



- 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

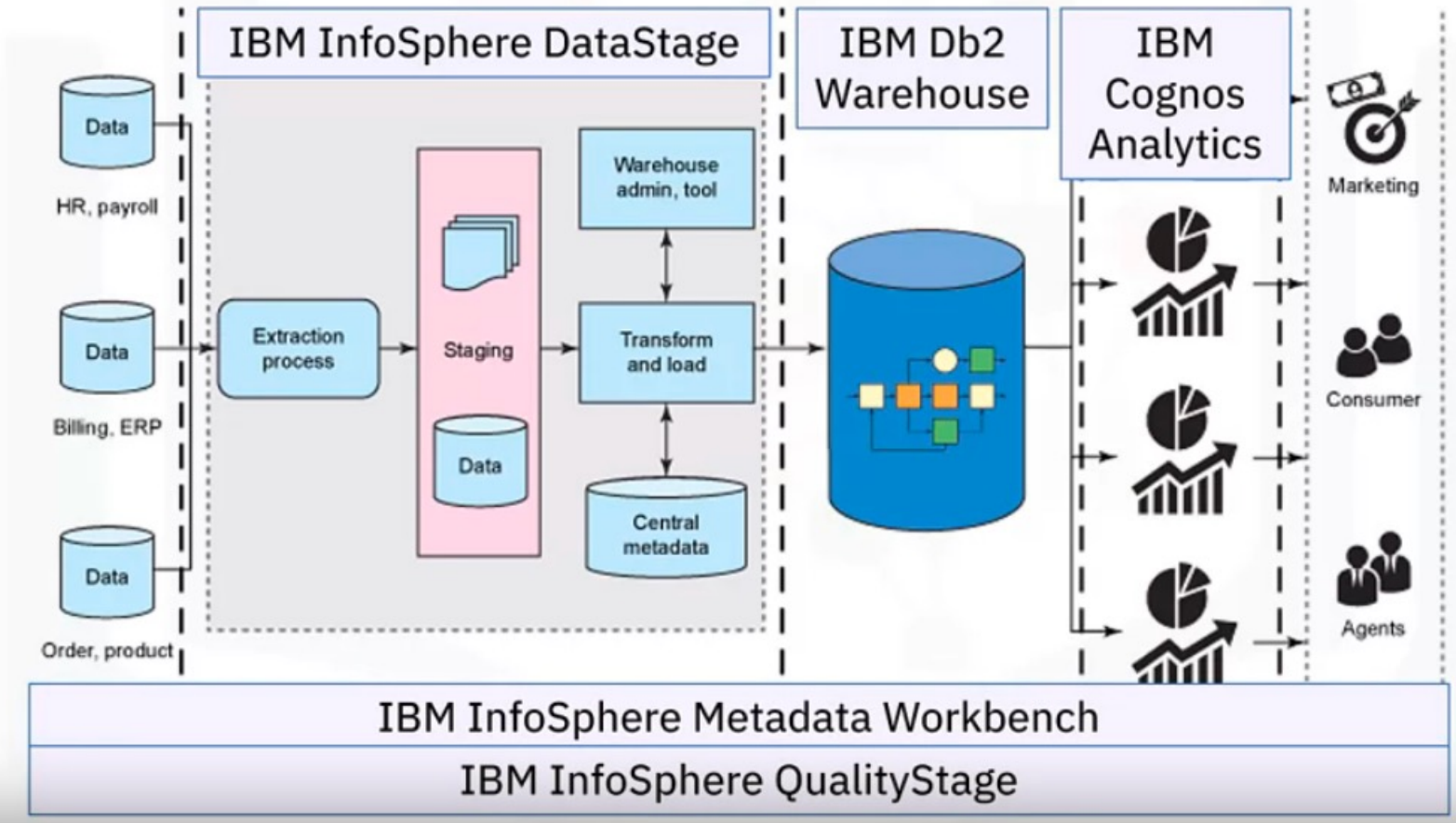


IBM-specific reference data warehouse architecture

Each layer of the architecture performs a specific function:

- 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 (ETL) 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.
- The final **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



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.

Summary

In this video, you learned that:

- A general architectural model includes data sources, ETL pipelines, optional staging and sandbox areas, EDW repository, optional data marts and analytics/BI tools
- Companies can adapt the general EDW architecture to suit their analytics requirements
- Vendors offer reference EDW architectures that are tested to ensure interoperability
- An IBM reference EDW solution combines InfoSphere with Db2 Warehouse and Cognos Analytics

Cubes, Rollups, and Materialized Views and Tables

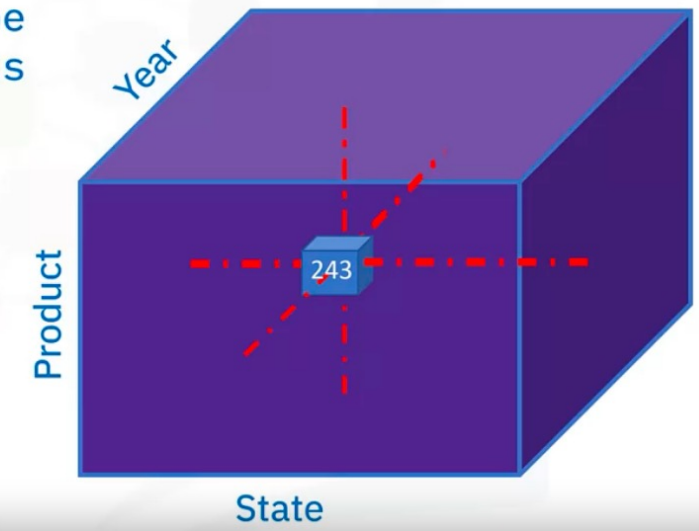
- Relate what a data cube is in terms of star schema.
- Discuss the terms slice, dice, drill up or down, roll up, and pivot in terms of data cubes.
- Describe what a materialized view is. And, recall two use cases for materialized views.

What is a data cube?

- Example: Sales OLAP cube
- Coordinates = dimensions
- Cells = facts

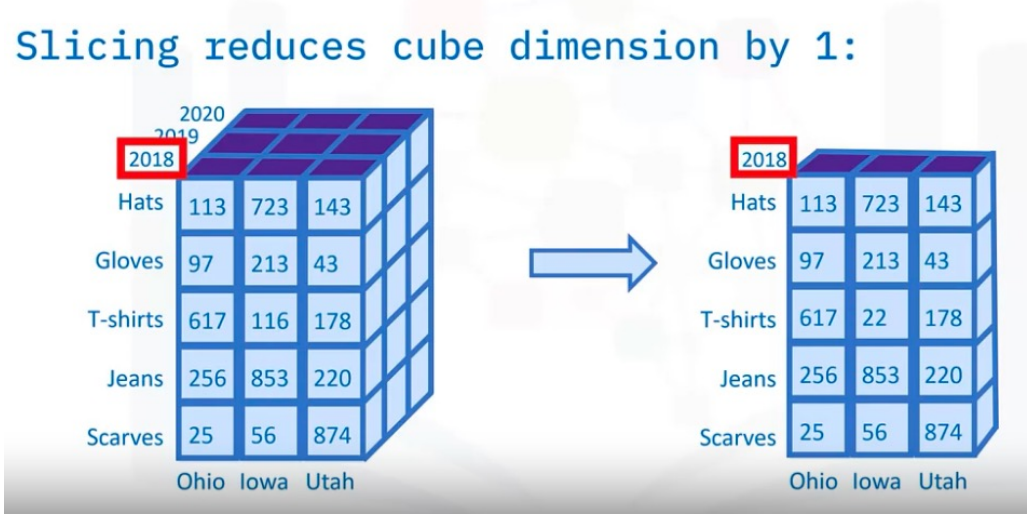
Cube operations:

- Slicing
- Dicing
- Drilling up and down
- Pivoting
- Rolling up



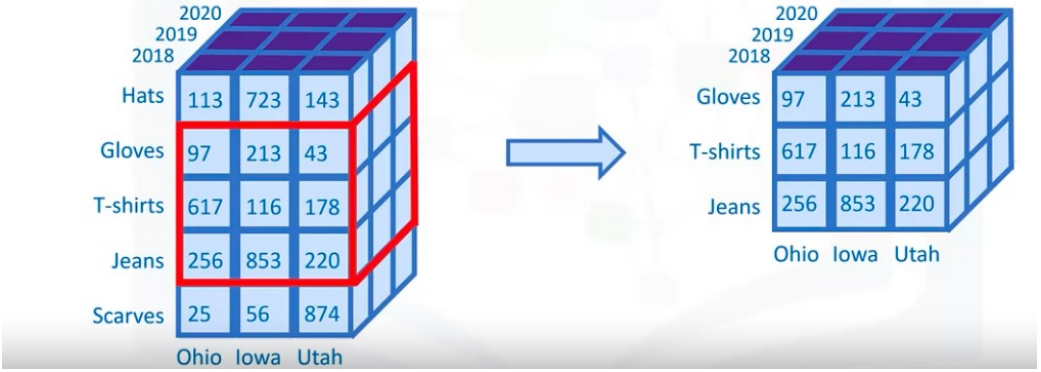
- Here is a cube generated from an **imaginary star schema for a Sales OLAP** (online analytical processing system).
- The **coordinates** of the cube are defined by a **set of dimensions**, which are selected from the star schema.
- In this illustration, we are only showing three dimensions, but data cubes can **have many dimensions.**
- We have the Product categories corresponding to the items sold, the State or Province the items were sold from, and the Year these products were sold in.
- 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.
- There are many operations you can perform on data cubes, such as slicing, dicing, drilling xup and down, pivoting, and rolling up.

Slicing data cubes



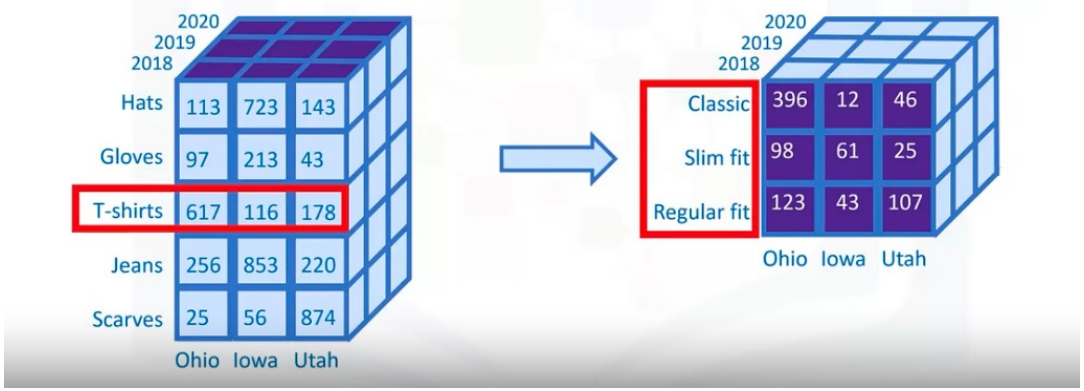
- Slicing a data cube involves **selecting a single member from a dimension, w**hich yields a data cube that has one dimension less than the original.
- For example, you can slice this sales cube by selecting only the year 2018 from the year dimension, allowing you to analyze sales totals for all sales states and all products for the year 2018.

Dicing data cubes



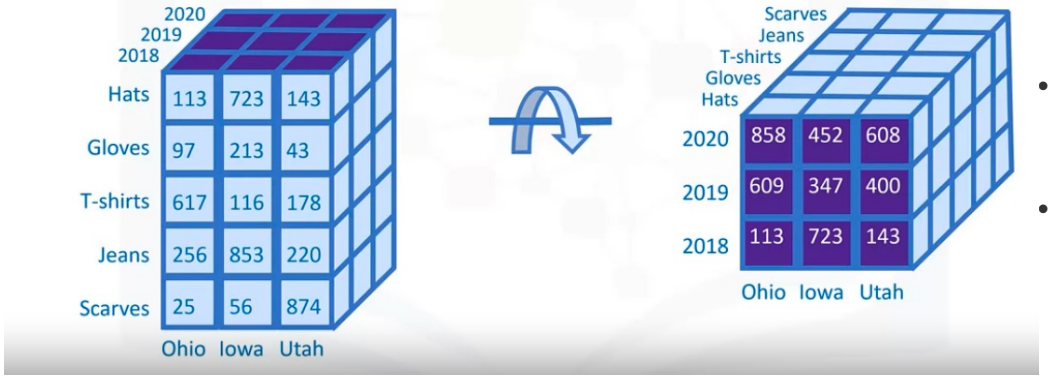
- dicing a cube involves **selecting a subset of values from a dimension, effectively shrinking it.**
- For example, you can dice this sales cube by selecting only “Gloves”, “T-shirts”, and “Jeans” from the Product-Type dimension, allowing you to restrict your view to just those product types.

Drilling up or down in data cubes



- In **Snowflake** schema, you will find hierarchies, or subcategories within some of your dimensions that you can drill into.
- Thus, for example, you can **“drill down”** into a particular member of **the “Product category”** dimension, such as “T-shirts,” resulting in this view, which may include more specific “product groups” such as “Classic,” “Slim fit,” and “Regular fit.”
- Drilling up is just the reverse process, which would take you back to the original data cube

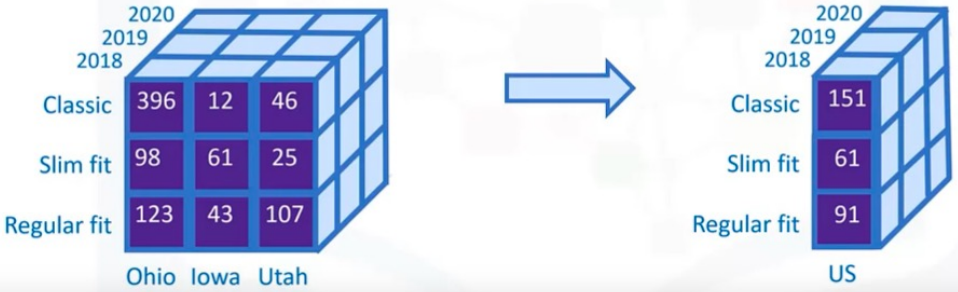
Pivoting data cubes



- Pivoting data cubes is straightforward. It involves a **rotation of the data cube.**
- In this case, the year and product dimensions have been interchanged, while the State dimension has been fixed "as is."
- 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

- Roll up = summarize a dimension
- Aggregate using COUNT, MIN, MAX, SUM, AVERAGE



- Rolling up means **summarizing along a dimension**.
- You can roll up a dimension by applying aggregations, such as **COUNT, MIN, MAX, SUM, and AVERAGE.**
- For example, you could calculate the average selling price of Classic, Slim fit, and Regular fit T-shirts by summing horizontally over the three US states and dividing by three.

Materialized views

- A "snapshot" containing results of a query
- Used to replicate data in a staging database, or
- Precompute expensive queries for a data warehouse
- Automatically keep query results synced to database
- Safely work without affecting source database

- 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 for different refresh options:

- Never - populated on creation only
- Upon request - manually or scheduled
- Immediately – automatically, after every statement

- 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 a materialized view in Oracle:

```
CREATE MATERIALIZED VIEW MY_MAT_VIEW
REFRESH FAST START WITH SYSDATE
NEXT SYSDATE + 1
AS SELECT * FROM <my_table_name>;
```

- Here is how you might create a materialized view in Oracle using SQL statements.
- Start by creating and naming a “materialized view” object called “My underscore Mat underscore View”, Specify the refresh type as fast, which means “incrementally refresh the data”.
- Specify today as the start date, and Refresh the view every day.
- The final statement selects all data from my underscore table underscore name.

Materialized view in PostgreSQL

Create a materialized view in PostgreSQL:

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

- Here is how you might create a materialized view in PostgreSQL to replicate a table.
- Start by creating a “materialized view” object called “My underscore Mat underscore View”, Specify some parameters, Specify the source tablespace, say “tablespace underscore name”, and Select all rows and columns from “table underscore name.”
- In PostgreSQL you can only refresh materialized views manually, using the “refresh material view” command.

Materialized view in Db2

In Db2, materialized views are called MQTs

```
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
```

- In Db2, materialized views are called **MQTs**, which stands for "**materialized query tables."**
- Here's an example, from IBM's online documentation, of creating a system-maintained "immediate refresh" MQT.
- 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.

Materialized view in Db2

```
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
```

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

Summary

In this video, you learned that:

- A data cube represents a schema’s dimensions as coordinates plus a fact for its values
- Many operations can be applied to data cubes, such as drilling down into hierarchical dimensions, slicing, dicing, and rolling up
- Materialized views can be used to replicate data or to precompute expensive queries
- Some tools can automatically refresh your material views for you

Facts and Dimensional Modeling

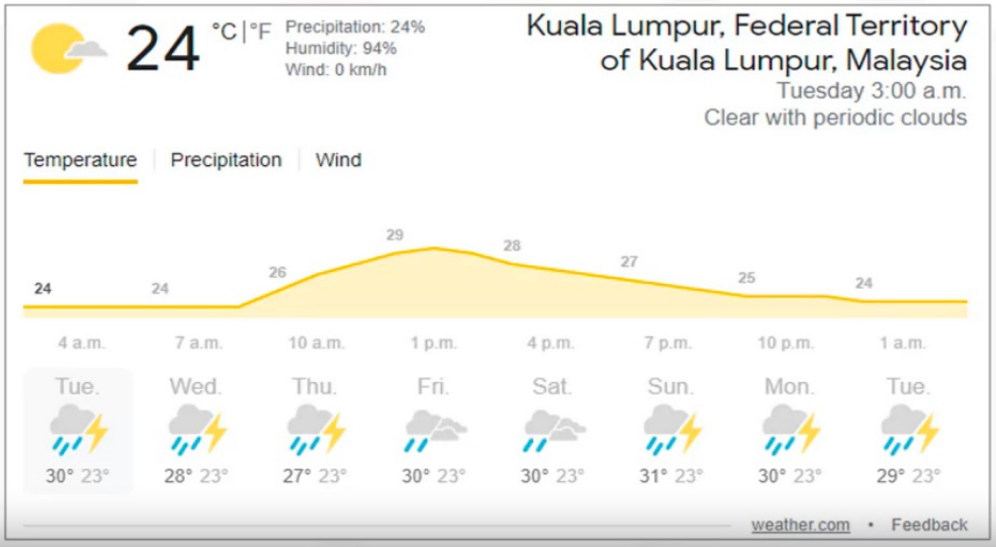
- Define what facts are in context of data warehousing.
- Define what a fact table is.
- Define what dimensions are in context of data warehousing.
- Define what a dimension table is.
- Describe an example of a fact and its dimensions.

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

- Data can be lumped into two categories: Facts and dimensions.
- **Facts are usually quantities** which can be measured, such as temperature, number of sales, or millimeters of rainfall.
- But facts **can also be qualitative** in nature. Dimensions are attributes which can be assigned to facts.
- Dimensions provide context to facts, which makes facts useful.
- For example, a temperature such as “24 degrees celsius,” all by itself, is not meaningful information.

Facts and dimensions



- Let's look at a familiar example. Here we have a weather report obtained by googling "weather in Kuala Lumpur."
- The **facts** here are things like the temperature, humidity, probability of precipitation, and wind speed. Notice that these are all quantities.
- But there are **other facts** here, such as this **icon,** which means "partly cloudy," or the **statement** "Clear with periodic clouds," and these other icons which indicate forecasted conditions, such as thundershowers.
- All of these **non-numeric facts** are examples of qualitative facts. To make sense of these facts, we need to provide context.
- Context is provided by dimensions, which includes things like the location, "Kuala Lumpur, Malaysia" and the day and time, "Tuesday at 3 a.m."
- The statement "24°C in Kuala Lumpur, Malaysia on Tuesday, August 17th at 3:00 a.m." means something

Fact tables

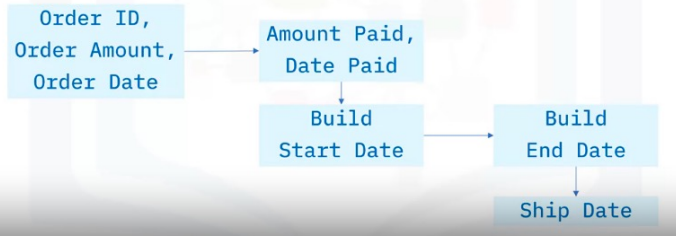
- Facts of a business process, plus
- Foreign keys to dimension tables
  - Dollar amounts for sales transactions
- Can contain detail level facts, or
- Facts that have been aggregated
- Summary tables contain aggregated facts
  - "Quarterly Sales" summary table, with
  - "store_id" as foreign key

- 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 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."

Accumulating snapshot fact tables

- Used to record events during a well-defined business process



Dimensions

- Dimensions categorize facts
- Called categorical variables in stats and machine learning
- Used to answer business questions
- Used for filtering, grouping, and labelling
  - People, product, and place names, and date or time stamps
- Dimension table stores dimensions of a fact
  - Joined to fact table via foreign key

- 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.

- Some examples of **various types of dimension** tables include: Product tables, which describe products, such as make, model, color, and size.
- Employee tables, which describe employees, such as name, title, and department.
- Temporal tables, which describe time at the level of granularity, or precision, at which events are recorded.
- Geography tables, for location data such as country, state, city, and postal or zip code.

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:

Example schema with fact & dimension tables

- Let us look at an example of a schema with both fact and dimension tables to help illustrate their relationships.
- 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.

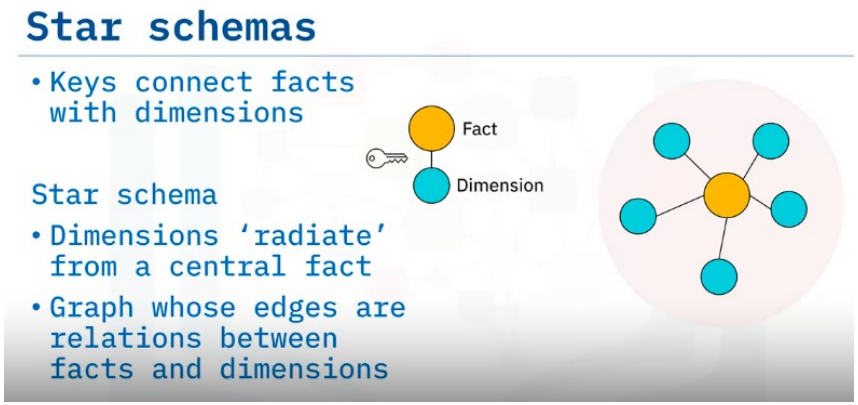
Summary

In this video, you learned that:

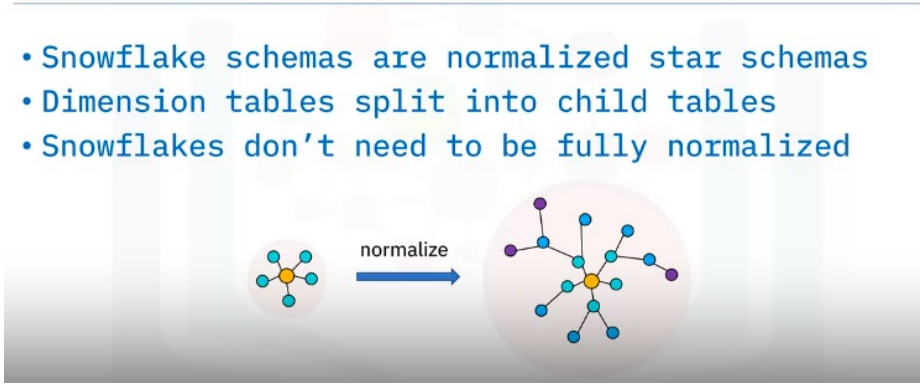
- Business data includes facts and dimensions
- Facts measure business processes, such as sales transactions
- Dimensions such as ‘sold_by’ and ‘store_id’ categorize facts
- Dimensions, also known as categorical variables, are used for filtering, grouping, and labelling
- Fact and dimensions are linked using foreign – primary keys

Data Modeling using Star and Snowflake Schemas

- Describe star schema modeling in terms of facts and dimensions.
- Describe snowflake schema as an extension of Star schema.
- Distinguish star from snowflake schema in terms of normalization.
- Recall that a fact table contains foreign keys that refer to the primary keys of dimension tables.



- 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."

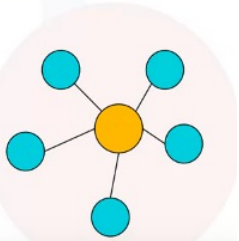


- **Snowflake s**chemas are **a generalization of star schemas** and can be seen as normalized star schemas.
- Normalization means separating the levels or hierarchies of a **dimension table into separate child tables.**
- A schema need not be fully normalized to be considered a snowflake, so long as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

Modeling with a star schema

Design considerations

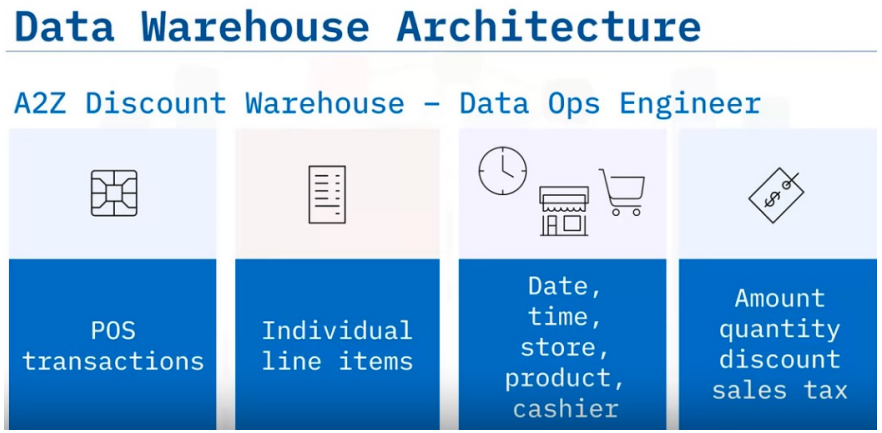
1. Select a business process
2. Choose level of detail
3. Identify the dimensions
4. Identify the facts



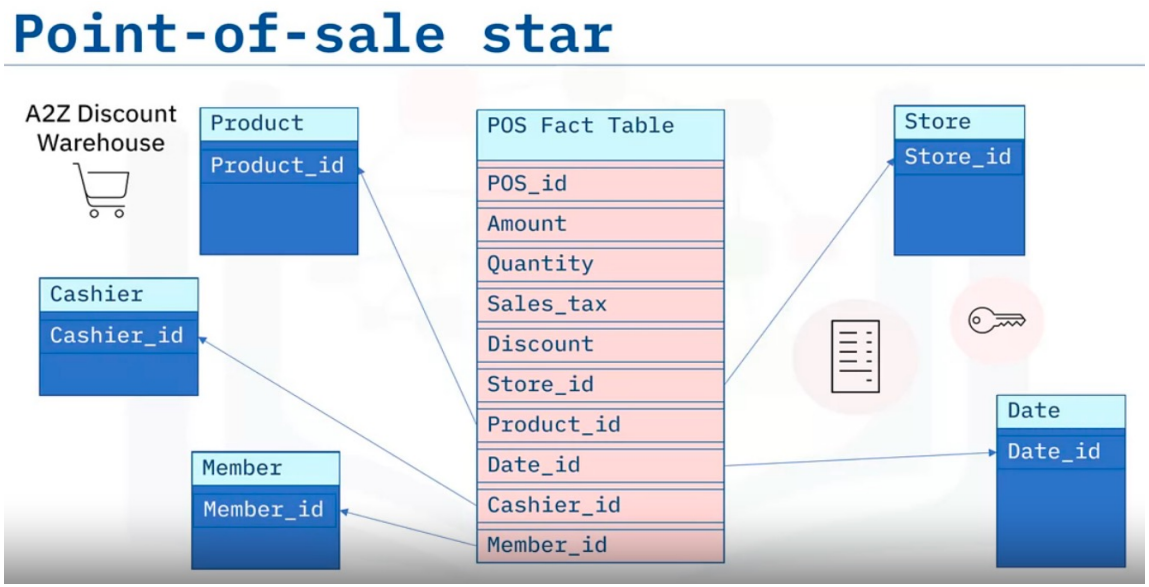
- Let's look at some general principles you need to consider when designing a data model for a star schema.
- The first step involves **selecting a business** process as the basis for what you want to model. You might be interested in processes such as sales, manufacturing, or supply chain logistics.
- In step two, you need to **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.
- Next in the process, you need to identify the dimensions. These may include attributes such as the date and time, and names of people, places, and things.
- The final consideration in designing star schemas is to identify the facts. These are the things being measured in the business process.

Let's 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, 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."
- 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.



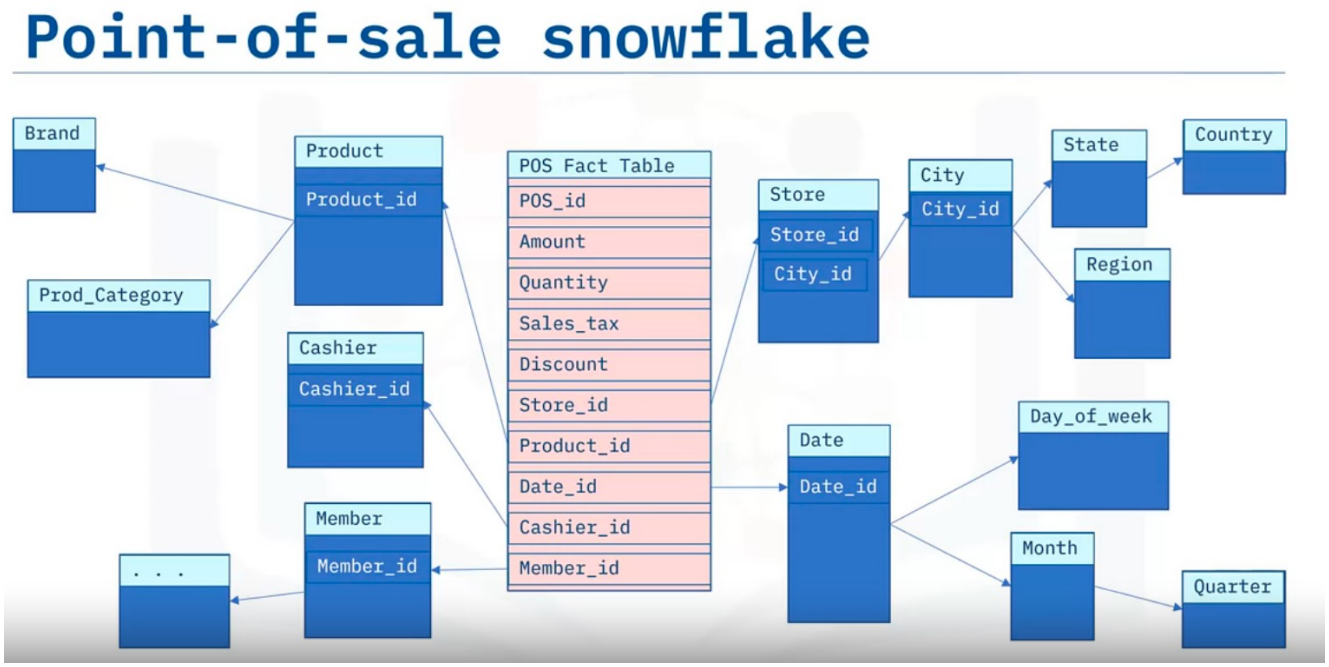
Attributes



Building star schema for "A to Z Discount Warehouse."

- At the **center** of your star schema sits a "**point-of-sales fact table**," which contains a unique ID, called "P O S ID," for each line item in the transaction, plus the following **facts, or measures**: the amount of the transaction in dollars, the quantity, or number of items involved, the sales tax, and any discount applied.
- There may be other facts to include, but these can be **adde**d later **as you discover** them.
- Each line item from a sales transaction has many dimensions associated with it.
- You include them as foreign keys in your fact table, or as links to the primary keys of your dimension tables.
- For example, the name of the store at which the item was sold is kept in a **dimension table called "store,"** which is identified in the fact-table by the value of the foreign "Store ID" key, which is the primary key for the Store table.
- Product information is stored in the Product table, which is uniquely identified by the "ProductID" key.
- Similarly, the date of the transaction is keyed by the "Date ID," which cashier entered the transaction is keyed by the "Cashier ID," and which member was involved is indicated by the "Member ID."

- Let's see how you can use **normalization** to extend your star schema to a Snowflake schema.
- 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.



- A separate city table can be used to record which city the store is in, while a foreign 'city id' key would be included in the 'Store' table to maintain the link.
- You might also have tables and keys for the city's state or province, and a pre-defined sales region for the store, and for which country the store resides in.
- We've left out the associated keys for simplicity.
- We can continue to normalize other dimensions, like the product's brand, and a "product category" that it belongs to, the day of week and the month corresponding to the date, plus the quarter, and so on.
- This normalized version of the star schema is called a snowflake schema, due to its multiple layers of branching which resembles a snowflake pattern.
- Much like how pointers are used to point to memory locations in computing, normalization reduces the memory footprint of the data.

Summary

In this video, you learned that:

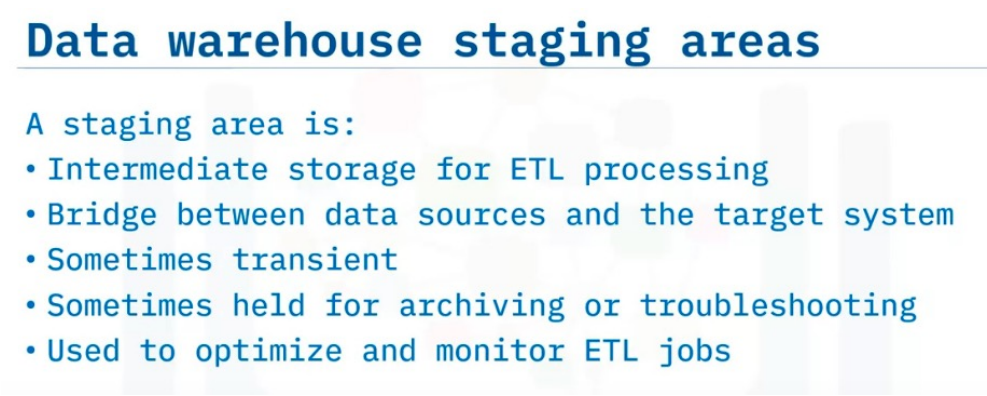
- Facts and dimension tables, linked by keys, form star or snowflake modeling schemas
- Design considerations include identifying a business process, its granularity, and its facts and dimensions
- Snowflake schemas are normalized star schemas
- Normalization involves separating dimensions into their hierarchies

Staging Areas for Data Warehouses

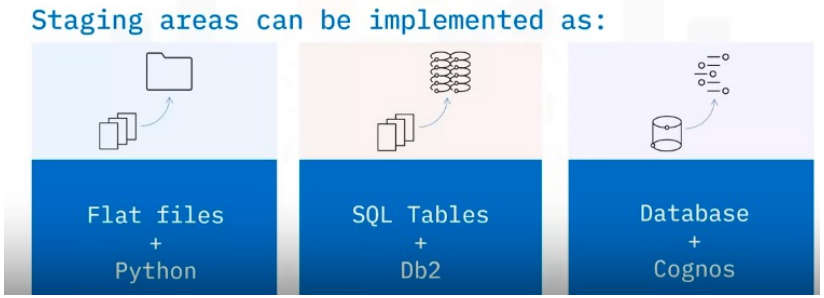
- Describe what a data warehouse staging area is.
- Describe why a staging area may be used.
- Relate how a staging area is used as a first step for integrating data sources.

What is a data warehouse staging area?

- You can think of a staging area as **an 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.



Data warehouse staging areas



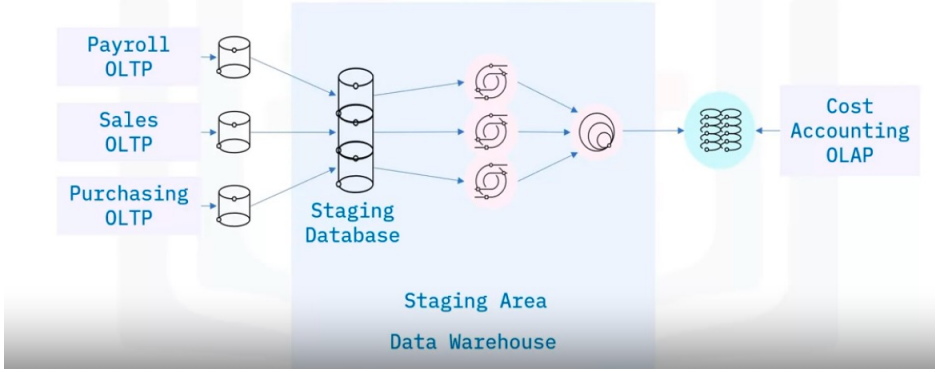
Staging areas can be implemented in many ways, including:

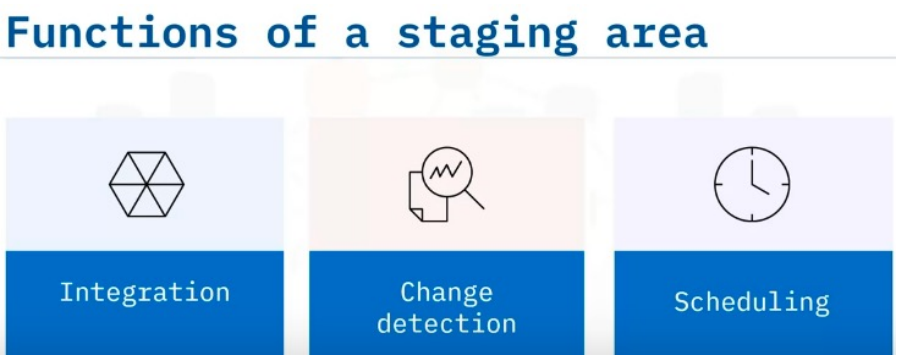
- **simple flat files**, such as csv files, stored in a directory, and managed using tools such as **Bash or Python**,
- or a set of SQL tables in a relational database such as Db2,
- or a self-contained database instance within a data warehousing or business intelligence platform such as Cognos Analytics.

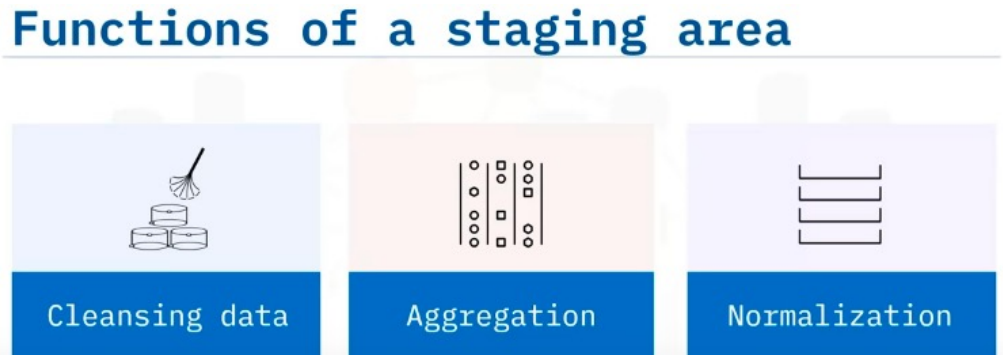
Example

- to illustrate a possible architecture for a Data Warehouse containing a **Staging Area,** which in turn includes an **associated Staging Database**.
- Imagine the enterprise would like to create a dedicated "Cost Accounting" Online Analytical Processing system.
- The required data is managed in **separate** OLTP 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.

DW staging area example







staging area can have many functions. Some typical ones include:

- **Integration:** 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**. For example, 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?

- Separate location where data from source systems is extracted to
- Decouples operations such as validation, cleansing and other processes
- Minimizes corruption risk
- Simplifies ETL workflows
- Simplifies recovery

- 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, and simplifies ETL workflow construction, operation, and maintenance.
- If any of the extracted data becomes corrupted somehow, you can easily recover.

Summary

In this video, you learned that staging areas:

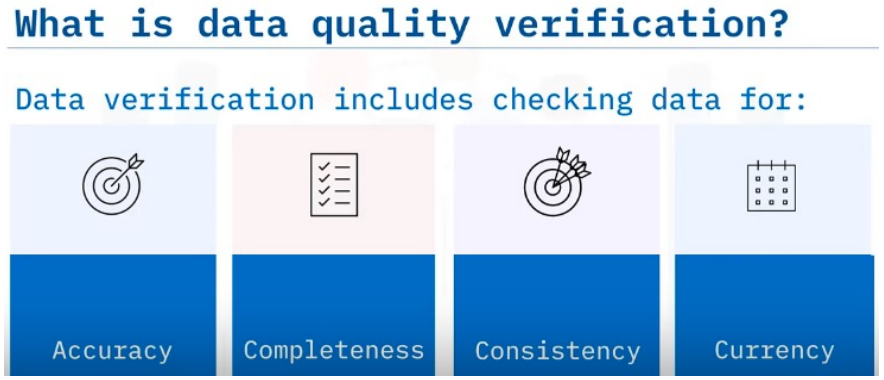
- Are used to integrate data sources into target systems
- Can be implemented as flat files, or as tables in a database
- Decouple processing from source systems, helping to minimize data corruption
- Are often transient, also used for archiving and troubleshooting

Verify Data Quality

- Define data quality verification.
- Identify why organizations verify data.
- List examples of data quality concerns.
- Outline a process for handling bad data.

Data verification includes checking your data for:

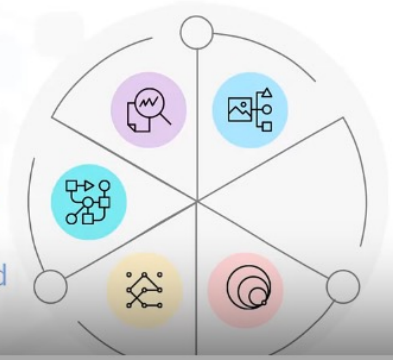
- Accuracy— Is your data correct?
- Completeness—Is there missing data?
- Consistency— Are fields consistently entered? and,
- Currency—Is your data up to date?



Why use data quality verification?

Data verification provides you with:

- A complete, connected view of your organization
- Data ready for advanced analysis
- Statistical modeling and machine learning



- 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.

Why use data quality verification?

"$3.1 trillion, IBM's estimate of the yearly cost of poor quality data, in the US alone, in 2016. While most people who deal in data every day know that bad data is costly, this figure stuns."

Harvard Business Review

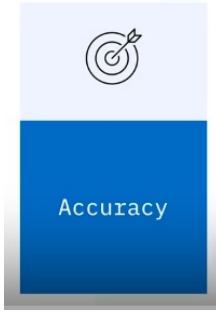
https://hbr.org/2016/09/bad-data-costs-the-u-s-3-trillion-per-year

- Data quality is not a top concern among the daily chaos of running a company.
- According to HBR, IBM's 2016 estimate of the yearly cost of poor data quality, in the US alone, was over 3 trillion dollars.
- Let's identify data quality concerns that organizations contend with.

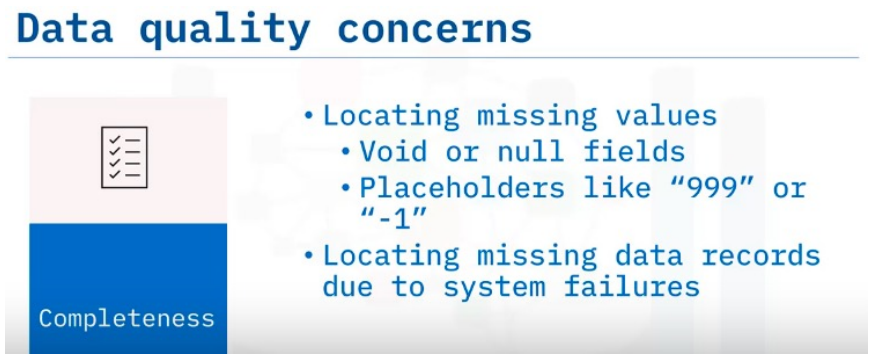
Accuracy.

- Accuracy includes ensuring a match between source data and destination data.
- How can accuracy become an issue?
- Data migrating from source systems often contains **duplicated records**.
- When users enter data manually**, typos** can find their way into the data records, yielding out-of-range values, outliers, and spelling mistakes.
- Sometimes large chunks of data become **misaligned**, causing **data corruption.**
- For example, a CSV file might contain a legitimate comma, which the new system can misinterpret as a column separator.

Data quality concerns

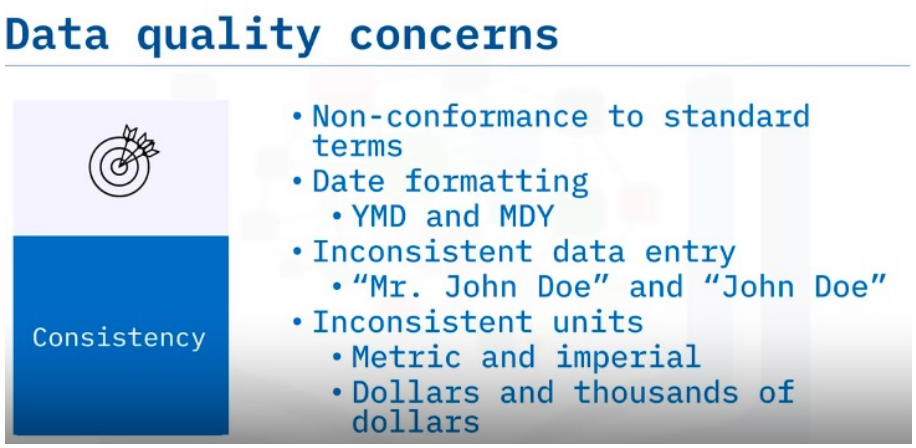


- Source and destination records match
- Duplicated records
- Typos
  - Out-of-range values
  - Spelling errors
- Mass misalignment
  - Misinterpretation of a comma as a separator



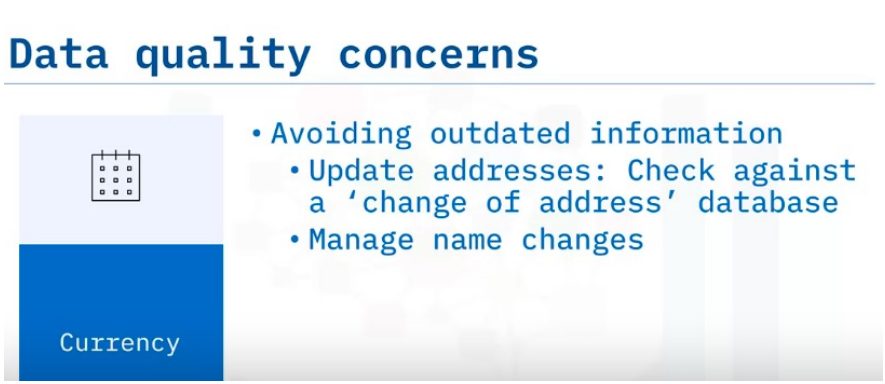
completeness.

- Data is incomplete when the business finds missing values, such as voids or nulls in fields that should be populated, or haphazard use of placeholders such as “9 9 9” or “minus 1” to indicate a missing value.
- Entire records can also be missing due to upstream system failures.



Consistency

- Are there deviations from standard terminology? Are dates entered consistently?
- For example, year-month-day and month-day-year formats are incompatible.
- Is data entered consistently? For example, Mr. John Doe and John Doe might refer to the same person in the real world, but the system will see them as distinct.
- Are the units consistent? For example, you are expecting ”kilograms,” but you might have entries based on “pounds,” or you are expecting 'dollar amounts,” but you might have entries based on “thousands of dollars.”



- Currency is about ensuring your data remains up to date.
- For example, you might have dimension tables that contain customer addresses, some of which might be outdated.
- In the US, you could check these against a change-of-address database and update your table as required.
- Another currency concern would be name changes as customers can change their names for various reasons.

- Determining how to resolve and prevent bad data can be a complex and iterative process.
- First, you'll implement rules to **detect** bad data.
- Then you'll apply those rules to **capture** and quarantine any bad data.
- You might need to **report** any bad data and share the findings with the appropriate domain experts.
- You and your team can **investigate** the root cause of each problem, searching for clues upstream in the data lineage. Once you **diagnose** each problem, you can begin **correcting** the issues.
- Ultimately, you want to automate the entire data cleaning workflow as much as possible.

Managing data quality – an example

Scenario: Data from specific data sources consistently has the following data quality issues:

- Missing data (NULL values)
- Out-of-range values
- Duplicate values
- Invalid values

- For 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. First, write SQL queries to detect these issues and test for them.
2. Next, 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. Next, 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

Managing data quality - an action plan

Here's how an organization might manage and resolve these issues:

1. Write SQL queries to detect and test for these conditions
2. Create rules for correcting and managing those conditions
3. Create a script that runs the data validation SQL queries every night
4. Automate the script created in step 3 that performs detection and correction processes
5. Review the automated report and address issues that could not be resolved automatically

Data quality solutions

- IBM InfoSphere Server, & IBM Data Refinery
- 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

IBM InfoSphere

End-to-end data quality tools to:

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

- The “IBM InfoSphere Information Server for Data Quality” is an example of a product that can help you perform data verification in a unified environment.
- “InfoSphere Information Server for Data Quality” enables you to continuously monitor the quality of your data, and keep your data clean on an ongoing basis, helping you turn your data into trusted information.
- In addition, the “IBM InfoSphere Information Server for Data Quality” comes with built-in, end-to-end data quality tools to:

1. Help you understand your data and its relationships.
2. Monitor and analyze data quality continuously.
3. Clean, standardize, and match data; and
4. Maintain data lineage, which is the history of the data’s origin and what happened to the data along the way.

Summary

In this video, you learned that:

- Data verification includes checking your data for accuracy, completeness, consistency, and currency
- Data verification is about managing data quality, enhancing data reliability, and maximizing data value
- Determining how to resolve and prevent bad data can be a complex and iterative process
- IBM InfoSphere Information Server for Data Quality can help you perform data verification in a unified environment

Populating a Data Warehouse

- Describe populating a data warehouse as an ongoing process.
- List the main steps for populating a data warehouse.
- List methods for change detection and incremental loading.
- Manually create and populate tables for a sales star schema.
- Recall the periodic maintenance required to keep your data warehouse running smoothly.

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

- You have an initial load followed by periodic incremental loads. For example, you may load new data every day or every week.
- Rarely, a full refresh may be required in case of major schema changes or catastrophic failures.
- Generally, fact tables are dynamic and require frequent updating while dimension tables don't change often.
- For example, lists of cities or stores are quite static, but sales happen every day.

Typical ways of loading data

- Many tools can help you automate the ongoing loading process
- Db2 has a Load utility that is faster than inserting a row at a time
- You can automate loading as part of your ETL pipeline using tools like Apache Airflow and Kafka
- You can use 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

- Many tools are available to automate the ongoing process of keeping your data warehouse current.
- Databases like Db2 have a Load utility that is faster than inserting a row at a time, and 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.
- And InfoSphere DataStage allows you to compile and run jobs to load your data.

Before populating your data warehouse, ensure that:

- Your schema has already been modeled.
- Your data has been staged in tables or files.
- And, you have mechanisms for verifying the data quality.
- Now you are ready to set up your data warehouse and implement the initial load.
- You first instantiate the data warehouse and its schema, then create the production tables.
- Next, establish relationships between your fact and dimension tables, and finally, load your transformed and cleaned data into them from your staging tables or files.

Populating your data warehouse

Setup and initial load:

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

Populating your data warehouse

Ongoing loads:

- You can automate incremental loads using a script as part of your ETL pipeline
- For example, append data daily or weekly
- Incremental loading requires you to have a method to correctly detect new and changed data

- Now that you've gone through the initial load, it's time to set up ongoing data loads.
- You can automate subsequent incremental loads using a script as part of your ETL data pipeline.
- You can also schedule your incremental loads to occur daily or weekly, depending on your needs.
- You will also need to include some logic to determine what data is new or updated in your staging area.

- 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.

Change detection

- Normally, you detect changes in the source system
- Many RDBMSs have change tracking to identify new, changed, or deleted records
- You might have timestamps identifying when data was first written and when it was modified
- Otherwise, you may need to use brute-force comparison to your staged data

Periodic maintenance

- Perform monthly or yearly data archiving
- Script both the deletion of older data and its archiving to slower, less costly storage

- 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

- Let's illustrate the process with a simplified example of manually populating a data **warehouse** with a star **schema called 'sales.'** We'll assume that you've already instantiated the data warehouse and the 'sales' schema.
- Here's a sample of some auto sales transaction data from a fictional company called Shiny Auto Sales.
- You can see several foreign key columns, such as "sales ID," which is a sequential key identifying the sales invoice number, "emp no," which is the employee number, and "class ID," which encodes the type of car sold, such as "small SUV."
- Each of these keys represents a dimension that points to a corresponding dimension table in the star schema. The "date" column is a dimension that indicates the sale date.
- The "amount" column is the sales amount, which happens to be the fact of interest.
- This table is already close to the form of a fact table. The only exception is the date column, which is not yet represented by a foreign "date ID" key.

Example: auto sales data

Manually populating 'sales' star schema DW:

	sales_id	amount	empl_no	class_id	date
0	1629	42000.00	642	30	2021-01-04
1	1630	17680.00	617	70	2021-01-04
2	1631	37100.00	642	70	2021-01-05
3	1632	26500.00	680	60	2021-01-05
4	1633	8200.00	707	71	2021-01-05
5	1634	42099.00	642	40	2021-01-05
6	1635	12099.00	720	61	2021-01-06
7	1636	51999.00	607	40	2021-01-06

Create a dimension table

Create a salesperson dimension table:

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

- Let's use PSQL, the terminal-based front end for PostGreSQL, to illustrate how you can create your dimension tables using the salesperson dimension as an example.
- Use the CREATE TABLE clause to create the "DimSalesPerson" table with the "sales" schema, along with "SalesPersonID" as a serial primary key, "SalespersonAltID", as the salesperson's employee number, and finally, a column for the salesperson's name.

- Now you can start populating the "DimSalesPerson" table, row by row.
- You use an "insert into" clause on the "sales dot DimSalesPerson" table, specifying the "SalesPersonAltID" and "SalesPersonName" columns, and begin inserting values such as employee number 680, "Cadillac Jack."
- You would similarly create and populate tables for the remaining dimensions

Populate the dimension tables

```
INSERT INTO sales.DimSalesPerson(SalesPersonAltID,
                                 SalesPersonName)
values
( 617, 'Gocart Joe'),
( 642, 'Jake Salesbouroughs'),
( 680, 'Cadillac Jack'),
( 707, 'Jane Honda'),
( 720, 'Kayla Kaycar'),
( 607, 'William Jeepman'),
( 609, 'Happy Dollarmaker'),
```

View the salesperson table

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

salespersonid   |  salespersonaltid |      salespersonname
----------------+-------------------+----------------------
            1   |               617 |           Gocart Joe
            2   |               642 |   Jake Salesboroughs
            3   |               680 |        Cadillac Jack
            4   |               707 |           Jane Honda
            5   |               720 |         Kayla Kaycar
```

- You can enter the SQL statement: “SELECT star FROM sales dot dim salesperson LIMIT 5” to view your salesperson dimension table, and see that everything seems to be correctly populated, such as record 1, employee number 617, and salesperson name “Go-cart Joe.”

Create the sales fact table

Create an auto sales fact table:

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

- Now it’s time to create your sales fact table, using “CREATE TABLE” with “sales dot FactAutoSales” as the table name, “TransactionID“ as the primary key, with “big serial” type and the various foreign keys, such as “SalesID” and “AutoClassID”, and finally the fact of interest, “amount” as type “money.”

Set up table relations

Create relations between fact and auto class tables:

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

- Next, you proceed with setting up the relations between the fact and dimension tables of the sales schema.
- For example, you can apply the ALTER TABLE statement and the ADD CONSTRAINT clause to the “sales dot FactAutoSales” fact table to add “KVAutoClassID” as a foreign key relating “AutoClassID” to the same column name in the “sales dot DimAutoCategory” table using the REFERENCES clause.
- You would then use the same method to set up the relations for the remaining dimension tables.

- After defining all the tables and setting up the corresponding relations, it’s finally time to start populating your fact table using the sales data that you started with.
- You can use the INSERT INTO statement on “sales dot FactAutoSales,” specifying the column names “SalesID,” "Amount," “SalesPersonID,” “AutoClassID,” and “SalesDateKey,” and entering rows of values such as 1629, 42000, 2, 1, and 4, which you would obtain using the auto sales data.

Populate the sales fact table

Populate the Auto 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;
```

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

- You can view the auto sales fact table by entering the SQL statement “select star" from “sales dot FactAutoSales Limit 5” to display its first 5 rows.
- Here you see the dollar amounts for individual auto sales, the primary key called “transactionID,” and the remaining columns, which are the foreign keys that you set up.

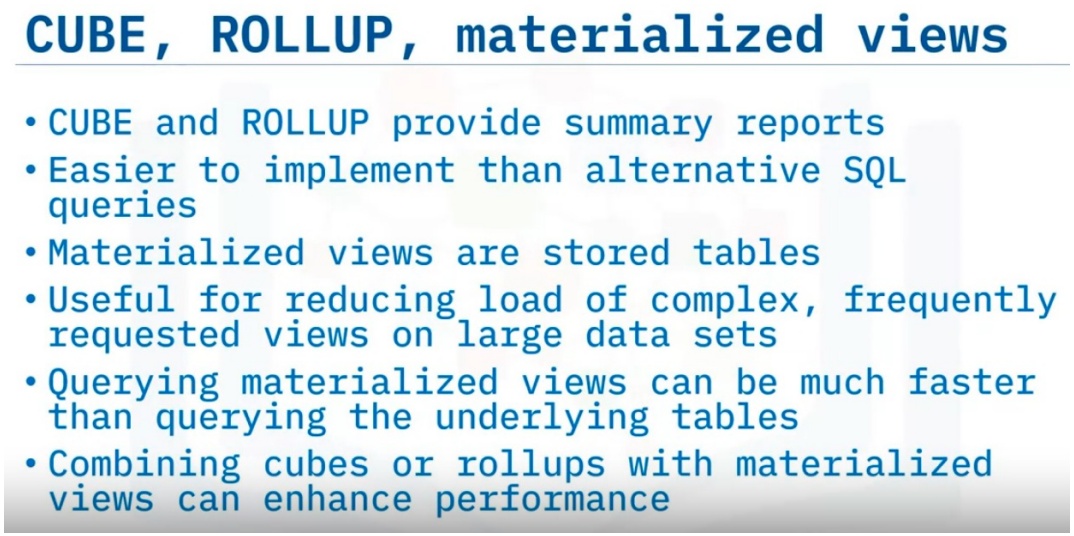
Summary

In this video, you learned that:

- Populating an EDW includes creation of production tables and their relations and loading of clean data into tables
- Loading the EDW is an ongoing process beginning with an initial load, followed by periodic incremental loads
- Fact tables require frequent updating while dimension tables don’t change often
- Incremental loading and periodic maintenance can be scripted and scheduled as part of your ETL pipeline

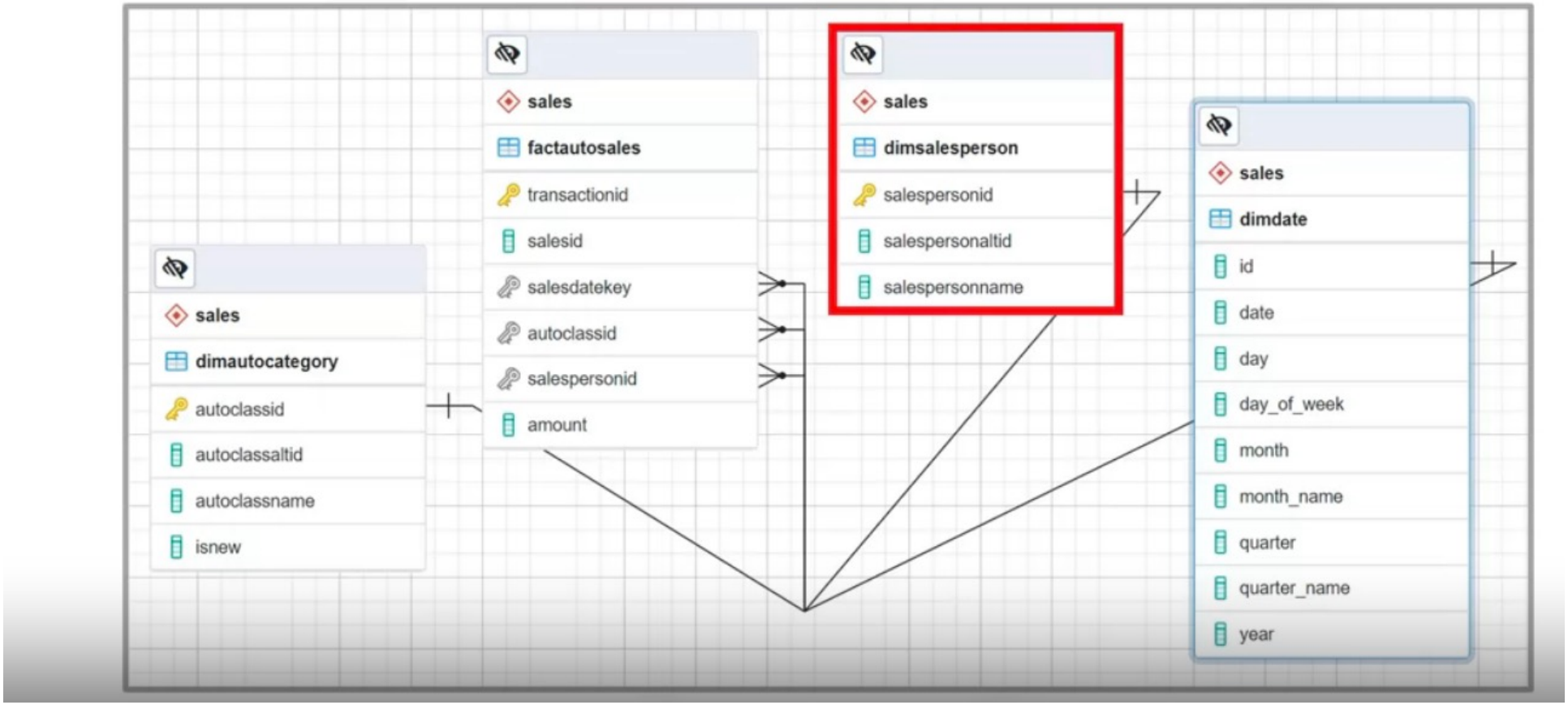
Querying the Data

- Interpret an entity-relationship diagram, or ERD, for a star schema and use the relations between tables to set up queries.
- Create a materialized view by denormalizing, or joining tables from, a star schema.
- Apply the CUBE and ROLLUP options in a GROUP BY clause to generate commonly requested total and subtotal summaries.



- CUBE and ROLLUP operations generate the kinds of summaries that management often requests.
- These summaries are much 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



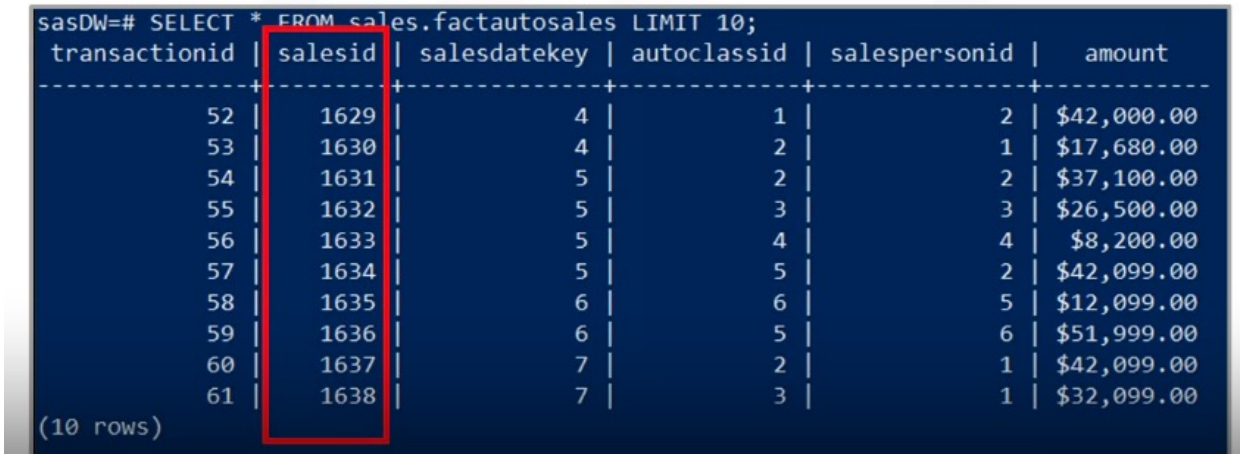
Consider the following scenario:

- You have the task of creating some live summary tables for reporting January sales by salesperson and automobile type for ShinyAutoSales.
- Begin by understanding the existing star schema in their data warehouse, called "sasDW," based on PostgreSQL.
- Then explore relevant ShinyAutoSales data by querying the tables from the "sales" star schema in the sasDW warehouse.
- After exploring the schema, you decide to create a materialized view as a staging table.
- Creating the view as a staging table provides you with the data you need while minimizing your impact on the database.
- You can incrementally refresh the data at will during off-peak hours.
- You start a PostgreSQL session and generate an entity relationship diagram, or ERD, which represents the "Sales" star schema implemented within the ShinyAutoSales data warehouse, "sasDW."
- Then, you locate the central fact table named "fact auto sales." This table contains the "amount" column, which is the measure you need.
- You also spot the three foreign keys in the sales fact table: "sales date key," "auto class ID," and "salesperson ID."

These keys link respectively to:

- The "Date dimension table," which contains dates and related values such as the day of the week, month name, and quarter.
- The "Auto category dimension table," which includes the "auto class name," and the Boolean "is new" column, and finally, the "Salesperson dimension table," which contains the "salesperson's name."

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)
```

- In this example, you are using PostgreSQL. Let's assume you already started up the terminal-based front-end to PostgreSQL, "P S Q L," and connected to the "S A S D W" data warehouse.
- Notice the command prompt contains the name of the data warehouse you are connected to, "S A S D W."
- Starting with the auto sales fact table, you'll enter the SQL statement "select star from sales dot fact auto sales limit 10" to display its first 10 rows.
- Here, you see the dollar amounts for individual auto sales, but the remaining columns are primary and foreign keys, which don't have any direct meaning for you yet.
- However, you notice that the sales ID values are sequential, but the numbering starts at 1,629 instead of 1.
- That's because ShinyAutoSales has provided you with access to a windowed subset of their data.

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)
```

Now, you can see meaningful names for various automobile classes, such as truck and compact
SUVs.
You notice duplicate entries for the truck class and wonder why they exist.
When you look more closely, you realize the duplicate entries exist because of the distinct
subclasses for new and used trucks.

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)
```

Similarly, you generate a view for the salesperson dimension table and find eight distinct salesperson
names, including "Gocart Joe" and "Jane Honda."

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)
```

you view the date dimension table.
You notice the dates only go back to January 1, 2021.
Your contact at Shiny Auto Sales informs you that she will provide you with more data later
and that for now, you can work with a smaller data set while you develop your queries.
The date table contains potentially useful date elements such as the day of the week,
month name, and quarter name

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
```

- At this stage, it would be more convenient to have a table of data that contains the dimensions you need with human interpretable columns, rather than just keys.
- Essentially, you want to create a denormalized view of the data by joining the dimensions back to the fact of interest.
- You proceed by selecting the "date," "auto class name," "is new," "salesperson name," and "amount" columns from their tables, and joining each dimension onto the "amount" fact using an inner join on the corresponding keys.
- Next, why not capture the view as a materialized view called "Denormalized sales" or "D N sales" for short?
- Then you can reuse the materialized view for different queries without having to recreate your work.
- You accomplish this task using the clause "CREATE MATERIALIZED VIEW D N sales AS," followed by the same query you used to generate the denormalized view.

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)
```

- Type "Select star from D N sales, LIMIT 10" to display your resulting materialized view.
- Now you have a tidy, human-readable, time-series of sales data available for further analysis.
- For example, you can see that "Cadillac Jack" sold a new midsize SUV on January 5 for $26,500.

- Next, you want to apply CUBE and ROLLUP operations to your denormalized, materialized view. Let's see the CUBE results.
- Here, you select the "auto class name," "salesperson name," and the "sum of the sales amounts" from "D N sales," where "is new" is set to "true."
- Finally, group the generated cube by the "auto class name" and "salesperson name."
- The output looks like this: The first row has no entries in the dimensions columns, which means 'all.'
- Thus, the value of $366,076 represents the total sales for all new cars.
- The next block of records has both dimension columns populated.
- So, for instance, you can read the total sales of new midsize SUVs by "Gocart Joe," which is $32,099.
- Similarly, the last two blocks summarize "new auto sales" by class, and by salesperson.

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)

- you apply a ROLLUP instead of a CUBE operation. You decide to keep the query the same as the previous query, except that you replace CUBE with ROLLUP.
- Here's what the resulting view looks like now.
- You have five fewer rows with the ROLLUP result than CUBE, resulting in 13 rows instead of 18 rows.
- The only difference in this result is that you don't have the "total sale amounts by salesperson" summary.
- While CUBE generates all possible permutations of the "GROUP BY" columns, ROLLUP only looks at the single permutation defined by the columns' order listed in the ROLLUP call.

Summary

In this video, you learned that:

- CUBE, ROLLUP, and materialized views help query and analyze data in data warehouses
- CUBE and ROLLUP operations easily generate fact summaries required by management
- You can denormalize star schemas using joins to bring together human-interpretable facts and dimensions in a single materialized view
- Staging tables can be implemented as incrementally refreshing materialized views