**Work with Data Warehouses using Azure Synapse Analytics**

Relational data warehouses are at the center of most enterprise business intelligence (BI) solutions. While the specific details may vary across data warehouse implementations, a common pattern based on a denormalized, multidimensional schema has emerged as the standard design for a relational data warehouse.

Azure Synapse Analytics includes a highly scalable relational database engine that is optimized for data warehousing workloads. By using *dedicated SQL pools* in Azure Synapse Analytics, you can create databases that are capable of hosting and querying huge volumes of data in relational tables.

In this module, you'll learn how to:

* Design a schema for a relational data warehouse.
* Create fact, dimension, and staging tables.
* Use SQL to load data into data warehouse tables.
* Use SQL to query relational data warehouse tables.

# Design a data warehouse schema

Like all relational databases, a data warehouse contains tables in which the data you want to analyze is stored. Most commonly, these tables are organized in a schema that is optimized for multidimensional modeling, in which numerical measures associated with events known as facts can be aggregated by the attributes of associated entities across multiple dimensions. For example, measures associated with a sales order (such as the amount paid or the quantity of items ordered) can be aggregated by attributes of the date on which the sale occurred, the customer, the store, and so on.

## Tables in a data warehouse

A common pattern for relational data warehouses is to define a schema that includes two kinds of table: dimension tables and fact tables.

### Dimension tables

Dimension tables describe business entities, such as products, people, places, and dates. Dimension tables contain columns for attributes of an entity. For example, a customer entity might have a first name, a last name, an email address, and a postal address (which might consist of a street address, a city, a postal code, and a country or region). In addition to attribute columns, a dimension table contains a unique key column that uniquely identifies each row in the table. In fact, it's common for a dimension table to include ***two*** key columns:

* a surrogate key that is specific to the data warehouse and uniquely identifies each row in the **dimension table** in the data warehouse - usually an incrementing integer number. (NOTE: **Surrogate keys are present in dimension tables as the primary key and in fact tables as foreign keys to dimensions**. **We don’t create surrogate keys in fact tables**)
* An alternate key, often a natural or business key that is used to identify a specific instance of an entity in the transactional source system from which the entity record originated - such as a product code or a customer ID.

**Note**

Why have two keys? There are a few good reasons:

* The data warehouse may be populated with data from multiple source systems, which can lead to the risk of duplicate or incompatible business keys.
* Simple numeric keys generally perform better in queries that join lots of tables - a common pattern in data warehouses.
* Attributes of entities may change over time - for example, a customer might change their address. Since the data warehouse is used to support historic reporting, you may want to retain a record for each instance of an entity at multiple points in time; so that, for example, sales orders for a specific customer are counted for the city where they lived at the time the order was placed. In this case, multiple customer records would have the same business key associated with the customer, but different surrogate keys for each discrete address where the customer lived at various times.

| **CustomerKey** | **CustomerAltKey** | **Name** | **Email** | **Street** | **City** | **PostalCode** | **CountryRegion** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 123 | I-543 | Navin Jones | navin1@contoso.com | 1 Main St. | Seattle | 90000 | United States |
| 124 | R-589 | Mary Smith | mary2@contoso.com | 234 190th Ave | Buffalo | 50001 | United States |
| 125 | I-321 | Antoine Dubois | antoine1@contoso.com | 2 Rue Jolie | Paris | 20098 | France |
| 126 | I-543 | Navin Jones | navin1@contoso.com | 24 125th Ave. | New York | 50000 | United States |
| ... | ... | ... | ... | ... | ... | ... | ... |

**Note**

Observe that the table contains two records for *Navin Jones*. Both records use the same alternate key to identify this person (*I-543*), but each record has a different surrogate key. From this, you can surmise that the customer moved from Seattle to New York. Sales made to the customer while living in Seattle are associated with the key *123*, while purchases made after moving to New York are recorded against record *126*.

In addition to dimension tables that represent business entities, it's common for a data warehouse to include a dimension table that represents *time*. This table enables data analysts to aggregate data over temporal intervals. Depending on the type of data you need to analyze, the lowest granularity (referred to as the *grain*) of a time dimension could represent times (to the hour, second, millisecond, nanosecond, or even lower), or dates.

An example of a time dimension table with a grain at the date level might contain the following data:

| **DateKey** | **DateAltKey** | **DayOfWeek** | **DayOfMonth** | **Weekday** | **Month** | **MonthName** | **Quarter** | **Year** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 19990101 | 01-01-1999 | 6 | 1 | Friday | 1 | January | 1 | 1999 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 20220101 | 01-01-2022 | 7 | 1 | Saturday | 1 | January | 1 | 2022 |
| 20220102 | 02-01-2022 | 1 | 2 | Sunday | 1 | January | 1 | 2022 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 20301231 | 31-12-2030 | 3 | 31 | Tuesday | 12 | December | 4 | 2030 |

The timespan covered by the records in the table must include the earliest and latest points in time for any associated events recorded in a related fact table. Usually there's a record for every interval at the appropriate grain in between.

### **Fact tables (In SAP it is txn tables)**

Fact tables store details of observations or events; for example, sales orders, stock balances, exchange rates, or recorded temperatures. A fact table contains columns for numeric values that can be aggregated by dimensions. In addition to the numeric columns, a fact table contains key columns that reference unique keys in related dimension tables.

For example, a fact table containing details of sales orders might contain the following data:

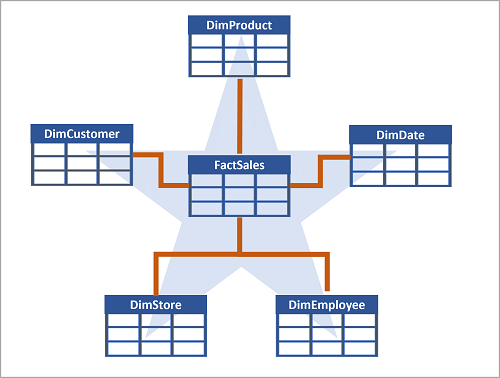
| **OrderDateKey** | **CustomerKey** | **StoreKey** | **ProductKey** | **OrderNo** | **LineItemNo** | **Quantity** | **UnitPrice** | **Tax** | **ItemTotal** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 20220101 | 123 | 5 | 701 | 1001 | 1 | 2 | 2.50 | 0.50 | 5.50 |
| 20220101 | 123 | 5 | 765 | 1001 | 2 | 1 | 2.00 | 0.20 | 2.20 |
| 20220102 | 125 | 2 | 723 | 1002 | 1 | 1 | 4.99 | 0.49 | 5.48 |
| 20220103 | 126 | 1 | 823 | 1003 | 1 | 1 | 7.99 | 0.80 | 8.79 |
| ... | ... | ... | ... | ... | ... | ... | ... |  | ... |

A fact table's dimension key columns determine its grain. For example, the sales orders fact table includes keys for dates, customers, stores, and products. An order might include multiple products, so the grain represents line items for individual products sold in stores to customers on specific days.

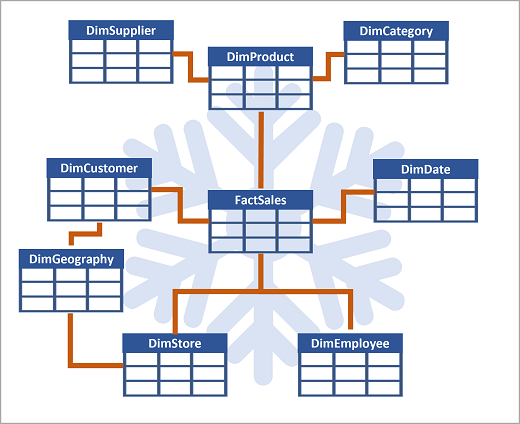
## Data warehouse schema designs

In most transactional databases that are used in business applications, the data is normalized to reduce duplication. In a data warehouse however, the dimension data is generally de-normalized to reduce the number of joins required to query the data.

Often, a data warehouse is organized as a star schema, in which a fact table is directly related to the dimension tables, as shown in this example:

]

The attributes of an entity can be used to aggregate measures in fact tables over multiple hierarchical levels - for example, to find total sales revenue by country or region, city, postal code, or individual customer. The attributes for each level can be stored in the same dimension table. However, when an entity has a large number of hierarchical attribute levels, or when some attributes can be shared by multiple dimensions (for example, both customers and stores have a geographical address), it can make sense to apply some normalization to the dimension tables and create a snowflake schema, as shown in the following example:



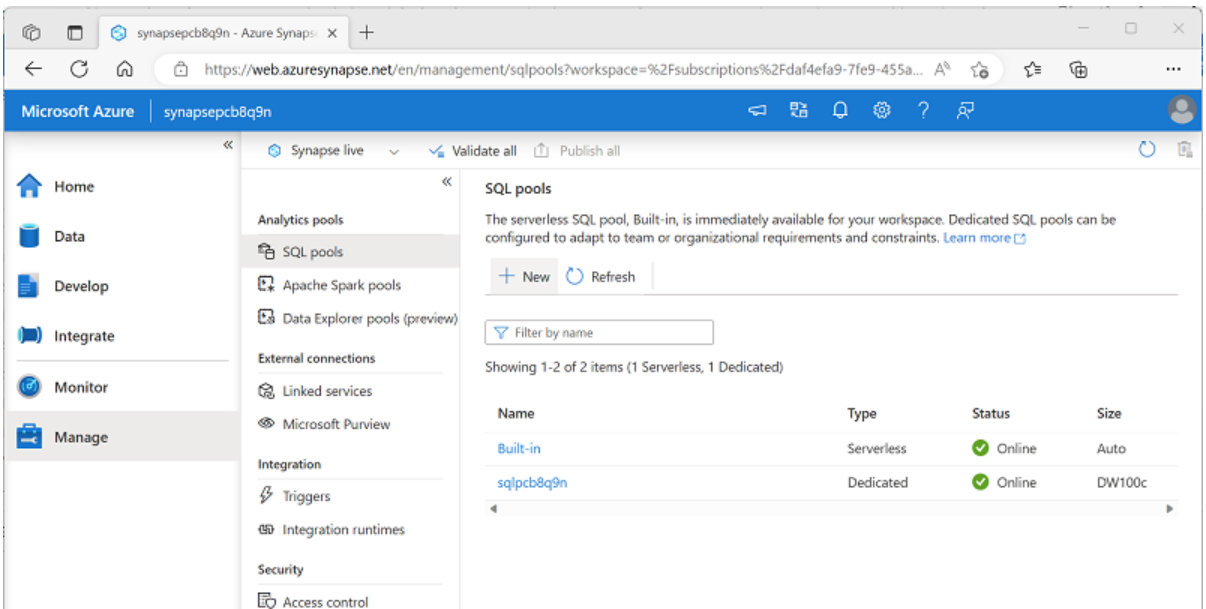
In this case, the **DimProduct** table has been normalized to create separate dimension tables for product categories and suppliers, and a **DimGeography** table has been added to represent geographical attributes for both customers and stores. Each row in the **DimProduct** table contains key values for the corresponding rows in the **DimCategory** and **DimSupplier** tables; and each row in the **DimCustomer** and **DimStore** tables contains a key value for the corresponding row in the **DimGeography** table.

<https://learn.microsoft.com/en-us/training/modules/design-multidimensional-schema-to-optimize-analytical-workloads/3-create-tables>

Now that you understand the basic architectural principles for a relational data warehouse schema, let's explore how to create a data warehouse.

## **Creating a dedicated SQL pool**

To create a relational data warehouse in Azure Synapse Analytics, you must create a dedicated SQL Pool. The simplest way to do this in an existing Azure Synapse Analytics workspace is to use the **Manage** page in Azure Synapse Studio, as shown here:



When provisioning a dedicated SQL pool, you can specify the following configuration settings:

* A unique name for the dedicated SQL pool.
* A performance level for the SQL pool, which can range from *DW100c* to *DW30000c* and which determines the cost per hour for the pool when it's running.
* Whether to start with an empty pool or restore an existing database from a backup.
* The *collation* of the SQL pool, which determines sort order and string comparison rules for the database. (*You can't change the collation after creation*).

After creating a dedicated SQL pool, you can control its running state in the **Manage** page of Synapse Studio; pausing it when not required to prevent unnecessary costs.

When the pool is running, you can explore it on the **Data** page, and create SQL scripts to run in it.

## **Considerations for creating tables**

To create tables in the dedicated SQL pool, you use the CREATE TABLE (or sometimes the CREATE EXTERNAL TABLE) Transact-SQL statement. The specific options used in the statement depend on the type of table you're creating, which can include:

* Fact tables
* Dimension tables
* Staging tables

**Note**

The data warehouse is composed of fact and dimension tables as discussed previously. Staging tables are often used as part of the data warehousing loading process to ingest data from source systems.

When designing a star schema model for small or medium sized datasets you can use your preferred database, such as Azure SQL. For larger data sets you may benefit from implementing your data warehouse in Azure Synapse Analytics instead of SQL Server. It's important to understand some key differences when creating tables in Synapse Analytics.

### **Data integrity constraints**

Dedicated SQL pools in Synapse Analytics don't support foreign key and unique constraints as found in other relational database systems like SQL Server. This means that jobs used to load data must maintain uniqueness and referential integrity for keys, without relying on the table definitions in the database to do so.

**Tip**

For more information about constraints in Azure Synapse Analytics dedicated SQL pools, see [**Primary key, foreign key, and unique key using dedicated SQL pool in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-table-constraints).

### **Indexes**

While Synapse Analytics dedicated SQL pools support clustered indexes as found in SQL Server, the default index type is clustered columnstore. This index type offers a significant performance advantage when querying large quantities of data in a typical data warehouse schema and should be used where possible. However, some tables may include data types that can't be included in a clustered columnstore index (for example, VARBINARY(MAX)), in which case a clustered index can be used instead.

**Tip**

For more information about indexing in Azure Synapse Analytics dedicated SQL pools, see [**Indexes on dedicated SQL pool tables in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-tables-index).

### **Distribution**

Azure Synapse Analytics dedicated SQL pools use a [massively parallel processing (MPP) architecture](https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/data-warehousing#data-warehousing-in-azure), as opposed to the symmetric multiprocessing (SMP) architecture used in most OLTP database systems. In an MPP system, the data in a table is distributed for processing across a pool of nodes. Synapse Analytics supports the following kinds of distribution:

* **Hash**: A deterministic hash value is calculated for the specified column and used to assign the row to a compute node.
* **Round-robin**: Rows are distributed evenly across all compute nodes.
* **Replicated**: A copy of the table is stored on each compute node.

The table type often determines which option to choose for distributing the table.

| **Table type** | **Recommended distribution option** |
| --- | --- |
| Dimension | Use replicated distribution for smaller tables to avoid data shuffling when joining to distributed fact tables. If tables are too large to store on each compute node, use hash distribution. |
| Fact | Use hash distribution with clustered columnstore index to distribute fact tables across compute nodes. |
| Staging | Use round-robin distribution for staging tables to evenly distribute data across compute nodes. |

**Tip**

For more information about distribution strategies for tables in Azure Synapse Analytics, see [**Guidance for designing distributed tables using dedicated SQL pool in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-tables-distribute).

## **Creating dimension tables**

|  |  |
| --- | --- |
| When you create a dimension table, ensure that the table definition includes surrogate and alternate keys as well as columns for the attributes of the dimension that you want to use to group aggregations. It's often easiest to use an IDENTITY column to auto-generate an incrementing surrogate key (otherwise you need to generate unique keys every time you load data). The following example shows a CREATE TABLE statement for a hypothetical **DimCustomer** dimension table. |  |

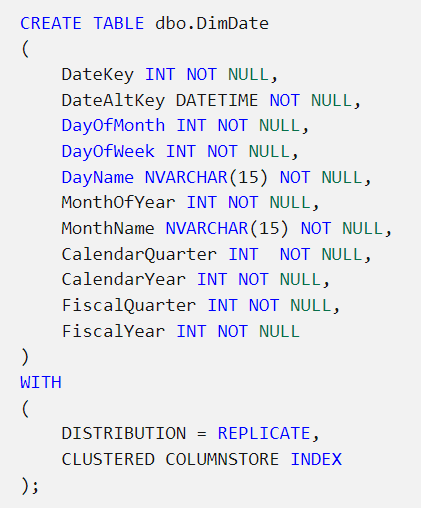
**Note**

If desired, you can create a specific *schema* as a namespace for your tables. In this example, the default **dbo** schema is used.

|  |  |
| --- | --- |
| If you intend to use a snowflake schema in which dimension tables are related to one another, you should include the key for the parent dimension in the definition of the child dimension table. For example, the following SQL code could be used to move the geographical address details from the **DimCustomer** table to a separate **DimGeography** dimension table: |  |

### Time dimension tables

Most data warehouses include a time dimension table that enables you to aggregate data by multiple hierarchical levels of time interval. For example, the following example creates a **DimDate** table with attributes that relate to specific dates.



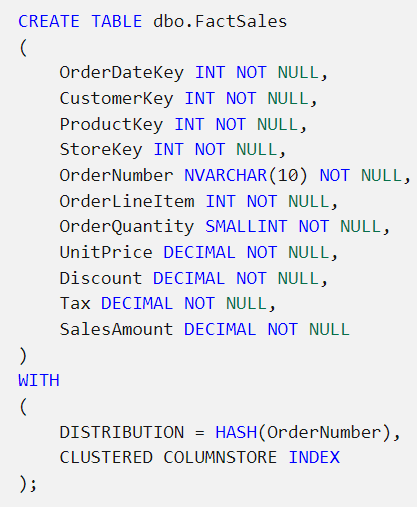
**Tip**

A common pattern when creating a dimension table for dates is to use the numeric date in *DDMMYYYY* or *YYYYMMDD* format as an integer surrogate key, and the date as a DATE or DATETIME datatype as the alternate key.

## **Creating fact tables**

Fact tables include the keys for each dimension to which they're related, and the attributes and numeric measures for specific events or observations that you want to analyze.

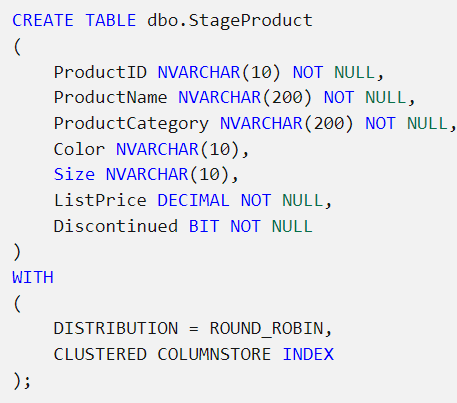
The following code example creates a hypothetical fact table named **FactSales** that is related to multiple dimensions through key columns (date, customer, product, and store)



## Creating staging tables

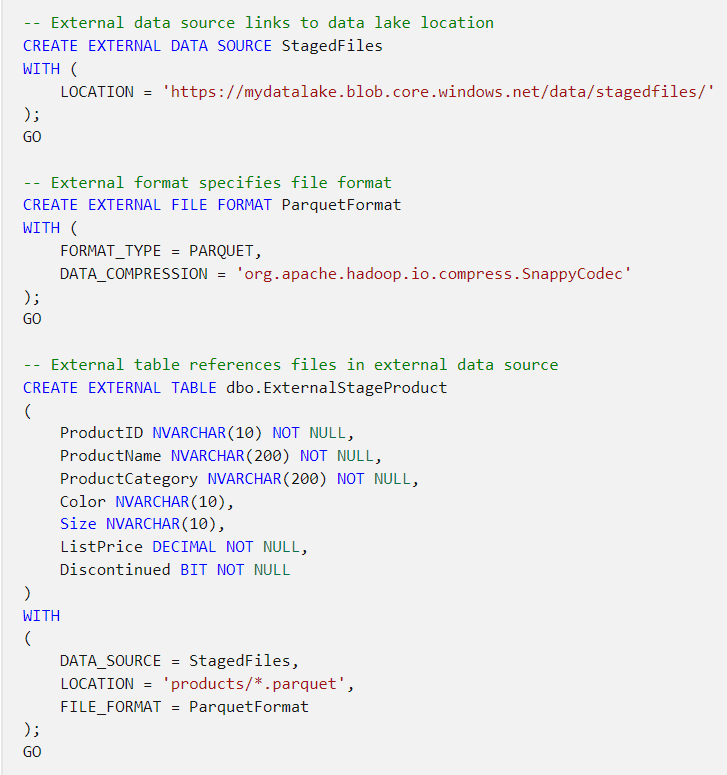
Staging tables are used as temporary storage for data as it's being loaded into the data warehouse. A typical pattern is to structure the table to make it as efficient as possible to ingest the data from its external source (often files in a data lake) into the relational database, and then use SQL statements to load the data from the staging tables into the dimension and fact tables.

The following code example creates a staging table for product data that will ultimately be loaded into a dimension table:



### **Using external tables**

In some cases, if the data to be loaded is in files with an appropriate structure, it can be more effective to create external tables that reference the file location. This way, the data can be read directly from the source files instead of being loaded into the relational store. The following example, shows how to create an external table that references files in the data lake associated with the Synapse workspace:

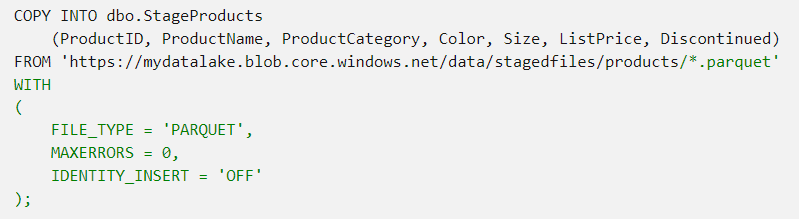


**Note**

For more information about using external tables, see [**Use external tables with Synapse SQL**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables) in the Azure Synapse Analytics documentation.

# Load data warehouse tables

At a basic level, loading a data warehouse is typically achieved by adding new data from files in a data lake into tables in the data warehouse. The COPY statement is an effective way to accomplish this task, as shown in the following example:



## **Considerations for designing a data warehouse load process**

One of the most common patterns for loading a data warehouse is to transfer data from source systems to files in a data lake, ingest the file data into staging tables, and then use SQL statements to load the data from the staging tables into the dimension and fact tables. Usually data loading is performed as a periodic batch process in which inserts and updates to the data warehouse are coordinated to occur at a regular interval (for example, daily, weekly, or monthly).

In most cases, you should implement a data warehouse load process that performs tasks in the following order:

1. Ingest the new data to be loaded into a data lake, applying pre-load cleansing or transformations as required.
2. Load the data from files into staging tables in the relational data warehouse.
3. Load the dimension tables from the dimension data in the staging tables, updating existing rows or inserting new rows and generating surrogate key values as necessary.
4. Load the fact tables from the fact data in the staging tables, looking up the appropriate surrogate keys for related dimensions.
5. Perform post-load optimization by updating indexes and table distribution statistics.

After using the COPY statement to load data into staging tables, you can use a combination of INSERT, UPDATE, MERGE, and CREATE TABLE AS SELECT (CTAS) statements to load the staged data into dimension and fact tables.

**Note**

Implementing an effective data warehouse loading solution requires careful consideration of how to manage surrogate keys, slowly changing dimensions, and other complexities inherent in a relational data warehouse schema. To learn more about techniques for loading a data warehouse, consider completing the [**Load data into a relational data warehouse**](https://learn.microsoft.com/en-us/training/modules/load-optimize-data-into-relational-data-warehouse) module.

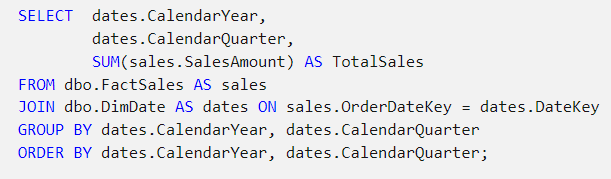
# Query a data warehouse

When the dimension and fact tables in a data warehouse have been loaded with data, you can use SQL to query the tables and analyze the data they contain. The Transact-SQL syntax used to query tables in a Synapse dedicated SQL pool is similar to SQL used in SQL Server or Azure SQL Database.

## **Aggregating measures by dimension attributes**

Most data analytics with a data warehouse involves aggregating numeric measures in fact tables by attributes in dimension tables. Because of the way a star or snowflake schema is implemented, queries to perform this kind of aggregation rely on JOIN clauses to connect fact tables to dimension tables, and a combination of aggregate functions and GROUP BY clauses to define the aggregation hierarchies.

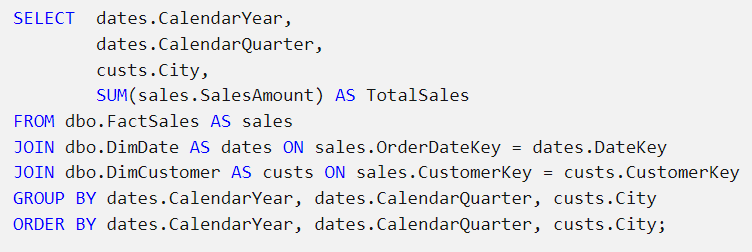
For example, the following SQL queries the **FactSales** and **DimDate** tables in a hypothetical data warehouse to aggregate sales amounts by year and quarter:



The results from this query would look similar to the following table:

| **CalendarYear** | **CalendarMonth** | **TotalSales** |
| --- | --- | --- |
| 2020 | 1 | 25980.16 |
| 2020 | 2 | 27453.87 |
| 2020 | 3 | 28527.15 |
| 2020 | 4 | 31083.45 |
| 2021 | 1 | 34562.96 |
| 2021 | 2 | 36162.27 |

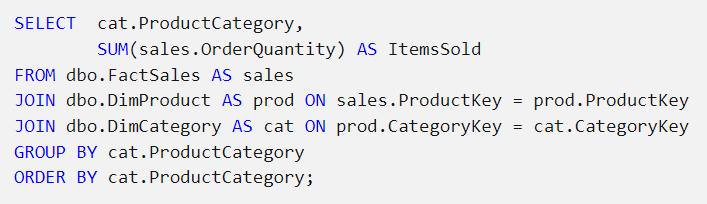
You can join as many dimension tables as needed to calculate the aggregations you need. For example, the following code extends the previous example to break down the quarterly sales totals by city based on the customer's address details in the **DimCustomer** table:



| **CalendarYear** | **CalendarMonth** | **City** | **TotalSales** |
| --- | --- | --- | --- |
| 2020 | 1 | Amsterdam | 5982.53 |
| 2020 | 1 | Berlin | 2826.98 |
| 2020 | 1 | Chicago | 5372.72 |
| ... | ... | ... | .. |
| 2020 | 2 | Amsterdam | 7163.93 |
| 2020 | 2 | Berlin | 8191.12 |
| 2020 | 2 | Chicago | 2428.72 |
| ... | ... | ... | .. |
| 2020 | 3 | Amsterdam | 7261.92 |
| 2020 | 3 | Berlin | 4202.65 |
| 2020 | 3 | Chicago | 2287.87 |
| ... | ... | ... | .. |
| 2020 | 4 | Amsterdam | 8262.73 |
| 2020 | 4 | Berlin | 5373.61 |
| 2020 | 4 | Chicago | 7726.23 |
| ... | ... | ... | .. |
| 2021 | 1 | Amsterdam | 7261.28 |
| 2021 | 1 | Berlin | 3648.28 |
| 2021 | 1 | Chicago | 1027.27 |
| ... | ... | ... | .. |

### **Joins in a snowflake schema**

When using a snowflake schema, dimensions may be partially normalized; requiring multiple joins to relate fact tables to snowflake dimensions. For example, suppose your data warehouse includes a **DimProduct** dimension table from which the product categories have been normalized into a separate **DimCategory** table. A query to aggregate items sold by product category might look similar to the following example:



The results from this query include the number of items sold for each product category:

| **ProductCategory** | **ItemsSold** |
| --- | --- |
| Accessories | 28271 |
| Bits and pieces | 5368 |
| ... | ... |

**Note**

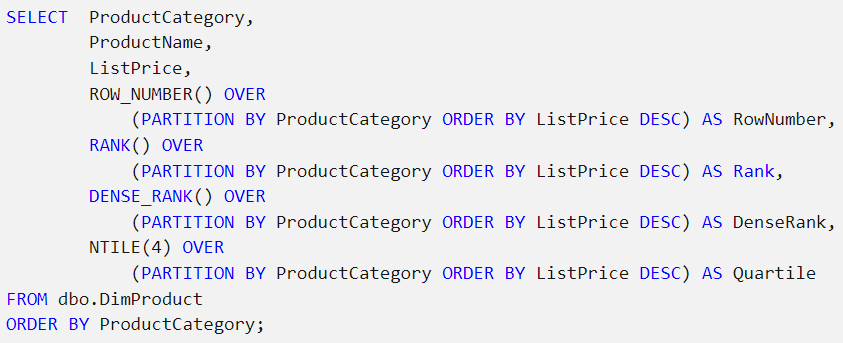
JOIN clauses for **FactSales** and **DimProduct** and for **DimProduct** and **DimCategory** are both required, even though no fields from **DimProduct** are returned by the query.

## Using ranking functions

Another common kind of analytical query is to partition the results based on a dimension attribute and rank the results within each partition. For example, you might want to rank stores each year by their sales revenue. To accomplish this goal, you can use Transact-SQL ranking functions such as ROW\_NUMBER, RANK, DENSE\_RANK, and NTILE. These functions enable you to partition the data over categories, each returning a specific value that indicates the relative position of each row within the partition:

* **ROW\_NUMBER** returns the ordinal position of the row within the partition. For example, the first row is numbered 1, the second 2, and so on.
* **RANK** returns the ranked position of each row in the ordered results. For example, in a partition of stores ordered by sales volume, the store with the highest sales volume is ranked 1. If multiple stores have the same sales volumes, they'll be ranked the same, and the rank assigned to subsequent stores reflects the number of stores that have higher sales volumes - including ties.
* **DENSE\_RANK** ranks rows in a partition the same way as **RANK**, but when multiple rows have the same rank, subsequent rows are ranking positions ignore ties.
* **NTILE** returns the specified percentile in which the row falls. For example, in a partition of stores ordered by sales volume, NTILE(4) returns the quartile in which a store's sales volume places it.

For example, consider the following query:



The query partitions products into groupings based on their categories, and within each category partition, the relative position of each product is determined based on its list price. The results from this query might look similar to the following table:

| **ProductCategory** | **ProductName** | **ListPrice** | **RowNumber** | **Rank** | **DenseRank** | **Quartile** |
| --- | --- | --- | --- | --- | --- | --- |
| Accessories | Widget | 8.99 | 1 | 1 | 1 | 1 |
| Accessories | Knicknak | 8.49 | 2 | 2 | 2 | 1 |
| Accessories | Sprocket | 5.99 | 3 | 3 | 3 | 2 |
| Accessories | Doodah | 5.99 | 4 | 3 | 3 | 2 |
| Accessories | Spangle | 2.99 | 5 | 5 | 4 | 3 |
| Accessories | Badabing | 0.25 | 6 | 6 | 5 | 4 |
| Bits and pieces | Flimflam | 7.49 | 1 | 1 | 1 | 1 |
| Bits and pieces | Snickity wotsit | 6.99 | 2 | 2 | 2 | 1 |
| Bits and pieces | Flange | 4.25 | 3 | 3 | 3 | 2 |
| ... | ... | ... | ... | ... | ... | ... |

**Note**

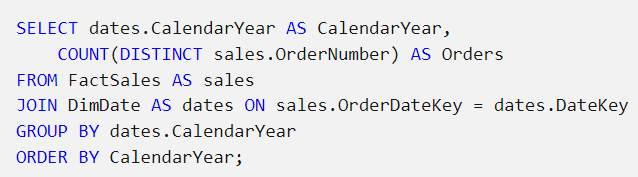
The sample results demonstrate the difference between RANK and DENSE\_RANK. Note that in the Accessories category, the Sprocket and Doodah products have the same list price; and are both ranked as the 3rd highest priced product. The next highest priced product has a RANK of 5 (there are four products more expensive than it) and a DENSE\_RANK of 4 (there are three higher prices).

To learn more about ranking functions, see [**Ranking Functions (Transact-SQL)**](https://learn.microsoft.com/en-us/sql/t-sql/functions/ranking-functions-transact-sql?view=azure-sqldw-latest) in the Azure Synapse Analytics documentation.

## **Retrieving an approximate count**

While the purpose of a data warehouse is primarily to support analytical data models and reports for the enterprise; data analysts and data scientists often need to perform some initial data exploration, just to determine the basic scale and distribution of the data.

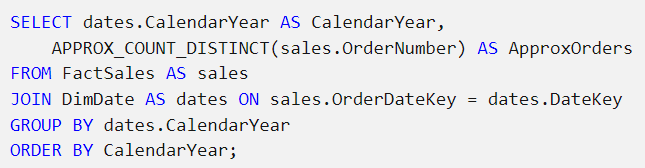
For example, the following query uses the COUNT function to retrieve the number of sales for each year in a hypothetical data warehouse:



The results of this query might look similar to the following table:

| **CalendarYear** | **Orders** |
| --- | --- |
| 2019 | 239870 |
| 2020 | 284741 |
| 2021 | 309272 |
| ... | ... |

The volume of data in a data warehouse can mean that even simple queries to count the number of records that meet specified criteria can take a considerable time to run. In many cases, a precise count isn't required - an approximate estimate will suffice. In such cases, you can use the APPROX\_COUNT\_DISTINCT function as shown in the following example:



The APPROX\_COUNT\_DISTINCT function uses a HyperLogLog algorithm to retrieve an approximate count. The result is guaranteed to have a maximum error rate of 2% with 97% probability, so the results of this query with the same hypothetical data as before might look similar to the following table:

| **CalendarYear** | **ApproxOrders** |
| --- | --- |
| 2019 | 235552 |
| 2020 | 290436 |
| 2021 | 304633 |
| ... | ... |

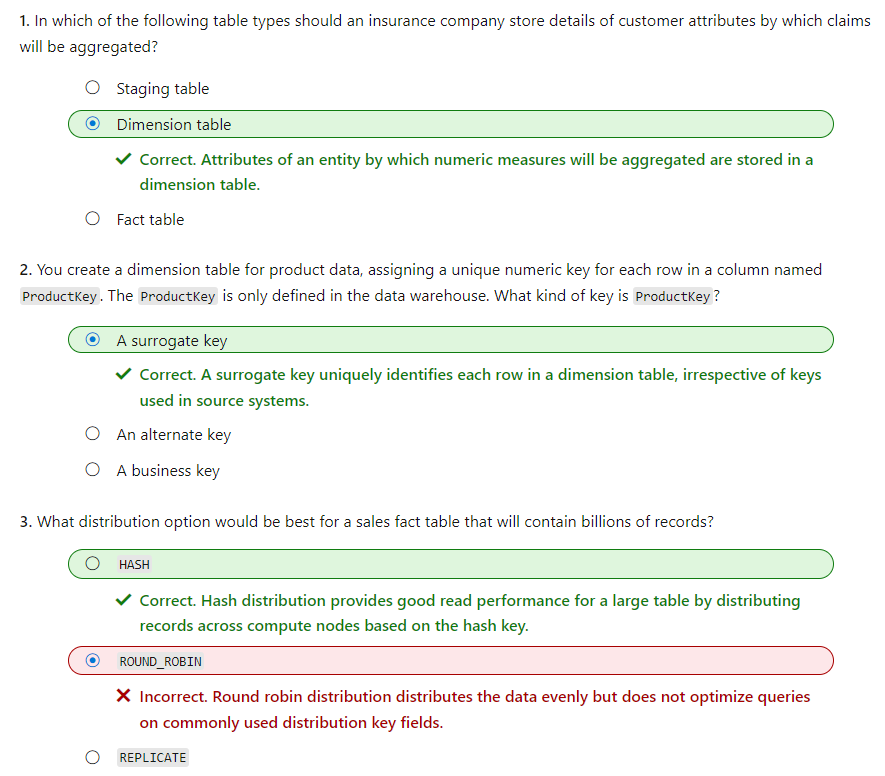
The counts are less accurate, but still sufficient for an approximate comparison of yearly sales. With a large volume of data, the query using the APPROX\_COUNT\_DISTINCT function completes more quickly, and the reduced accuracy may be an acceptable trade-off during basic data exploration.

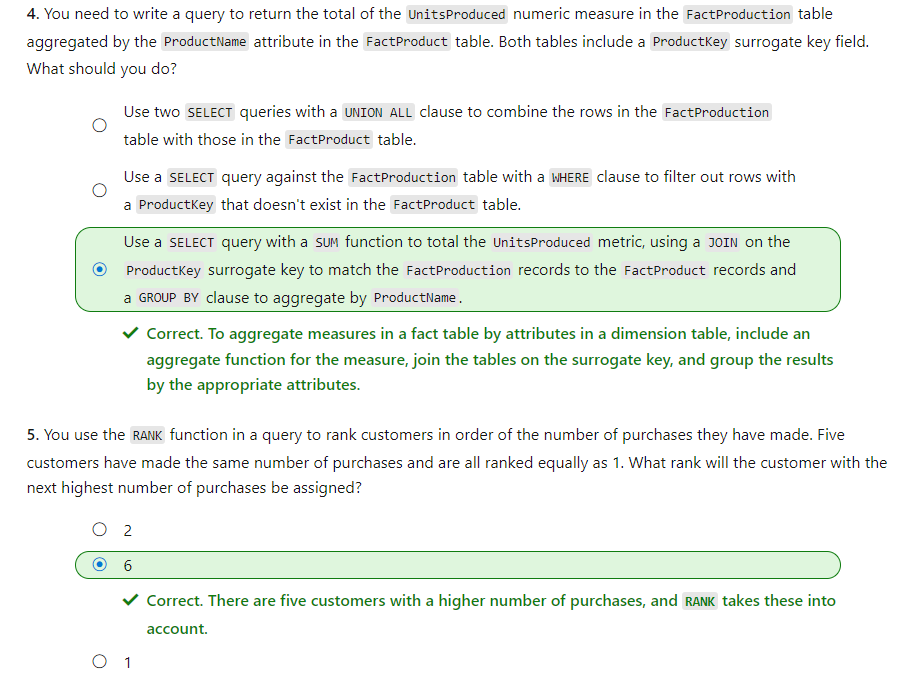
**Note**

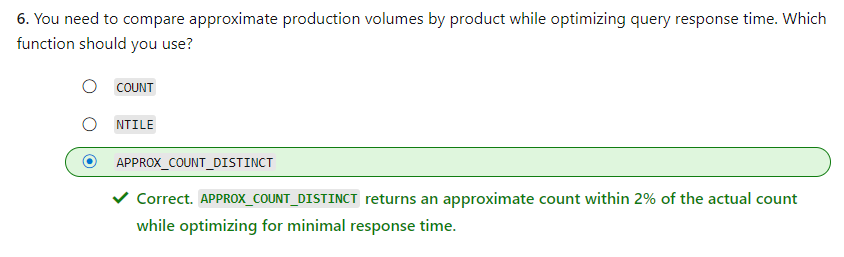
See the [**APPROX\_COUNT\_DISTINCT**](https://learn.microsoft.com/en-us/sql/t-sql/functions/approx-count-distinct-transact-sql?view=azure-sqldw-latest) function documentation for more details.

Exercises:

<https://microsoftlearning.github.io/DP-500-Azure-Data-Analyst/Instructions/labs/03-Explore-data-warehouse.html>







# Load data into a relational data warehouse

Many enterprise analytical solutions include a relational data warehouse. Data engineers are responsible for implementing ingestion solutions that load data into the data warehouse tables, usually on a regular schedule.

As a data engineer, you need to be familiar with the considerations and techniques that apply to loading a data warehouse. In this module, we'll focus on ways that you can use SQL to load data into tables in a dedicated SQL pool in Azure Synapse Analytics.

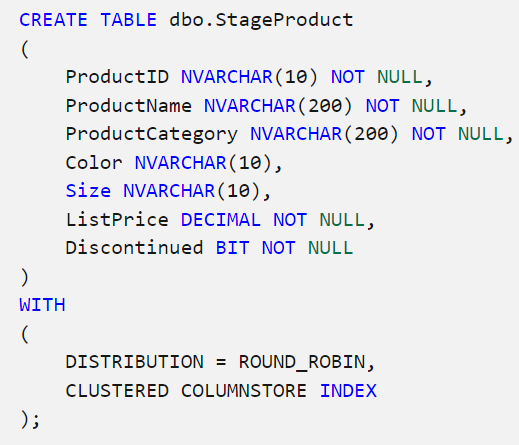
# Load staging tables

One of the most common patterns for loading a data warehouse is to transfer data from source systems to files in a data lake, ingest the file data into staging tables, and then use SQL statements to load the data from the staging tables into the dimension and fact tables. Usually data loading is performed as a periodic batch process in which inserts and updates to the data warehouse are coordinated to occur at a regular interval (for example, daily, weekly, or monthly).

## **Creating staging tables**

Many organized warehouses have standard structures for staging the database and may even use a specific schema for staging the data. The following code example creates a staging table for product data that will ultimately be loaded into a dimension table:

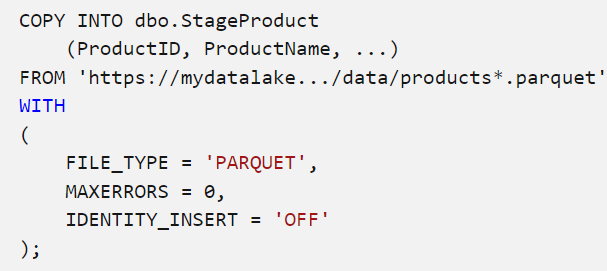
**Note:** This example creates a staging table in the default **dbo** schema. You can also create separate schemas for staging tables with a meaningful name, such as **stage** so architects and users understand the purpose of the schema.



## **Using the COPY command**

You can use the COPY statement to load data from the data lake, as shown in the following example:

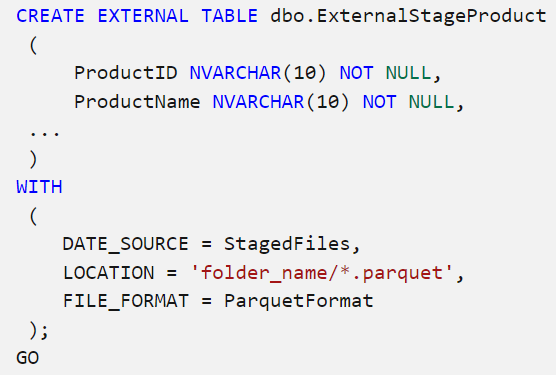
**Note:** This is generally the recommended approach to load staging tables due to its high performance throughput.



**Tip:** To learn more about the COPY statement, see [**COPY (Transact-SQL)**](https://learn.microsoft.com/en-us/sql/t-sql/statements/copy-into-transact-sql) in the Transact-SQL documentation.

## **Using external tables**

In some cases, if the data to be loaded is stored in files with an appropriate structure, it can be more effective to create external tables that reference the file location. This way, the data can be read directly from the source files instead of being loaded into the relational store. The following example, shows how to create an external table that references files in the data lake associated with the Azure Synapse Analytics workspace:



**Tip**

For more information about using external tables, see [**Use external tables with Synapse SQL**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables) in the Azure Synapse Analytics documentation.

# Load dimension tables

After staging dimension data, you can load it into dimension tables using SQL.

## **Using a CREATE TABLE AS (CTAS) statement**

|  |  |
| --- | --- |
| One of the simplest ways to load data into a new dimension table is to use a CREATE TABLE AS (CTAS) expression. This statement creates a new table based on the results of a SELECT statement. |  |

**Note**

You can't use IDENTITY to generate a unique integer value for the surrogate key when using a CTAS statement, so this example uses the ROW\_NUMBER function to generate an incrementing row number for each row in the results ordered by the **ProductID** business key in the staged data.

|  |  |
| --- | --- |
| You can also load a combination of new and updated data into a dimension table by using a CREATE TABLE AS (CTAS) statement to create a new table that UNIONs the existing rows from the dimension table with the new and updated records from the staging table. After creating the new table, you can delete or rename the current dimension table, and rename the new table to replace it. |  |

While this technique is effective in merging new and existing dimension data, lack of support for IDENTITY columns means that it's difficult to generate a surrogate key.

**Tip**

For more information, see [**CREATE TABLE AS SELECT (CTAS)**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-develop-ctas) in the Azure Synapse Analytics documentation.

## **Using an INSERT statement**

|  |  |
| --- | --- |
| When you need to load staged data into an existing dimension table, you can use an INSERT statement. This approach works if the staged data contains only records for new dimension entities (not updates to existing entities). This approach is much less complicated than the technique in the last section, which required a UNION ALL and then renaming table objects. |  |

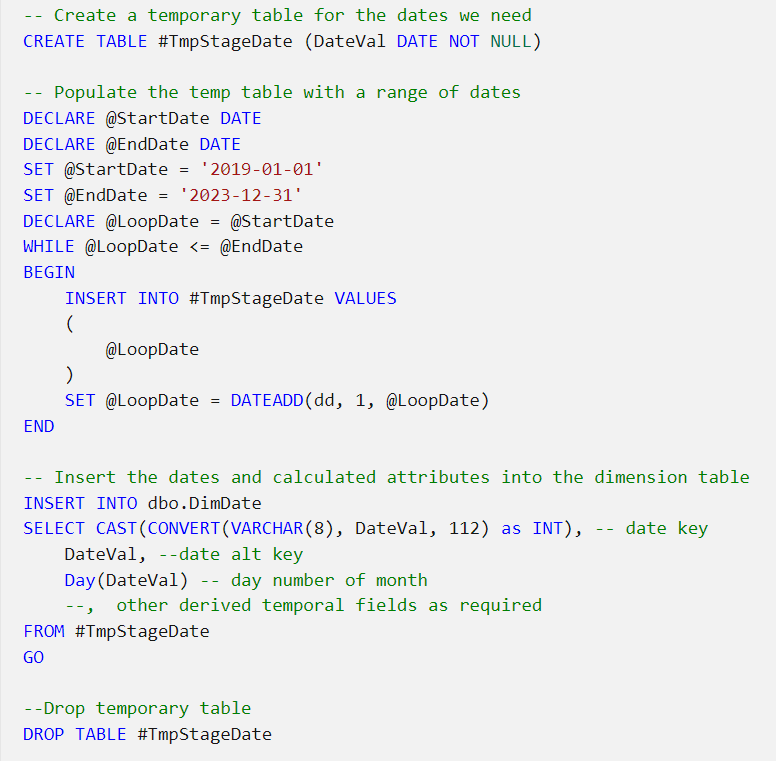
**Note**

Assuming the **DimCustomer** dimension table is defined with an IDENTITY **CustomerKey** column for the surrogate key (as described in the previous unit), the key will be generated automatically and the remaining columns will be populated using the values retrieved from the staging table by the SELECT query.

# Load time dimension tables

Time dimension tables store a record for each time interval based on the *grain* by which you want to aggregate data over time. For example, a time dimension table at the *date* grain contains a record for each date between the earliest and latest dates referenced by the data in related fact tables.

The following code example shows how you can generate a sequence of time dimension values based on a *date* grain.



**Tip**

Scripting this in SQL may be time-consuming in a dedicated SQL pool – it may be more efficient to prepare the data in Microsoft Excel or an external script and import it using the COPY statement.

As the data warehouse is populated in the future with new fact data, you periodically need to extend the range of dates related time dimension tables.

# Load slowly changing dimensions

In most relational data warehouses, you need to handle updates to dimension data and support what are commonly referred to as slowly changing dimensions (SCDs).

## Types of slowly changing dimension

There are multiple kinds of slowly changing dimension, of which three are commonly implemented:

### **Type 0**

Type 0 dimension data can't be changed. Any attempted changes fail.

| **DateKey** | **DateAltKey** | **Day** | **Month** | **Year** |
| --- | --- | --- | --- | --- |
| 20230101 | 01-01-2023 | Sunday | January | 2023 |

### **Type 1**

In type 1 dimensions, the dimension record is updated in-place. Changes made to an existing dimension row apply to all previously loaded facts related to the dimension.

| **StoreKey** | **StoreAltKey** | **StoreName** |
| --- | --- | --- |
| 123 | EH199J | Town Central Store |

### **Type 2**

In a type 2 dimension, a change to a dimension results in a new dimension row. Existing rows for previous versions of the dimension are retained for historical fact analysis and the new row is applied to future fact table entries.

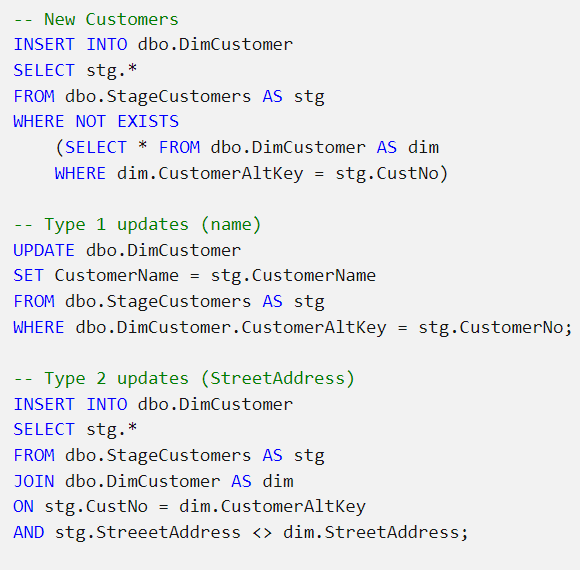
| **CustomerKey** | **CustomerAltKey** | **Name** | **Address** | **City** | **DateFrom** | **DateTo** | **IsCurrent** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1211 | jo@contoso.com | Jo Smith | 999 Main St | Seattle | 20190101 | 20230105 | False |
| 2996 | jo@contoso.com | Jo Smith | 1234 9th Ave | Boston | 20230106 |  | True |

**Note:** Type 2 dimensions often include columns to track the effective time periods for each version of an entity, and/or a flag to indicate which row represents the current version of the entity. If you’re using an incrementing surrogate key and you only need to track the most recently added version of an entity, then you may not need these columns; but before making that decision, consider how you’ll look up the appropriate version of an entity when a new fact is entered based on the time at which the event the fact relates to occurred.

**NOTE: SCD type is a target table concept. Incremental loading is a source side concept. You can do SCD 1 / 2 / 3 with either full or incremental load.**

## Combining INSERT and UPDATE statements

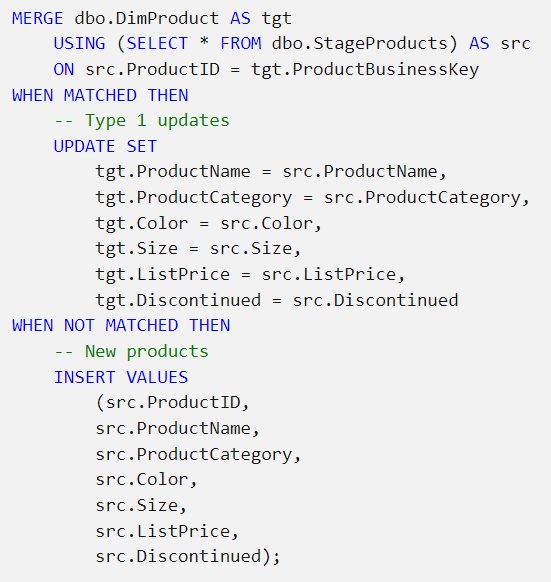
Logic to implement Type 1 and Type 2 updates can be complex, and there are various techniques you can use. For example, you could use a combination of UPDATE and INSERT statements.



In the previous example, it's assumed that an incrementing surrogate key based on an IDENTITY column identifies each row, and that the highest value surrogate key for a given alternate key indicates the most recent or "current" instance of the dimension entity associated with that alternate key. In practice, many data warehouse designers include a Boolean column to indicate the current active instance of a changing dimension or use DateTime fields to indicate the active time periods for each version of the dimension instance. With these approaches, the logic for a type 2 change must include an INSERT of the new dimension row and an UPDATE to mark the current row as inactive.

## **Using a MERGE statement**

As an alternative to using multiple INSERT and UPDATE statements, you can use a single MERGE statement to perform an "upsert" operation to insert new records and update existing ones.



**Note**

For more information about the MERGE statement, see the [**MERGE documentation for Azure Synapse Analytics**](https://learn.microsoft.com/en-us/sql/t-sql/statements/merge-transact-sql?view=azure-sqldw-latest&preserve-view=true).

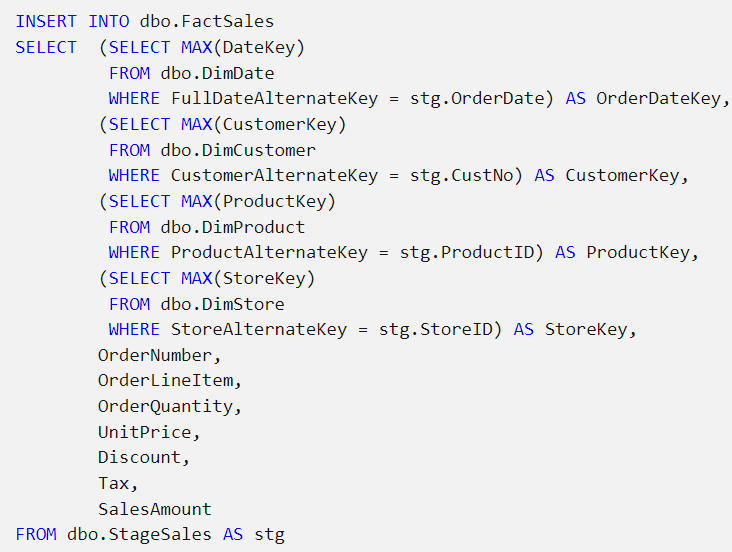
# Load fact tables

Typically, a regular data warehouse load operation loads fact tables after dimension tables. This approach ensures that the dimensions to which the facts will be related are already present in the data warehouse.

The staged fact data usually includes the business (alternate) keys for the related dimensions, so your logic to load the data must look up the corresponding surrogate keys. When the data warehouse slowly changing dimensions, the appropriate version of the dimension record must be identified to ensure the correct surrogate key is used to match the event recorded in the fact table with the state of the dimension at the time the fact occurred.

In many cases, you can retrieve the latest "current" version of the dimension; but in some cases you might need to find the right dimension record based on DateTime columns that indicate the period of validity for each version of the dimension.

The following example assumes that the dimension records have an incrementing surrogate key, and that the most recently added version of a specific dimension instance (which will have the highest key value) should be used.



# Perform post load optimization

After loading new data into the data warehouse, it's a good idea to rebuild the table indexes and update statistics on commonly queried columns.

## **Rebuild indexes**

The following example rebuilds all indexes on the **DimProduct** table.

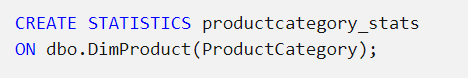


**Tip**

For more information about rebuilding indexes, see the [**Indexes on dedicated SQL pool tables in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-tables-index) article in the Azure Synapse Analytics documentation.

## **Update statistics**

The following example creates statistics on the **ProductCategory** column of the **DimProduct** table:

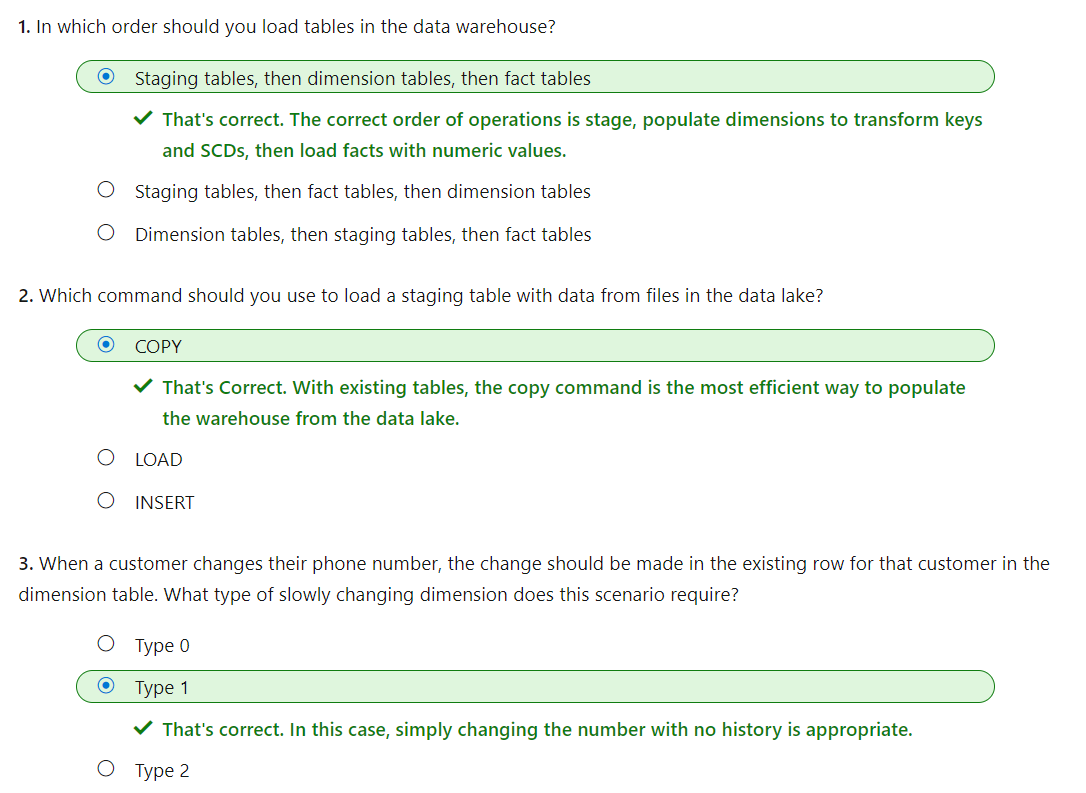


**Tip**

For more information about updating statistics, see the [**Table statistics for dedicated SQL pool in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-tables-statistics) article in the Azure Synapse Analytics documentation.

# Exercise - load data into a relational data warehouse

<https://microsoftlearning.github.io/dp-203-azure-data-engineer/Instructions/Labs/09-Load-Data-into-Data-Warehouse.html>



Loading data warehouse tables is a core task for data engineers. This module introduced some common SQL-based techniques that you can use to stage and load data in a relational data warehouse that's hosted in a dedicated SQL pool in Azure Synapse Analytics.