Cloud Academy Required Reading

Identify data formats

Completed100 XP

- 5 minutes

Data is a collection of facts such as numbers, descriptions, and observations used to record information. Data structures in which this data is organized often represents *entities* that are important to an organization (such as customers, products, sales orders, and so on). Each entity typically has one or more *attributes*, or characteristics (for example, a customer might have a name, an address, a phone number, and so on).

You can classify data as *structured*, *semi-structured*, or *unstructured*.

Structured data

Structured data is data that adheres to a fixed *schema*, so all of the data has the same fields or properties. Most commonly, the schema for structured data entities is *tabular* - in other words, the data is represented in one or more tables that consist of rows to represent each instance of a data entity, and columns to represent attributes of the entity. For example, the following image shows tabular data representations for *Customer* and *Product* entities.

Structured data is often stored in a database in which multiple tables can reference one another by using key values in a *relational* model; which we'll explore in more depth later.

Semi-structured data

Semi-structured data is information that has some structure, but which allows for some variation between entity instances. For example, while most customers may have an email address, some might have multiple email addresses, and some might have none at all.

One common format for semi-structured data is *JavaScript Object Notation* (JSON). The example below shows a pair of JSON documents that represent customer information. Each customer document includes address and contact information, but the specific fields vary between customers.

JSONCopy

```
// Customer 1
{
  "firstName": "Joe",
  "lastName": "Jones",
  "address":
  {
    "streetAddress": "1 Main St.",
    "city": "New York",
    "state": "NY",
    "postalCode": "10099"
  },
  "contact":
  [
    {
      "type": "home",
      "number": "555 123-1234"
    },
    {
      "type": "email",
      "address": "joe@litware.com"
    }
  ]
}

// Customer 2
{
  "firstName": "Samir",
  "lastName": "Nadoy",
  "address":
  {
    "streetAddress": "123 Elm Pl.",
    "unit": "500",
    "city": "Seattle",
    "state": "WA",
    "postalCode": "98999"
  },
  "contact":
  [
    {
      "type": "email",
      "address": "samir@northwind.com"
    }
  ]
}
```

Note

JSON is just one of many ways in which semi-structured data can be represented. The point here is not to provide a detailed examination of JSON syntax, but rather to illustrate the flexible nature of semi-structured data representations.

Unstructured data

Not all data is structured or even semi-structured. For example, documents, images, audio and video data, and binary files might not have a specific structure. This kind of data is referred to as *unstructured* data.

Data stores

Organizations typically store data in structured, semi-structured, or unstructured format to record details of entities (for example, customers and products), specific events (such as sales transactions), or other information in documents, images, and other formats. The stored data can then be retrieved for analysis and reporting later.

There are two broad categories of data store in common use:

- File stores
- Databases

We'll explore both of these types of data store in subsequent topics.

Explore file storage

Completed 100 XP

- 5 minutes

The ability to store data in files is a core element of any computing system. Files can be stored in local file systems on the hard disk of your personal computer, and on removable media such as USB drives; but in most organizations, important data files are stored centrally in some kind of shared file storage system. Increasingly, that central storage location is hosted in the cloud, enabling cost-effective, secure, and reliable storage for large volumes of data.

The specific file format used to store data depends on a number of factors, including:

- The type of data being stored (structured, semi-structured, or unstructured).
- The applications and services that will need to read, write, and process the data.
- The need for the data files to be readable by humans, or optimized for efficient storage and processing.

Some common file formats are discussed below.

Delimited text files

Data is often stored in plain text format with specific field delimiters and row terminators. The most common format for delimited data is comma-separated values (CSV) in which fields are separated by commas, and rows are terminated by a carriage return / new line. Optionally, the first line may include the field names. Other common formats include tab-separated values (TSV) and space-delimited (in which tabs or spaces are used to separate fields), and fixed-width data in which each field is allocated a fixed number of characters. Delimited text is a good choice for structured data that needs to be accessed by a wide range of applications and services in a human-readable format.

The following example shows customer data in comma-delimited format:

Copy
```
FirstName,LastName,Email
Joe,Jones,joe@litware.com
Samir,Nadoy,samir@northwind.com
```

JavaScript Object Notation (JSON)

JSON is a ubiquitous format in which a hierarchical document schema is used to define data entities (objects) that have multiple attributes. Each attribute might be an object (or a collection of objects); making JSON a flexible format that's good for both structured and semi-structured data.

The following example shows a JSON document containing a collection of customers. Each customer has three attributes (*firstName*, *lastName*, and *contact*), and the *contact* attribute contains a collection of objects that represent one or more contact methods (email or phone). Note that objects are enclosed in braces (**{..}**) and collections are enclosed in square brackets (**[..]**). Attributes are represented by *name* **:** *value* pairs and separated by commas (**,**).

JSONCopy

```json
{
  "customers":
  [
    {
      "firstName": "Joe",
      "lastName": "Jones",
      "contact":
      [
        {
          "type": "home",
          "number": "555 123-1234"
        },
        {
          "type": "email",
          "address": "joe@litware.com"
        }
      ]
    },
    {
      "firstName": "Samir",
      "lastName": "Nadoy",
      "contact":
      [
        {
          "type": "email",
          "address": "samir@northwind.com"
        }
      ]
    }
  ]
}
```

Extensible Markup Language (XML)

XML is a human-readable data format that was popular in the 1990s and 2000s. It's largely been superseded by the less verbose JSON format, but there are still some systems that use XML to represent data. XML uses *tags* enclosed in angle-brackets (**<../>**) to define *elements* and *attributes*, as shown in this example:

XMLCopy
```xml
<Customers>
  <Customer name="Joe" lastName="Jones">
    <ContactDetails>
      <Contact type="home" number="555 123-1234"/>
      <Contact type="email" address="joe@litware.com"/>
    </ContactDetails>
  </Customer>
  <Customer name="Samir" lastName="Nadoy">
    <ContactDetails>
      <Contact type="email" address="samir@northwind.com"/>
    </ContactDetails>
  </Customer>
</Customers>
```

Binary Large Object (BLOB)

Ultimately, all files are stored as binary data (1's and 0's), but in the human-readable formats discussed above, the bytes of binary data are mapped to printable characters (typically through a character encoding scheme such as ASCII or Unicode). Some file formats however, particularly for unstructured data, store the data as raw binary that must be interpreted by applications and rendered. Common types of data stored as binary include images, video, audio, and application-specific documents.

When working with data like this, data professionals often refer to the data files as *BLOBs* (Binary Large Objects).

Optimized file formats

While human-readable formats for structured and semi-structured data can be useful, they're typically not optimized for storage space or processing. Over time, some specialized file formats that enable compression, indexing, and efficient storage and processing have been developed.

Some common optimized file formats you might see include *Avro*, *ORC*, and *Parquet*:

- *Avro* is a row-based format. It was created by Apache. Each record contains a header that describes the structure of the data in the record. This header is stored as JSON. The data is stored as binary information. An application uses the information in the header to parse the binary data and extract the fields it contains. Avro is a good format for compressing data and minimizing storage and network bandwidth requirements.

- *ORC* (Optimized Row Columnar format) organizes data into columns rather than rows. It was developed by HortonWorks for optimizing read and write operations in Apache Hive (Hive is a data warehouse system that supports fast data summarization and querying over large datasets). An ORC file contains *stripes* of data. Each stripe holds the data for a column or set of columns. A stripe contains an index into the rows in the stripe, the data for each row, and a footer that holds statistical information (count, sum, max, min, and so on) for each column.
- *Parquet* is another columnar data format. It was created by Cloudera and Twitter. A Parquet file contains row groups. Data for each column is stored together in the same row group. Each row group contains one or more chunks of data. A Parquet file includes metadata that describes the set of rows found in each chunk. An application can use this metadata to quickly locate the correct chunk for a given set of rows, and retrieve the data in the specified columns for these rows. Parquet specializes in storing and processing nested data types efficiently. It supports very efficient compression and encoding schemes.

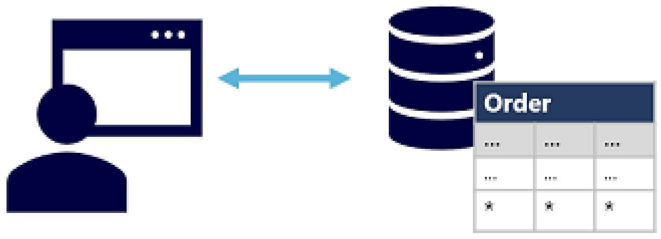
Explore transactional data processing

Completed100 XP

- 5 minutes

A transactional data processing system is what most people consider the primary function of business computing. A transactional system records *transactions* that encapsulate specific events that the organization wants to track. A transaction could be financial, such as the movement of money between accounts in a banking system, or it might be part of a retail system, tracking payments for goods and services from customers. Think of a transaction as a small, discrete, unit of work.

Transactional systems are often high-volume, sometimes handling many millions of transactions in a single day. The data being processed has to be accessible very quickly. The work performed by transactional systems is often referred to as Online Transactional Processing (OLTP).



OLTP solutions rely on a database system in which data storage is optimized for both read and write operations in order to support transactional workloads in which data records are created, retrieved, updated, and deleted (often referred to as *CRUD* operations). These operations are applied transactionally, in a way that ensures the integrity of the data stored in the database. To accomplish this, OLTP systems enforce transactions that support so-called ACID semantics:

- **Atomicity** – each transaction is treated as a single unit, which succeeds completely or fails completely. For example, a transaction that involved debiting funds from one account and crediting the same amount to another account must complete both actions. If either action can't be completed, then the other action must fail.
- **Consistency** – transactions can only take the data in the database from one valid state to another. To continue the debit and credit example above, the completed state of the transaction must reflect the transfer of funds from one account to the other.
- **Isolation** – concurrent transactions cannot interfere with one another, and must result in a consistent database state. For example, while the transaction to transfer funds from one account to another is in-process, another transaction that checks the balance of these accounts must return consistent results - the balance-checking transaction can't retrieve a value for one account that reflects the balance *before* the transfer, and a value for the other account that reflects the balance *after* the transfer.
- **Durability** – when a transaction has been committed, it will remain committed. After the account transfer transaction has completed, the revised account balances are persisted so that even if the database system were to be switched off, the committed transaction would be reflected when it is switched on again.

OLTP systems are typically used to support live applications that process business data - often referred to as *line of business* (LOB) applications.

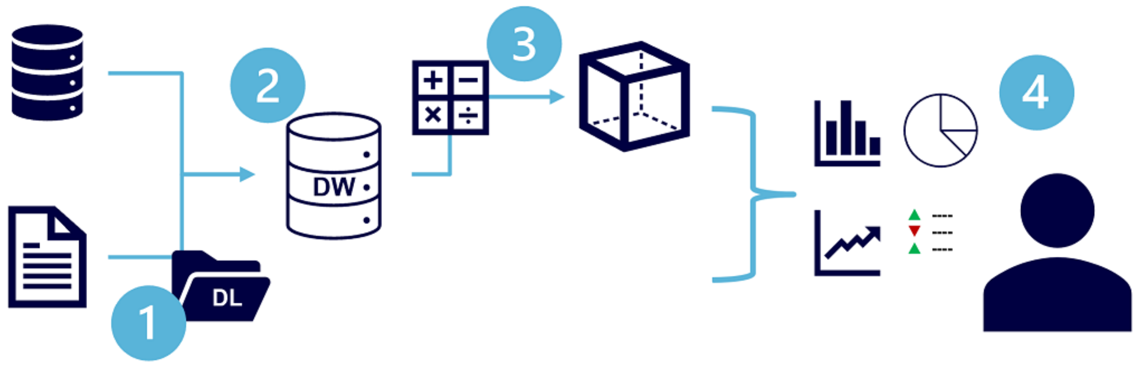
Explore analytical data processing

Completed 100 XP

- 5 minutes

Analytical data processing typically uses read-only (or read-*mostly*) systems that store vast volumes of historical data or business metrics. Analytics can be based on a snapshot of the data at a given point in time, or a series of snapshots.

The specific details for an analytical processing system can vary between solutions, but a common architecture for enterprise-scale analytics looks like this:



1. Data files may be stored in a central data lake for analysis.
2. An extract, transform, and load (ETL) process copies data from files and OLTP databases into a data warehouse that is optimized for read activity. Commonly, a data warehouse schema is based on *fact* tables that contain numeric values you want to analyze (for example, sales amounts), with related *dimension* tables that represent the entities by which you want to measure them (for example, customer or product),
3. Data in the data warehouse may be aggregated and loaded into an online analytical processing (OLAP) model, or *cube*. Aggregated numeric values (*measures*) from fact tables are calculated for intersections of *dimensions* from dimension tables. For example, sales revenue might be totaled by date, customer, and product.
4. The data in the data lake, data warehouse, and analytical model can be queried to produce reports, visualizations, and dashboards.

Data lakes are common in large-scale data analytical processing scenarios, where a large volume of file-based data must be collected and analyzed.

Data warehouses are an established way to store data in a relational schema that is optimized for read operations – primarily queries to support reporting and data visualization. The data warehouse schema may require some denormalization of data in an OLTP data source (introducing some duplication to make queries perform faster).

An OLAP model is an aggregated type of data storage that is optimized for analytical workloads. Data aggregations are across dimensions at different levels, enabling you to *drill up/down* to view aggregations at multiple hierarchical levels; for example to find total sales by region, by city, or for an individual address. Because OLAP data is pre-aggregated, queries to return the summaries it contains can be run quickly.

Different types of user might perform data analytical work at different stages of the overall architecture. For example:

- Data scientists might work directly with data files in a data lake to explore and model data.
- Data Analysts might query tables directly in the data warehouse to produce complex reports and visualizations.
- Business users might consume pre-aggregated data in an analytical model in the form of reports or dashboards.

Explore job roles in the world of data

Completed100 XP

- 5 minutes

There's a wide variety of roles involved in managing, controlling, and using data. Some roles are business-oriented, some involve more engineering, some focus on research, and some are hybrid roles that combine different aspects of data management. Your organization may define roles differently, or give them different names, but the roles described in this unit encapsulate the most common division of tasks and responsibilities.

The three key job roles that deal with data in most organizations are:

- **Database administrators** manage databases, assigning permissions to users, storing backup copies of data and restore data in the event of a failure.
- **Data engineers** manage infrastructure and processes for data integration across the organization, applying data cleaning routines, identifying data

governance rules, and implementing pipelines to transfer and transform data between systems.
- **Data analysts** explore and analyze data to create visualizations and charts that enable organizations to make informed decisions.

Note

The job *roles* define differentiated tasks and responsibilities. In some organizations, the same *person* might perform multiple roles; so in their role as database administrator they might provision a transactional database, and then in their role as a data engineer they might create a pipeline to transfer data from the database to a data warehouse for analysis.

Database Administrator

A database administrator is responsible for the design, implementation, maintenance, and operational aspects of on-premises and cloud-based database systems. They're responsible for the overall availability and consistent performance and optimizations of databases. They work with stakeholders to implement policies, tools, and processes for backup and recovery plans to recover following a natural disaster or human-made error.

The database administrator is also responsible for managing the security of the data in the database, granting privileges over the data, granting or denying access to users as appropriate.

Data Engineer

A data engineer collaborates with stakeholders to design and implement data-related workloads, including data ingestion pipelines, cleansing and transformation activities, and data stores for analytical workloads. They use a wide range of data platform technologies, including relational and non-relational databases, file stores, and data streams.

They're also responsible for ensuring that the privacy of data is maintained within the cloud and spanning from on-premises to the cloud data stores. They own the management and monitoring of data pipelines to ensure that data loads perform as expected.

Data Analyst

A data analyst enables businesses to maximize the value of their data assets. They're responsible for exploring data to identify trends and relationships, designing and building analytical models, and enabling advanced analytics capabilities through reports and visualizations.

A data analyst processes raw data into relevant insights based on identified business requirements to deliver relevant insights.

Note

The roles described here represent the key data-related roles found in most medium to large organizations. There are additional data-related roles not mentioned here, such as *data scientist* and *data architect*; and there are other technical professionals that work with data, including *application developers* and *software engineers*.

Understand normalization

Completed100 XP

- 6 minutes

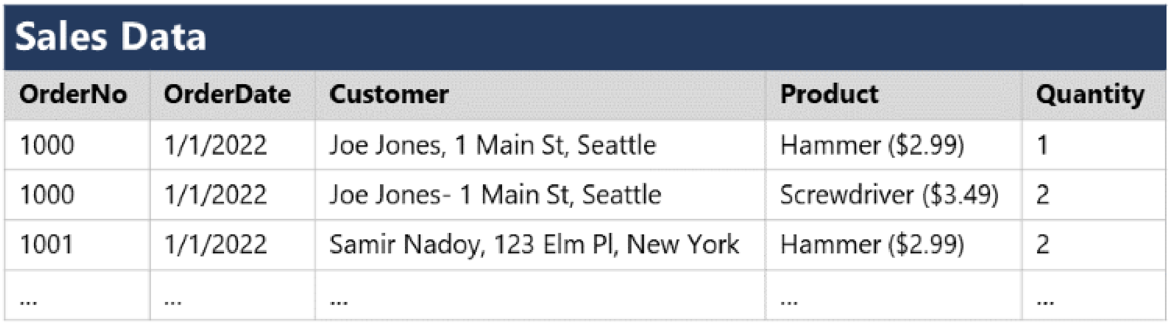
Normalization is a term used by database professionals for a schema design process that minimizes data duplication and enforces data integrity.

While there are many complex rules that define the process of refactoring data into various levels (or *forms*) of normalization, a simple definition for practical purposes is:

1. Separate each *entity* into its own table.
2. Separate each discrete *attribute* into its own column.
3. Uniquely identify each entity instance (row) using a *primary key*.

4. Use *foreign key* columns to link related entities.

To understand the core principles of normalization, suppose the following table represents a spreadsheet that a company uses to track its sales.

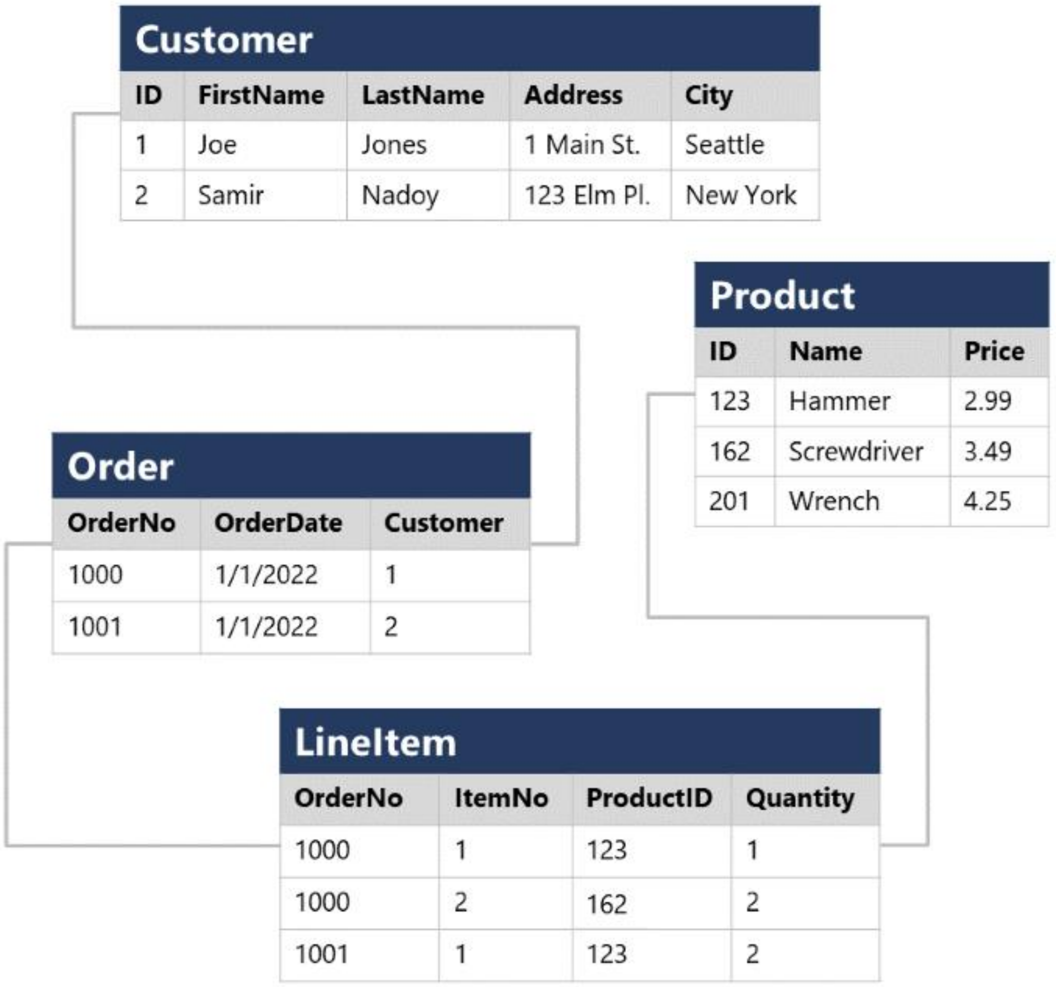


Sales Data

OrderNo	OrderDate	Customer	Product	Quantity
1000	1/1/2022	Joe Jones, 1 Main St, Seattle	Hammer ($2.99)	1
1000	1/1/2022	Joe Jones- 1 Main St, Seattle	Screwdriver ($3.49)	2
1001	1/1/2022	Samir Nadoy, 123 Elm Pl, New York	Hammer ($2.99)	2
...	...	...	...	...

Notice that the customer and product details are duplicated for each individual item sold; and that the customer name and postal address, and the product name and price are combined in the same spreadsheet cells.

Now let's look at how normalization changes the way the data is stored.



Customer

ID	FirstName	LastName	Address	City
1	Joe	Jones	1 Main St.	Seattle
2	Samir	Nadoy	123 Elm Pl.	New York

Product

ID	Name	Price
123	Hammer	2.99
162	Screwdriver	3.49
201	Wrench	4.25

Order

OrderNo	OrderDate	Customer
1000	1/1/2022	1
1001	1/1/2022	2

LineItem

OrderNo	ItemNo	ProductID	Quantity
1000	1	123	1
1000	2	162	2
1001	1	123	2

Each entity that is represented in the data (customer, product, sales order, and line item) is stored in its own table, and each discrete attribute of those entities is in its own column.

Recording each instance of an entity as a row in an entity-specific table removes duplication of data. For example, to change a customer's address, you need only modify the value in a single row.

The decomposition of attributes into individual columns ensures that each value is constrained to an appropriate data type - for example, product prices must be decimal values, while line item quantities must be integer numbers. Additionally, the creation of individual columns provides a useful level of granularity in the data for querying - for example, you can easily filter customers to those who live in a specific city.

Instances of each entity are uniquely identified by an ID or other key value, known as a *primary key*; and when one entity references another (for example, an order has an associated customer), the primary key of the related entity is stored as a *foreign key*. You can look up the address of the customer (which is stored only once) for each record in the **Order** table by referencing the corresponding record in the **Customer** table. Typically, a relational database management system (RDBMS) can enforce referential integrity to ensure that a value entered into a foreign key field has an existing corresponding primary key in the related table – for example, preventing orders for non-existent customers.

In some cases, a key (primary or foreign) can be defined as a *composite* key based on a unique combination of multiple columns. For example, the **LineItem** table in the example above uses a unique combination of **OrderNo** and **ItemNo** to identify a line item from an individual order.

Describe Azure services for open-source databases

Completed100 XP

- 6 minutes

In addition to Azure SQL services, Azure data services are available for other popular relational database systems, including MySQL, MariaDB, and PostgreSQL. The primary reason for these services is to enable organizations that use them in on-premises apps to move to Azure quickly, without making significant changes to their applications.

What are MySQL, MariaDB, and PostgreSQL?

MySQL, MariaDB, and PostgreSQL are relational database management systems that are tailored for different specializations.

MySQL started life as a simple-to-use open-source database management system. It's the leading open source relational database for *Linux, Apache, MySQL, and PHP* (LAMP) stack apps. It's available in several editions; Community, Standard, and Enterprise. The Community edition is available free-of-charge, and has historically been popular as a database management system for web applications, running under Linux. Versions are also available for Windows. Standard edition offers higher performance, and uses a different technology for storing data. Enterprise edition provides a comprehensive set of tools and features, including enhanced security, availability, and scalability. The Standard and Enterprise editions are the versions most frequently used by commercial organizations, although these versions of the software aren't free.

MariaDB is a newer database management system, created by the original developers of MySQL. The database engine has since been rewritten and optimized to improve performance. MariaDB offers compatibility with Oracle Database (another popular commercial database management system). One notable feature of MariaDB is its built-in support for temporal data. A table can hold several versions of data, enabling an application to query the data as it appeared at some point in the past.

PostgreSQL is a hybrid relational-object database. You can store data in relational tables, but a PostgreSQL database also enables you to store custom data types, with their own non-relational properties. The database management system is extensible; you can add code modules to the database, which can be run by queries. Another key feature is the ability to store and manipulate geometric data, such as lines, circles, and polygons.

PostgreSQL has its own query language called *pgsql*. This language is a variant of the standard relational query language, SQL, with features that enable you to write stored procedures that run inside the database.

Azure Database for MySQL

Azure Database for MySQL is a PaaS implementation of MySQL in the Azure cloud, based on the MySQL Community Edition.

The Azure Database for MySQL service includes high availability at no additional cost, and scalability as required. You only pay for what you use. Automatic backups are provided, with point-in-time restore.

The server provides connection security to enforce firewall rules and, optionally, require SSL connections. Many server parameters enable you to configure server settings such as lock modes, maximum number of connections, and timeouts.

Azure Database for MySQL provides a global database system that scales up to large databases without the need to manage hardware, network components, virtual servers, software patches, and other underlying components.

Certain operations aren't available with Azure Database for MySQL. These functions are primarily concerned with security and administration. Azure manages these aspects of the database server itself.

Benefits of Azure Database for MySQL

You get the following features with Azure Database for MySQL:

- High availability features built-in.
- Predictable performance.
- Easy scaling that responds quickly to demand.
- Secure data, both at rest and in motion.
- Automatic backups and point-in-time restore for the last 35 days.
- Enterprise-level security and compliance with legislation.

The system uses pay-as-you-go pricing so you only pay for what you use.

Azure Database for MySQL servers provides monitoring functionality to add alerts, and to view metrics and logs.

Azure Database for MariaDB

Azure Database for MariaDB is an implementation of the MariaDB database management system adapted to run in Azure. It's based on the MariaDB Community Edition.

The database is fully managed and controlled by Azure. Once you've provisioned the service and transferred your data, the system requires almost no additional administration.

Benefits of Azure Database for MariaDB

Azure Database for MariaDB delivers:

- Built-in high availability with no additional cost.
- Predictable performance, using inclusive pay-as-you-go pricing.
- Scaling as needed within seconds.
- Secured protection of sensitive data at rest and in motion.
- Automatic backups and point-in-time-restore for up to 35 days.

- Enterprise-grade security and compliance.

Azure Database for PostgreSQL

If you prefer PostgreSQL, you can choose Azure Database for PostgreSQL to run a PaaS implementation of PostgreSQL in the Azure Cloud. This service provides the same availability, performance, scaling, security, and administrative benefits as the MySQL service.

Some features of on-premises PostgreSQL databases aren't available in Azure Database for PostgreSQL. These features are mostly concerned with the extensions that users can add to a database to perform specialized tasks, such as writing stored procedures in various programming languages (other than pgsql, which is available), and interacting directly with the operating system. A core set of the most frequently used extensions is supported, and the list of available extensions is under continuous review.

Azure Database for PostgreSQL Flexible Server

The flexible-server deployment option for PostgreSQL is a fully managed database service. It provides a high level of control and server configuration customizations, and provides cost optimization controls.

Benefits of Azure Database for PostgreSQL

Azure Database for PostgreSQL is a highly available service. It contains built-in failure detection and failover mechanisms.

Users of PostgreSQL will be familiar with the **pgAdmin** tool, which you can use to manage and monitor a PostgreSQL database. You can continue to use this tool to connect to Azure Database for PostgreSQL. However, some server-focused functionality, such as performing server backup and restore, aren't available because the server is managed and maintained by Microsoft.

Azure Database for PostgreSQL records information about queries run against databases on the server, and saves them in a database named *azure_sys*. You query the *query_store.qs_view* view to see this information, and use it to monitor the queries that users are running. This information can prove invaluable if you need to fine-tune the queries performed by your applications.

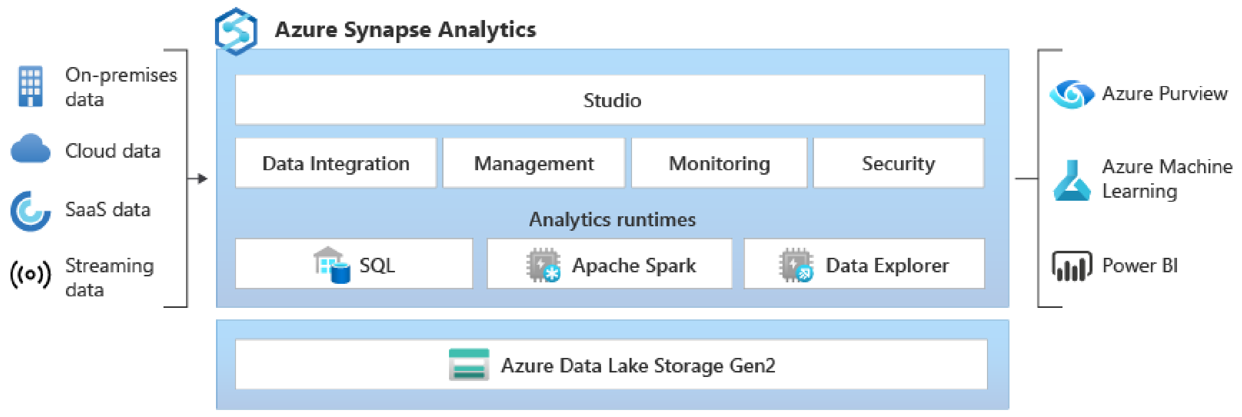
What is Azure Synapse Data Explorer? (Preview)

- Article
- 03/14/2023
- 5 minutes to read
- 4 contributors

Feedback

In this article

Azure Synapse Data Explorer provides customers with an interactive query experience to unlock insights from log and telemetry data. To complement existing SQL and Apache Spark analytics runtime engines, the Data Explorer analytics runtime is optimized for efficient log analytics using powerful indexing technology to automatically index free-text and semi-structured data commonly found in telemetry data.



To learn more, see the following video:

What makes Azure Synapse Data Explorer unique?

- **Easy ingestion** - Data Explorer offers built-in integrations for no-code/low-code, high-throughput data ingestion, and caching data from real-time sources. Data can be ingested from sources such as Event Hub, Kafka, Azure Data Lake, open source agents such as Fluentd/Fluent Bit, and a wide variety of cloud and on-premises data sources.
- **No complex data modeling** - With Data Explorer, there is no need to build complex data models and no need for complex scripting to transform data before it's consumed.
- **No index maintenance** - There is no need for maintenance tasks to optimize data for query performance and no need for index maintenance. With Data Explorer, all raw data is available immediately, allowing you to run high-performance and high-concurrency queries on your streaming and persistent data. You can use these queries to build near real-time dashboards and alerts, and connect operational analytics data with the rest of data analytics platform.
- **Democratizing data analytics** - Data Explorer democratizes self-service, big data analytics with the intuitive Kusto Query Language (KQL) that provides the expressiveness and power of SQL with the simplicity of Excel. KQL is highly optimized for exploring raw telemetry and time series data by leveraging Data Explorer's best-in-class text indexing technology for efficient free-text and regex search, and comprehensive parsing capabilities for querying traces\text data and JSON semi-structured data including arrays and nested structures. KQL offers advanced time series support for creating, manipulating, and analyzing multiple time series with in-engine Python execution support for model scoring.
- **Proven technology at petabyte scale** - Data Explorer is a distributed system with compute and storage that can scale independently, enabling analytics on gigabytes or petabytes of data.
- **Integrated** - Azure Synapse Analytics provides interoperability across data between Data Explorer, Apache Spark, and SQL engines empowering data engineers, data scientists, and data analysts to easily, and securely, access and collaborate on the same data in the data lake.

When to use Azure Synapse Data Explorer?

Use Data Explorer as a data platform for building near real-time log analytics and IoT analytics solutions to:

- Consolidate and correlate your logs and events data across on-premises, cloud, and third-party data sources.
- Accelerate your AI Ops journey (pattern recognition, anomaly detection, forecasting, and more).
- Replace infrastructure-based log search solutions to save cost and increase productivity.
- Build IoT analytics solutions for your IoT data.
- Build analytics SaaS solutions to offer services to your internal and external customers.

Data Explorer pool architecture

Data Explorer pools implement a scale out architecture by separating the compute and storage resources. This enables you to independently scale each resource and, for example, run multiple read only computes on the same data. Data Explorer pools consist of a set of computes running the engine that is responsible for automatically indexing, compressing, caching, and serving distributed queries. They also have a second set of computes running the data management service responsible for background system jobs, and managed and queued data ingestion. All data is persisted on managed blob storage accounts using a compressed columnar format.

Data Explorer pools support a rich ecosystem for ingesting data using connectors, SDKs, REST APIs, and other managed capabilities. It offers various ways to consume data for adhoc queries, reports, dashboards, alerts, REST APIs, and SDKs.

There are many unique capabilities that makes Data Explore the best analytical engine for log and time series analytics on Azure. If you are interested in learning more about how Data Explorer works, see Azure Data Explorer white paper.

The following sections highlight the key differentiators.

Free-text and semi-structured data indexing enables near real time high performance and high concurrent queries

Data Explorer indexes semi-structured data (JSON) and unstructured data (free text) that makes running queries very performant on this type of data. By default, every field is indexed during the data ingestion with the option to use a low-level encoding policy to fine tune or disable the index for specific fields. The scope of the index is a single data shard.

The implementation of the index depends on the type of the field, as follows:

Field type	Indexing implementation
String	The engine builds an inverted term index for string column values. Each string value is analyzed and split into n an ordered list of logical positions, containing record ordinals, is recorded for each term. The resulting sorted li associated positions is stored as an immutable B-tree.
Numeric **DateTime** **TimeSpan**	The engine builds a simple range-based forward index. The index records the min/max values for each block, fo for the entire column within the data shard.
Dynamic	The ingestion process enumerates all "atomic" elements within the dynamic value, such as property names, va elements, and forwards them to the index builder. Dynamic fields have the same inverted term index as string

These efficient indexing capabilities enables Data Explore to make the data available in near-real-time for high-performance and high-concurrency queries. The system automatically optimizes data shards to further boost performance.

Kusto Query Language

KQL has a large, growing community with the rapid adoption of Azure Monitor Log Analytics and Application Insights, Microsoft Sentinel, Azure Data Explorer, and other Microsoft offerings. The language is well designed with an easy-to-read syntax and provides a smooth transition from simple one-liner to complex data processing queries. This allows Data Explorer to provide rich Intellisense support and a rich set of language construct and built-in capabilities for aggregations, time series, and user analytics that aren't available in SQL for rapid exploration of telemetry data.

Explore Apache Spark on Microsoft Azure

Completed 100 XP

- 3 minutes

Apache Spark is a distributed processing framework for large scale data analytics. You can use Spark on Microsoft Azure in the following services:

- Azure Synapse Analytics
- Azure Databricks
- Azure HDInsight

Spark can be used to run code (usually written in Python, Scala, or Java) in parallel across multiple cluster nodes, enabling it to process very large volumes of data efficiently. Spark can be used for both batch processing and stream processing.

Spark Structured Streaming

To process streaming data on Spark, you can use the *Spark Structured Streaming* library, which provides an application programming interface (API) for ingesting, processing, and outputting results from perpetual streams of data.

Spark Structured Streaming is built on a ubiquitous structure in Spark called a *dataframe*, which encapsulates a table of data. You use the Spark Structured Streaming API to read data from a real-time data source, such as a Kafka hub, a file store, or a network port, into a "boundless" dataframe that is continually populated with new data from the stream. You then define a query on the dataframe that selects, projects, or aggregates the data - often in temporal windows. The results of the query generate another dataframe, which can be persisted for analysis or further processing.

Spark Structured Streaming is a great choice for real-time analytics when you need to incorporate streaming data into a Spark based data lake or analytical data store.

Note

For more information about Spark Structured Streaming, see the **Spark Structured Streaming programming guide.**

Delta Lake

Delta Lake is an open-source storage layer that adds support for transactional consistency, schema enforcement, and other common data warehousing features to data lake storage. It also unifies storage for streaming and batch data, and can be used in Spark to define relational tables for both batch and stream processing. When used for stream processing, a Delta Lake table can be used as a streaming source for queries against real-time data, or as a sink to which a stream of data is written.

The Spark runtimes in Azure Synapse Analytics and Azure Databricks include support for Delta Lake.

Delta Lake combined with Spark Structured Streaming is a good solution when you need to abstract batch and stream processed data in a data lake behind a relational schema for SQL-based querying and analysis.

Note

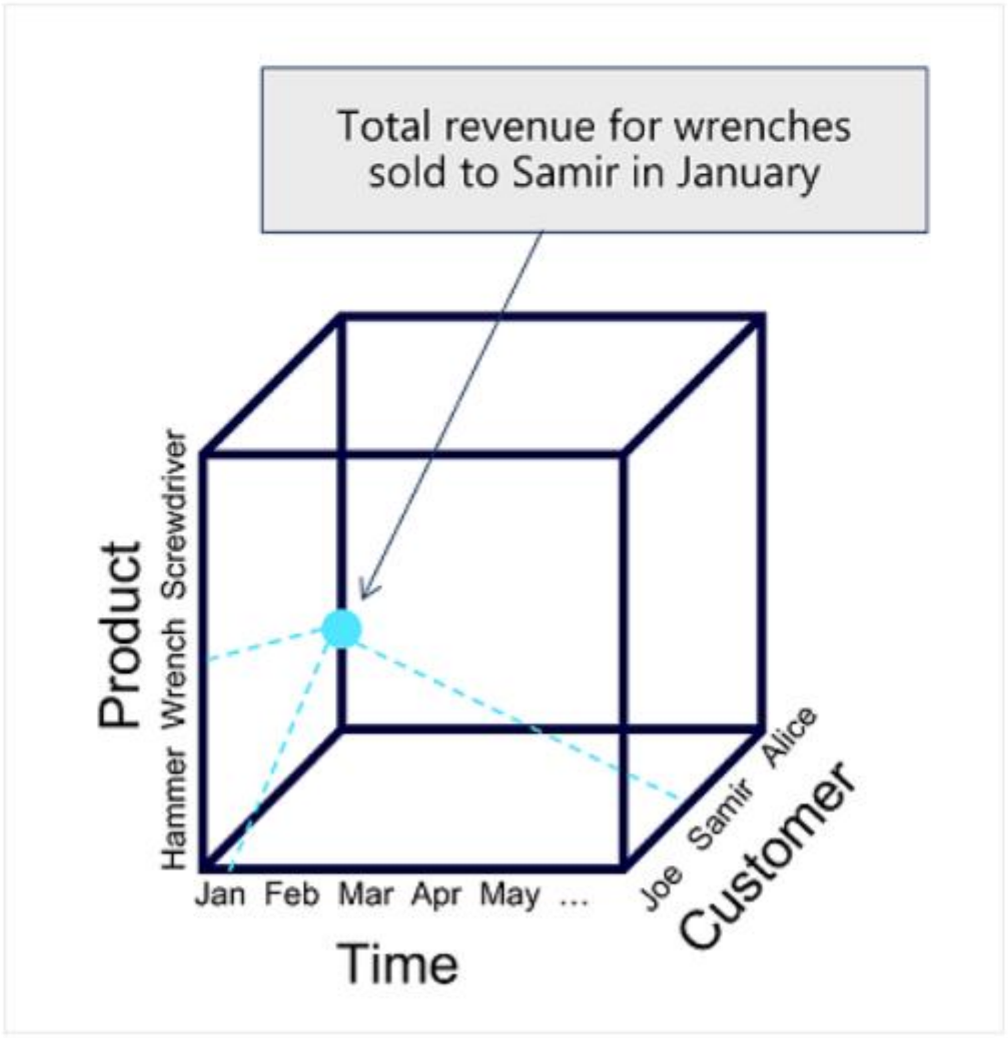
For more information about Delta Lake, see **What is Delta Lake?**

Describe core concepts of data modeling

Completed100 XP

- 5 minutes

Analytical models enable you to structure data to support analysis. Models are based on related tables of data and define the numeric values that you want to analyze or report (known as *measures*) and the entities by which you want to aggregate them (known as *dimensions*). For example, a model might include a table containing numeric measures for sales (such as revenue or quantity) and dimensions for products, customers, and time. This would enable you aggregate sale measures across one or more dimensions (for example, to identify total revenue by customer, or total items sold by product per month). Conceptually, the model forms a multidimensional structure, which is commonly referred to as a *cube*, in which any point where the dimensions intersect represents an aggregated measure for those dimensions.)



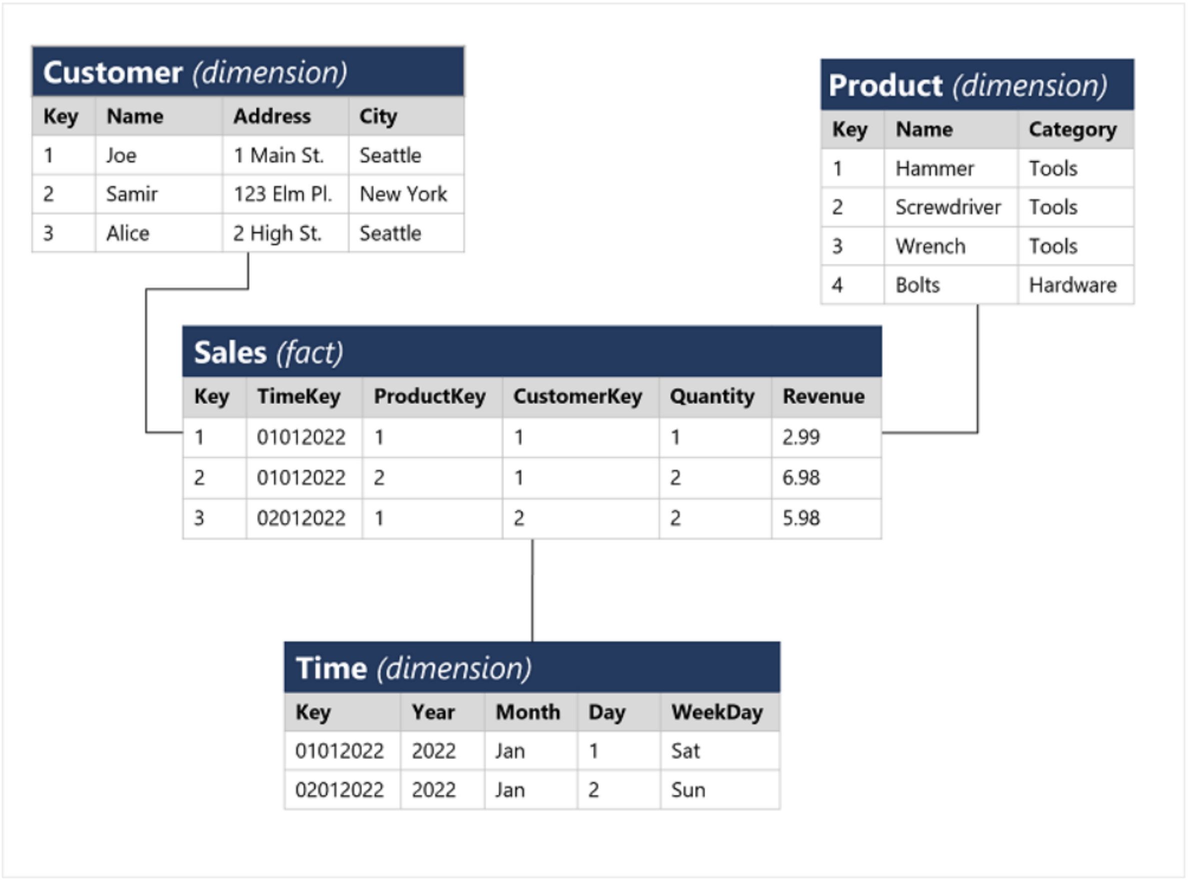
Note

Although we commonly refer to an analytical model as a *cube*, there can be more (or fewer) than three dimensions – it's just not easy for us to visualize more than three!

Tables and schema

Dimension tables represent the entities by which you want to aggregate numeric measures – for example product or customer. Each entity is represented by a row with a unique key value. The remaining columns represent attributes of an entity – for example, products have names and categories, and customers have addresses and cities. It's common in most analytical models to include a *Time* dimension so that you can aggregate numeric measures associated with events over time.

The numeric measures that will be aggregated by the various dimensions in the model are stored in *Fact* tables. Each row in a fact table represents a recorded event that has numeric measures associated with it. For example, the **Sales** table in the schema below represents sales transactions for individual items, and includes numeric values for quantity sold and revenue.



Customer *(dimension)*

Key	Name	Address	City
1	Joe	1 Main St.	Seattle
2	Samir	123 Elm Pl.	New York
3	Alice	2 High St.	Seattle

Product *(dimension)*

Key	Name	Category
1	Hammer	Tools
2	Screwdriver	Tools
3	Wrench	Tools
4	Bolts	Hardware

Sales *(fact)*

Key	TimeKey	ProductKey	CustomerKey	Quantity	Revenue
1	01012022	1	1	1	2.99
2	01012022	2	1	2	6.98
3	02012022	1	2	2	5.98

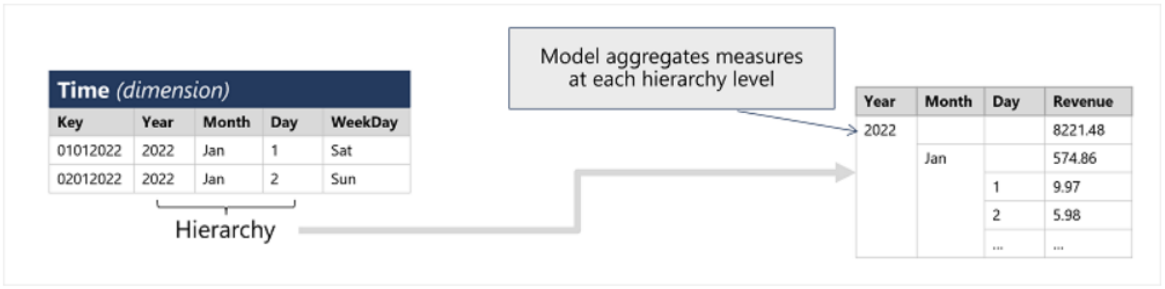
Time *(dimension)*

Key	Year	Month	Day	WeekDay
01012022	2022	Jan	1	Sat
02012022	2022	Jan	2	Sun

This type of schema, where a fact table is related to one or more dimension tables, is referred to as a star schema (imagine there are five dimensions related to a single fact table – the schema would form a five-pointed star!). You can also define a more complex schema in which dimension tables are related to additional tables containing more details (for example, you could represent attributes of product categories in a separate **Category** table that is related to the **Product** table – in which case the design is referred to as a snowflake schema. The schema of fact and dimension tables is used to create an analytical model, in which measure aggregations across all dimensions are pre-calculated; making performance of analysis and reporting activities much faster than calculating the aggregations each time.)

Attribute hierarchies

One final thing worth considering about analytical models is the creation of attribute *hierarchies* that enable you to quickly *drill-up* or *drill-down* to find aggregated values at different levels in a hierarchical dimension. For example, consider the attributes in the dimension tables we've discussed so far. In the **Product** table, you can form a hierarchy in which each category might include multiple named products. Similarly, in the **Customer** table, a hierarchy could be formed to represent multiple named customers in each city. Finally, in the **Time** table, you can form a hierarchy of year, month, and day. The model can be built with pre-aggregated values for each level of a hierarchy, enabling you to quickly change the scope of your analysis – for example, by viewing total sales by year, and then drilling down to see a more detailed breakdown of total sales by month.

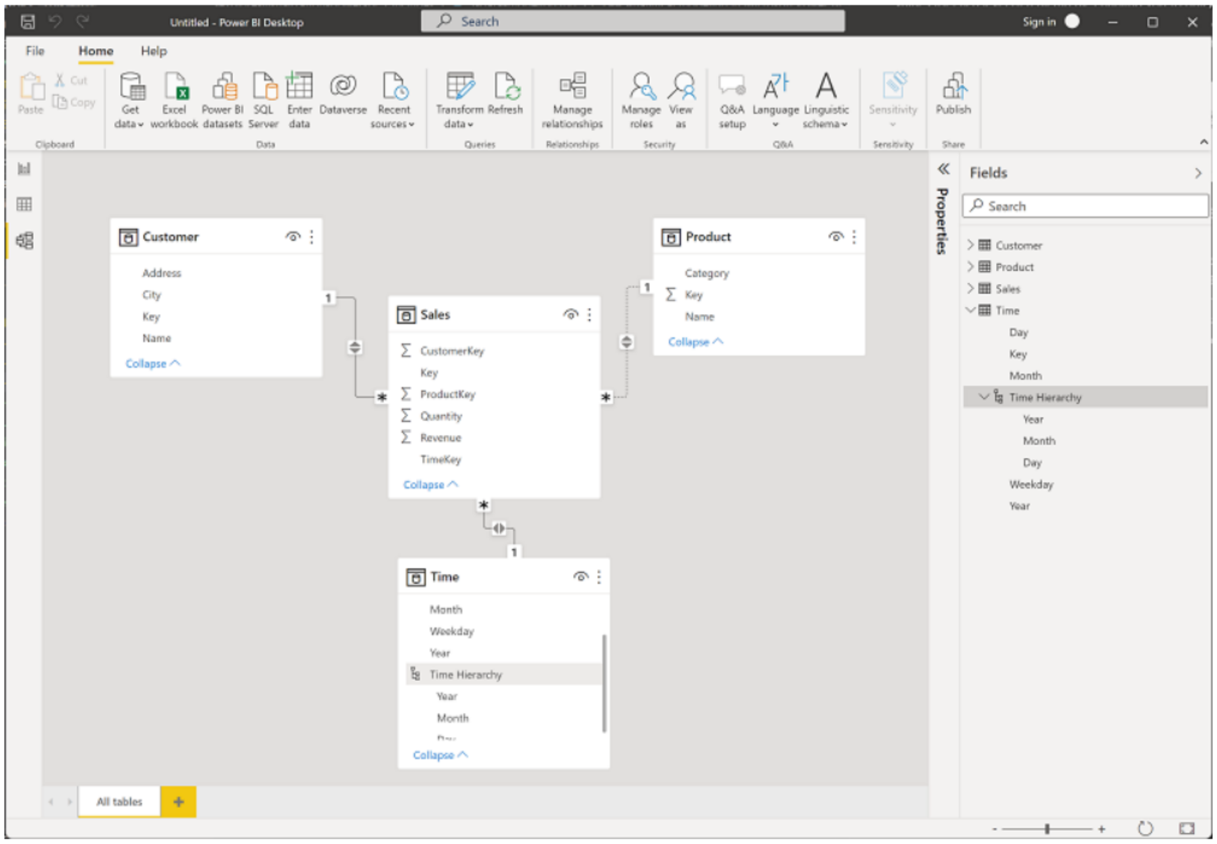


Key	Year	Month	Day	WeekDay
01012022	2022	Jan	1	Sat
02012022	2022	Jan	2	Sun

Year	Month	Day	Revenue
2022			8221.48
	Jan		574.86
		1	9.97
		2	5.98
		...	...

Analytical modeling in Microsoft Power BI

You can use Power BI to define an analytical model from tables of data, which can be imported from one or more data source. You can then use the data modeling interface on the **Model** tab of Power BI Desktop to define your analytical model by creating relationships between fact and dimension tables, defining hierarchies, setting data types and display formats for fields in the tables, and managing other properties of your data that help define a rich model for analysis.


Untitled - Power BI Desktop
Search
Sign in
File
Home
Help
Paste
Cut
Copy
Get data
Excel workbook
Power BI datasets
SQL Server
Enter data
Dataverse
Recent sources
Transform data
Refresh
Manage relationships
Manage roles
View as
Q&A setup
Language
Linguistic schema
Sensitivity
Publish
Clipboard
Data
Queries
Relationships
Security
Q&A
Sensitivity
Share
Customer
Address
City
Key
Name
Collapse
Sales
CustomerKey
Key
ProductKey
Quantity
Revenue
TimeKey
Collapse
Product
Category
Key
Name
Collapse
Time
Month
Weekday
Year
Time Hierarchy
Year
Month
Collapse
1
*
1
*
*
1
Properties
Fields
Search
Customer
Product
Sales
Time
Day
Key
Month
Time Hierarchy
Year
Month
Day
Weekday
Year
All tables