23.1 Big data

Transactional data and big data

The relational model and SQL were developed as research projects in the 1970s and released as products in the 1980s. During this period, the primary focus of databases was storing and retrieving highly structured data created within an organization, such as airline reservations, student records, and sales transactions. Typical database sizes reached gigabytes and occasionally terabytes (thousand billion bytes). Structured data created within an organization, with sizes ranging from gigabytes to terabytes, is called ***transactional data***.

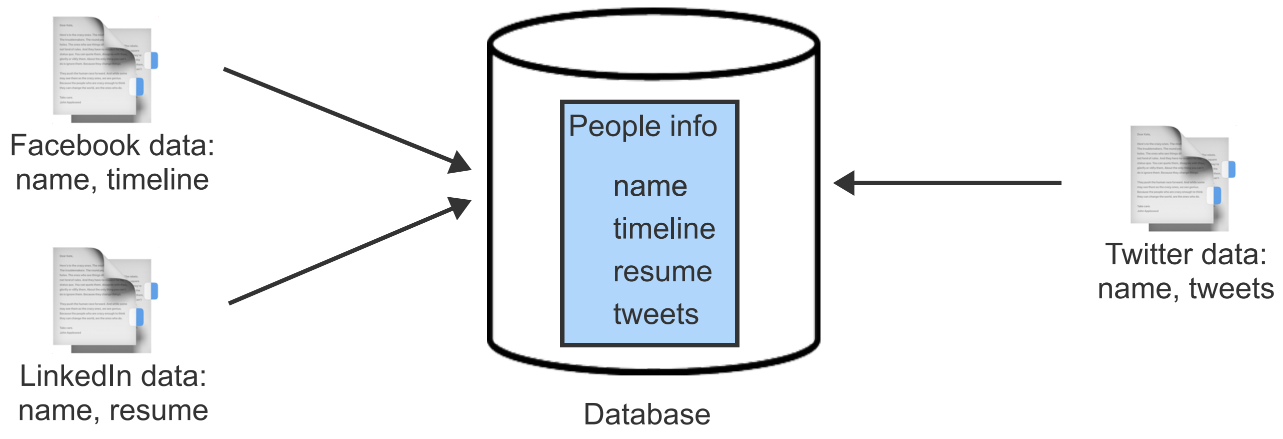
Since the explosion of the internet and digital media in the 1990s and 2000s, new computer applications have emerged with significantly different database requirements. Example applications include:

- Internet of things (devices such as thermostats connected to the internet)
- Email
- Social media (Ex: Facebook, Instagram, and Twitter)
- Web advertising

Data generated by new internet and multimedia applications is commonly called ***big data*** and differs from transactional data in four ways:

- *Volume*. Typical size ranges from terabytes to petabytes (million billion bytes), occasionally reaching exabytes (billion billion bytes).
- *Velocity*. Big data is generated at extremely high rates. Facebook users upload roughly a billion photos per day, or 10,000 per second. Twitter generates roughly 6,000 tweets per second. Click rates on popular websites can be significantly higher.
- *Variety*. Variety means both unstructured and rapidly changing data types. Unstructured data refers to information embedded in complex data types like images, video, GPS coordinates, and natural language. Rapidly changing data means the information content of records vary greatly, as in data collected from social media. Both unstructured and rapidly changing data are common in big data.
- *Veracity*. Transactional data is typically created by an organization's employees or trusted partners. Big data is often generated by the general public. Consequently, the accuracy of big data varies much more than transactional data.

PARTICIPATION ACTIVITY 23.1.1: Big data variety.


Facebook data: name, timeline
LinkedIn data: name, resume
People info
name
timeline
resume
tweets
Database
Twitter data: name, tweets

Animation content:

Static figure:
Two diagrams appear. In the first diagram, five icons represent different types of unstructured data - text, video, web log, social media, and sensor data. In the second diagram, a database icon is surrounded by data sources named Facebook, LinkedIn, and Twitter. Arrows represent data flowing from each data source to the database.

Animation captions:

1. Variety refers to unstructured data, such as text files, video, web logs, social media, and sensor data.
2. Variety also refers to variable data structures. Ex: Facebook, LinkedIn, and Twitter contain different information about people, which might be combined in big data.

PARTICIPATION ACTIVITY | 23.1.2: Transactional data and big data.

Determine if each data set is transactional, big, or has characteristics of both.

1) The California Department of Motor Vehicles processes vehicle registrations, driver's licenses, and traffic violations for 26 million registered drivers. Assume each driver creates 4 transactions per year at 1000 bytes per transaction, on average, and 5 years of data are stored in a database.

- ○ Transactional data
- ○ Big data
- ○ Characteristics of both

2) YouTube users view approximately 4 million videos per minute worldwide. YouTube tracks all views and information about video content to optimize advertising shown to each user. Assume YouTube stores approximately 10,000 bytes per view annually.

- ○ Transactional data
- ○ Big data
- ○ Characteristics of both

3) Visa processes approximately 30 billion credit card transactions per quarter (three months) and stores 10 years of history in a database for use in fraud detection. Assume each transaction is approximately 1000 bytes.

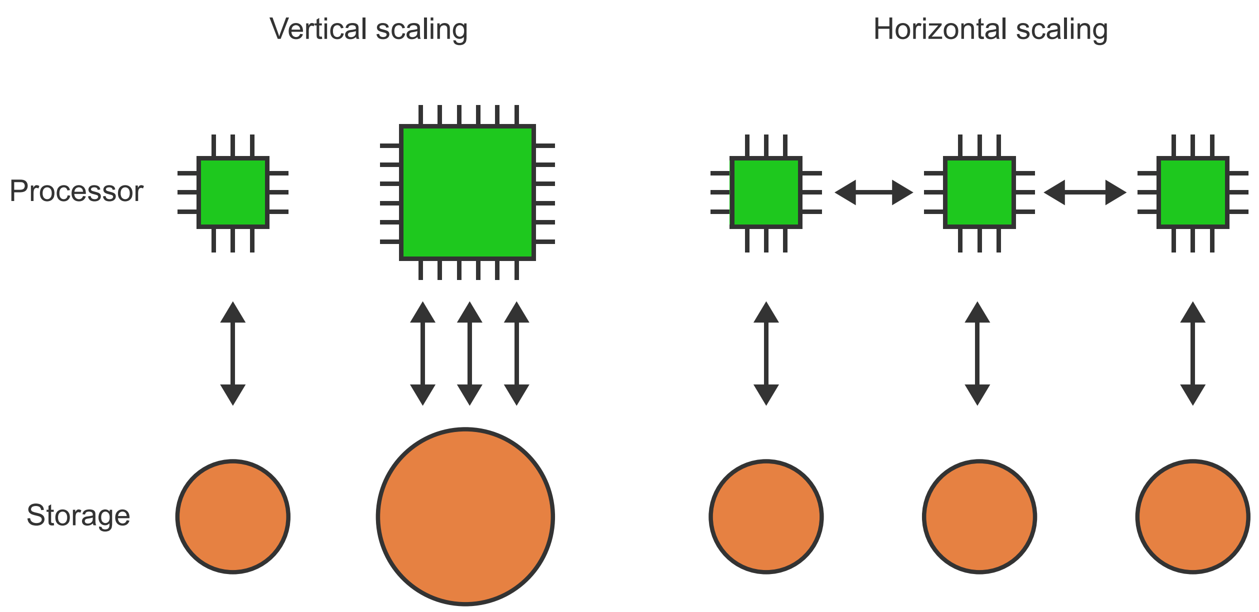
- ○ Transactional data
- ○ Big data
- ○ Characteristics of both

Transactional data requirements

Typically, transactional databases have the following processing requirements:

- *Stable schema*. Database designs are created once and change slowly.
- *Vertical scaling*. ***Vertical scaling*** increases capacity by increasing speed and size of CPUs and storage devices for a limited number of machines. ***Horizontal scaling*** increases capacity by adding large numbers of low-cost components like standard disk drives and CPUs. To accommodate increasing database sizes, transactional applications commonly scale vertically, not horizontally.
- *Stringent transaction requirements*. In transactional data, each transaction must be accurately tracked and must obey ACID properties.
- *Limited replication*. Data is often replicated to ensure availability in the event a component fails. However, the number of copies is limited since ensuring consistency of each copy entails overhead and slows processing.
- *Partitioning*. ***Partitioning*** splits large tables into separate physical files on one machine.
- *Rapid insert, update, delete*. Typically, transactional databases prioritize insert, update, and delete operations to ensure many users can submit transactions quickly while maintaining ACID properties. When data analysis speed is a priority, analysis might be executed on a separate copy of the data, such as a data warehouse.

PARTICIPATION ACTIVITY | 23.1.3: Vertical and horizontal scaling.



Animation content:

Static figure:
A diagram on the left depicts vertical scaling. A diagram on the right depicts horizontal scaling. The vertical scaling diagram shows a small processor icon above a small storage icon, next to a large processor icon above a large storage icon. The horizontal scaling diagram shows three small processor icons above three small storage icons. In both diagrams, a double-headed arrow connects each processor to the storage below. In the horizontal scaling diagram, additional double-headed arrows connect all processors.

Animation captions:

1. Vertical scaling increases processing speed, memory, and storage of a limited number of machines.
2. Horizontal scaling adds an unlimited number of machines working in parallel.

PARTICIPATION ACTIVITY 23.1.4: Relational database capabilities.

Relational databases were developed prior to big data. Historically, which of the following requirements were prioritized by relational databases?

1) Stable schema
- ○ True
- ○ False

2) Horizontal scaling
- ○ True
- ○ False

3) Relaxed transaction requirements
- ○ True
- ○ False

4) Partitioning
- ○ True
- ○ False

Big data requirements

Due to volume, velocity, variety, and veracity, big data has different processing requirements:

- *Flexible schema*. Databases must easily accommodate complex and variable data structures. Programming must be easy, and performance must be fast.
- *Horizontal scaling*. As database volume increases, vertical scaling becomes prohibitively expensive. Instead, big data applications typically scale by adding low-cost machines processing in parallel.
- *Relaxed transaction requirements*. The ACID requirements are relaxed in many big data applications. Ex: Propagation of updates across replicas may be asynchronous, causing queries against replicas to be slightly out of date.
- *Extensive replication*. Horizontally scaled components are less expensive and fail more often than vertically scaled components. Consequently, more copies are needed. Since transactional requirements are relaxed, many replicas can be added without slowing performance.
- *Sharding*. ***Sharding*** splits data sets across multiple machines . In comparison, partitioning splits data sets across multiple files on a single machine. Sharding and partitioning are alternative ways to split large data sets into smaller parts. Sharding enables horizontal scaling and supports massive volumes of data.
- *Rapid insert and analysis*. To accommodate high velocity, rapid insert is important. To analyze high volume data sets, rapid read and aggregation are important. Update and delete are less important, as big data applications often store a new version of data rather than modify existing data.

Table 23.1.1: Transactional data vs. big data.

	Transactional data	Big data
Schema	Stable	Flexible
Scaling	Vertical	Horizontal
Transactions	Stringent	Relaxed
Replication	Limited	Extensive
Fragmentation	Partitioning	Sharding
Processing	Insert, update, delete	Insert, read, aggregate

PARTICIPATION ACTIVITY | 23.1.5: Processing requirements for transactional and big data.

Match the term to the definition.

If unable to drag and drop, refresh the page.

Vertical scaling | Partitioning | Horizontal scaling | Sharding

	Increase capacity by increasing speed and size of a limited number of machines.
	Increase capacity by adding large numbers of low-cost components working in parallel.
	Split large data sets, such as a table, across separate files on one machine.
	Split large data sets across multiple machines working in parallel.

Reset

NoSQL databases

To meet big data requirements, dozens of non-relational databases have emerged since 2000. Initially, most non-relational databases did not support SQL, and hence are called NoSQL. Due to market demand for the SQL language, some NoSQL databases have introduced support for SQL or a SQL-like query language. As a result, the term NoSQL is now interpreted as 'Not Only SQL' and is sometimes written as NOSQL.

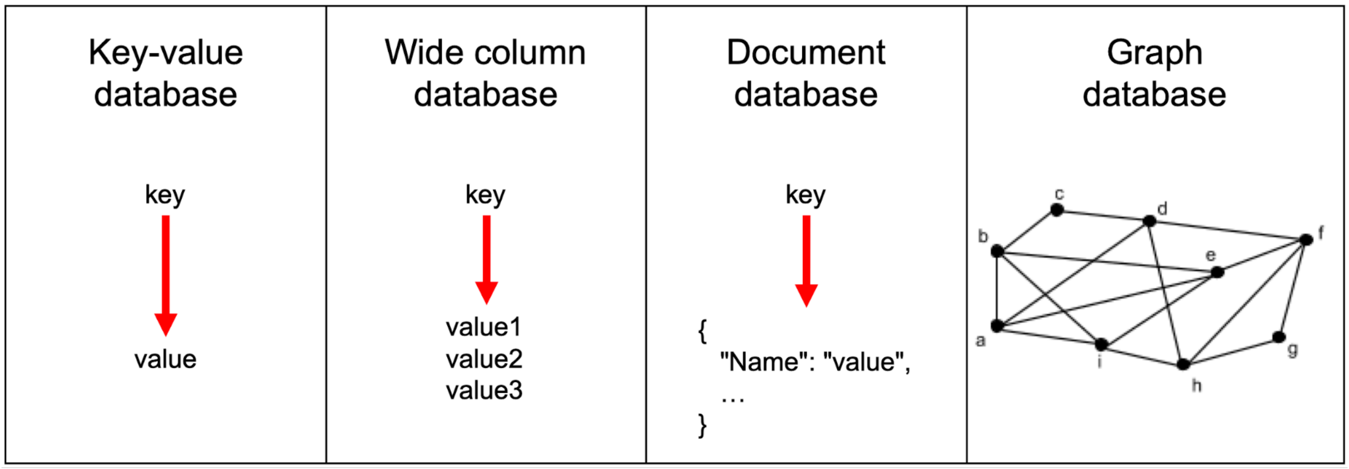
Four major categories of NoSQL databases have emerged:

- ***Key-value database***. Data is represented as a single key and an associated value. The key is used to access the value.

- ***Wide column database***, also called column-based, column family, or tabular database. Data is represented as a key and multiple values. Since each record has multiple values, a descriptive name is stored with each value.
- ***Document database***. Data is represented as a key and a 'document'. Usually the document is in a structured, human-readable format such as XML or JSON.
- ***Graph database***. Data is represented as a graph with nodes and edges.

The database models for the categories above are fundamentally different than the relational model.

Figure 23.1.1: NoSQL database categories.



NoSQL database technology is evolving rapidly, and some NoSQL databases do not fit precisely in the major categories. A ***multi-model database***, also called a ***hybrid database***, supports the data models of several categories. Ex: OrientDB is a leading hybrid database and supports database models of all four categories.

Relational databases such as PostgreSQL are also adding support for big data. Relational performance is catching up with, and in some cases exceeds, NoSQL performance on big data.

PARTICIPATION ACTIVITY | 23.1.6: NoSQL databases.

1) What does 'NoSQL' stand for?

- ○ SQL is not supported.
- ○ SQL is not the only query language supported.
- ○ Both answers are correct.

2) All NoSQL databases fall into exactly one category (key-value, wide column, document, and graph database).

- ○ True
- ○ False

3) Compared to NoSQL databases, what level of big data support do relational databases offer today?

- ○ Relational databases do not support big data requirements.
- ○ Relational database support for big data is similar to NoSQL database support.
- ○ Relational database support for big data is catching up to NoSQL databases in many big data requirements.

Exploring further:

- Overview of NoSQL databases

23.2 Key-value databases

Logical structure

A key-value database represents data as a key and a value. The key is a unique identifier used to access values. The value is the data managed by the system, usually kilobytes or several megabytes per value. Ex: The key might be a student identification number, and the value might be a photograph of the student.

The values structure varies depending on the key-value database. In some databases, the value is an unstructured series of bytes, which must be interpreted by the application program. In other databases, the value can have structure, such as numeric, character, JSON, or geospatial data.

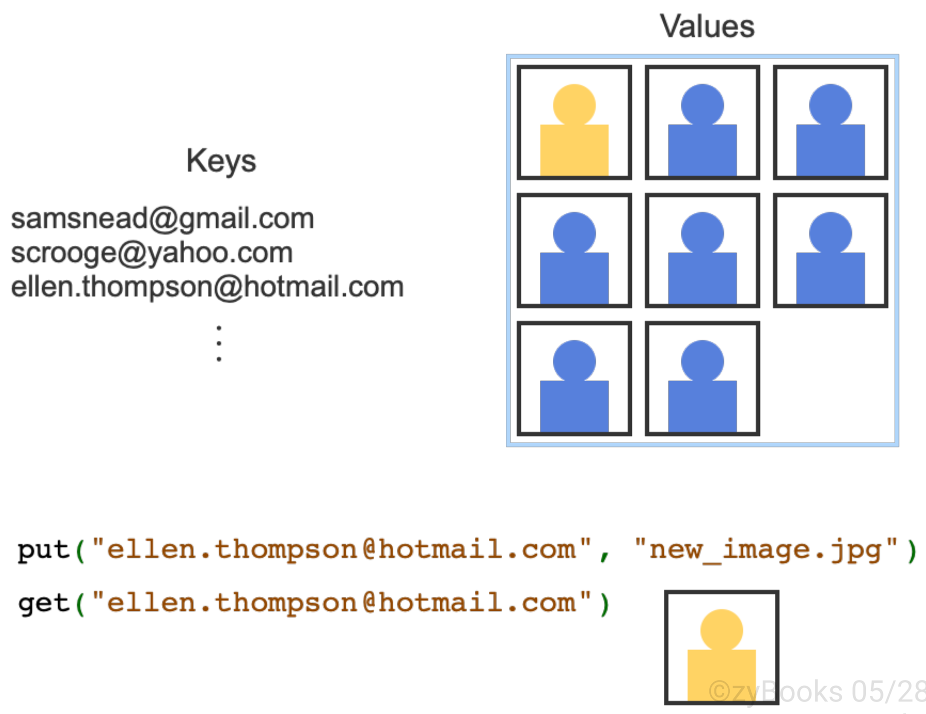
Some key-value databases allow multiple values per key. Each value has a name that describes the meaning of the value. The names and number of values can vary from one key to the next, giving rise to a flexible schema.

Key-value systems support a limited set of queries, such as:

- `put(key, value)` - Stores the value in the database, indexed by key.
- `get(key)` - Retrieves the value associated with the key.
- `multiGet(key1, ..., keyn)` - Retrieves the values associated with keys 1 through n.
- `delete(key)` - Deletes the value associated with the key.

Most key-value systems cannot access data via non-key values, and consequently do not support SQL natively. However, some databases provide interfaces that convert limited SQL queries to native queries.

PARTICIPATION ACTIVITY | 23.2.1: Key-value logical structure.



```
put("ellen.thompson@hotmail.com", "new_image.jpg")
get("ellen.thompson@hotmail.com")
```

Animation content:

Static figure:
On the left, the caption Keys is above a list of email addresses. On the right, the caption Values is

appear:
Begin code:
put(ellen.thompson@hotmail.com, new_image.jpg)
get(ellen.thompson@hotmail.com)
End code.
An icon representing a photograph of Ellen Thompson is to the right of the get statement.

Animation captions:

1. Keys are used to identify and locate values. In this example, the key is an email address.
2. Values are photographs of the person associated with the email address.
3. Each key is associated with one value.
4. The **`put()`** function stores a value in the database.
5. The **`get()`** function retrieves the value associated with a key.

PARTICIPATION ACTIVITY | 23.2.2: Key-value logical structure.

1) In a key-value database, the value is:

- ○ Always an unstructured series of bytes, which must be interpreted by the application software.
- ○ Always formatted as numeric or character data.
- ○ Unstructured or structured data, depending on the key-value database.

2) In a key-value database, the key:

- ○ Uniquely identifies a single value.
- ○ Specifies the location of a value in storage.
- ○ Is unique and also specifies the location of a value.

3) What is the size of a typical value?

○ Usually less than a kilobyte.

○ Usually kilobytes or megabytes.

○ Usually gigabytes to terabytes.

Physical structure

Key-value databases are commonly used for applications with the following requirements:

- *Simple data models*. Although the internal structure of the value might be complex, the internal structure is usually managed by the application program rather than the database.
- *Data access via the key*. In SQL, any column can appear in the WHERE clause. In key-value databases, the `get` query specifies the key only.
- *Large numbers of small records*. A typical key-value database might have terabytes of records, each kilobytes or megabytes wide.
- *Fast read and write*. Like most big data applications, key-value databases support high-velocity read and write but are not optimized for update.

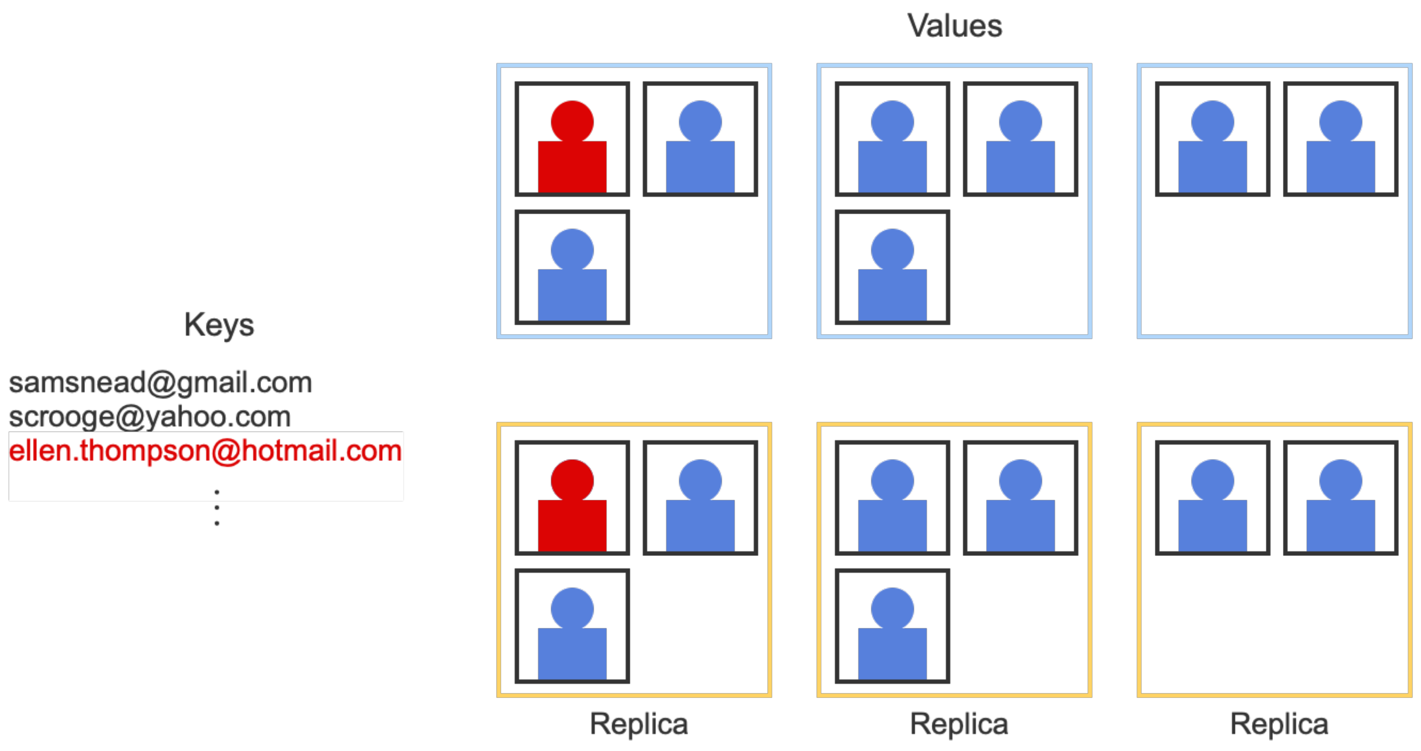
Website shopping cart data might exhibit the above requirements. Ex: The key is the shopper's username, and the value are items in the shopping cart. A key-value database enables fast storage and retrieval of shopping cart data for millions of concurrent shoppers.

In support of the application requirements above, key-value databases have physical structures such as:

- *Sharding*. Key-value databases can shard data across thousands of machines.
- *Replicated data*. Values are replicated for high availability in the event that storage devices fail.
- *Eventual consistency*. In key-value databases, replicas may become inconsistent at times. Allowing inconsistency across multiple replicas enables faster writes to individual replicas.
- *Hashing*. Keys are usually implemented as hash indexes, and values are stored in hash buckets, described elsewhere in this material. Implementing keys as hash indexes enables fast reads and writes. Hash buckets can be stored on different machines, enabling horizontal scaling.

Some key-value databases can maintain all data in random-access memory. This limits database size but enables extremely fast reads and writes.

PARTICIPATION ACTIVITY | 23.2.3: Key-value physical structure.



Animation content:

Static figure:
On the left, the caption keys is above a list of email addresses. On the right, the caption Values is above three groups of photograph icons. A copy of each photograph group, with caption replica, appears below the original .

Step 1: Values are grouped in hash buckets. The list of email addresses appear with caption keys. The upper photograph groups appear with caption values. Each photograph group has a caption hash bucket.

Step 2: Values are replicated on multiple devices for high availability and fast access. The photograph group copies appear with replica captions.

Step 3: Updates to values are applied to one replica. For fast updates and high availability, additional replicas are not updated within a transaction. One email address and one photograph icon in an upper group are highlighted, indicating the photo associated with the email address has been updated. The lower photograph groups fade out.

Step 4: If other replicas are accessed before an update is propagated, obsolete values are returned. The replica photograph groups fade back in. The replica photographs are not highlighted, indicating the replicas have not been updated.

Step 5: Eventually, the update is propagated to all replicas. A replica photograph is highlighted, indicating the replica has been updated.

Animation captions:

1. Values are grouped in hash buckets.
2. Values are replicated on multiple devices for high availability and fast access.
3. Updates to values are applied to one replica. For fast updates and high availability, additional replicas are not updated within a transaction.
4. If other replicas are accessed before an update is propagated, obsolete values are returned.
5. Eventually, the update is propagated to all replicas.

PARTICIPATION ACTIVITY | 23.2.4: Key-value physical structure.

1) Key-value databases require a fixed schema for all values.

○ True

○ False

2) All key-value databases support transactions.

○ True

○ False

3) Key-value databases store and access data using hashing techniques.

○ True

○ False

4) Key-value databases are optimized for queries that specify the key.

○ True

○ False

5) Key-value databases support foreign

keys and referential integrity.

- ○ True
- ○ False

Database systems

Prominent key-value databases include:

- *Redis from Redis Labs*. The name Redis is derived from Remote Dictionary Server. Redis was initially released in 2009 and is now available as open source, a commercial version, or cloud service. Redis ranks highest of all key-value databases in the DB-Engines ranking for 2019. Redis supports in-memory data for extremely fast read and write. Redis also supports complex data types such as sets of values and arrays of bits , enabling native operations like sorting set values or changing individual bits in a bit array.
- *DynamoDB from Amazon Web Services*. DynamoDB is available as a cloud service. DynamoDB was initially offered in 2012 as an improved version of an internal Amazon database. DynamoDB uses the term 'table', but DynamoDB tables are different than relational tables. Each table consists of a key and a group of items. Each item contains one or more name-value pairs. The names of each pair can vary, so a schema can change from one item to the next.
- *Oracle NoSQL*. Like Redis, Oracle NoSQL is available as open source and commercial software. Unlike Redis, Oracle NoSQL does not support complex data types — values are byte sequences that must be interpreted by application software. However, recent releases offer a software layer that interprets values as relational tables that support SQL queries.

Many key-value databases support additional database models since the key-value model is relatively simple and limited. Redis, DynamoDB, and Oracle NoSQL all support additional models.

Table 23.2.1: Key-value database systems.

	Developer	Initial release	Database models	License	DB-Engines rank (May 2020)
Redis	Redis Labs	2009	Key-value Document Graph	Cloud Commercial Open source	8

DynamoDB	Amazon Web Services	2012	Key-value Document	Cloud	16
Oracle NoSQL	Oracle	2011	Key-value Relational	Cloud Commercial Open source	76

PARTICIPATION ACTIVITY 23.2.5: Key-value database systems.

Match the database to its characteristics.

If unable to drag and drop, refresh the page.

DynamoDB **Oracle NoSQL** **Redis**

	Stores values as unformatted series of bytes.
	Stores key-value data in random-access memory.
	Available only as a cloud service from Amazon Web Services.

Reset

CHALLENGE ACTIVITY 23.2.1: Key-value databases.

544874.3500394.qx3zqy7

Start

Select all scenarios that may be captured as a key-value pair.

- ☐ User: Email-"Profile picture"
- ☐ Patient: "Full name"-Info
- ☐ Stock: Symbol-Price

1 2 3

Check Next

Exploring further:

- DB-Engines Database Ranking
- Redis
- Amazon DynamoDB
- Oracle NoSQL

23.3 Wide column databases

Logical structure

At a high level, wide column databases look like relational databases. Many wide column databases use relational terms like table, row, and column, but with different meanings.

A wide column database consists of multiple tables, each with a key and multiple column families:

- The key is a unique value used to identify and access individual rows.
- Each ***column family*** has a name. The number of column families and column family names are the same for all rows of a table.

Each column family contains one or more columns. Each column has a name, unique within the column family. One value is stored for each column in each row. The value is unstructured and must be interpreted by the application program.

Unlike a relational database, the columns of a column family may vary across rows. This enables wide column databases to efficiently store variable data, such as web pages with different structure and content. Despite the similarity in terminology to relational databases, wide column databases do not support foreign keys, joins, and SQL.

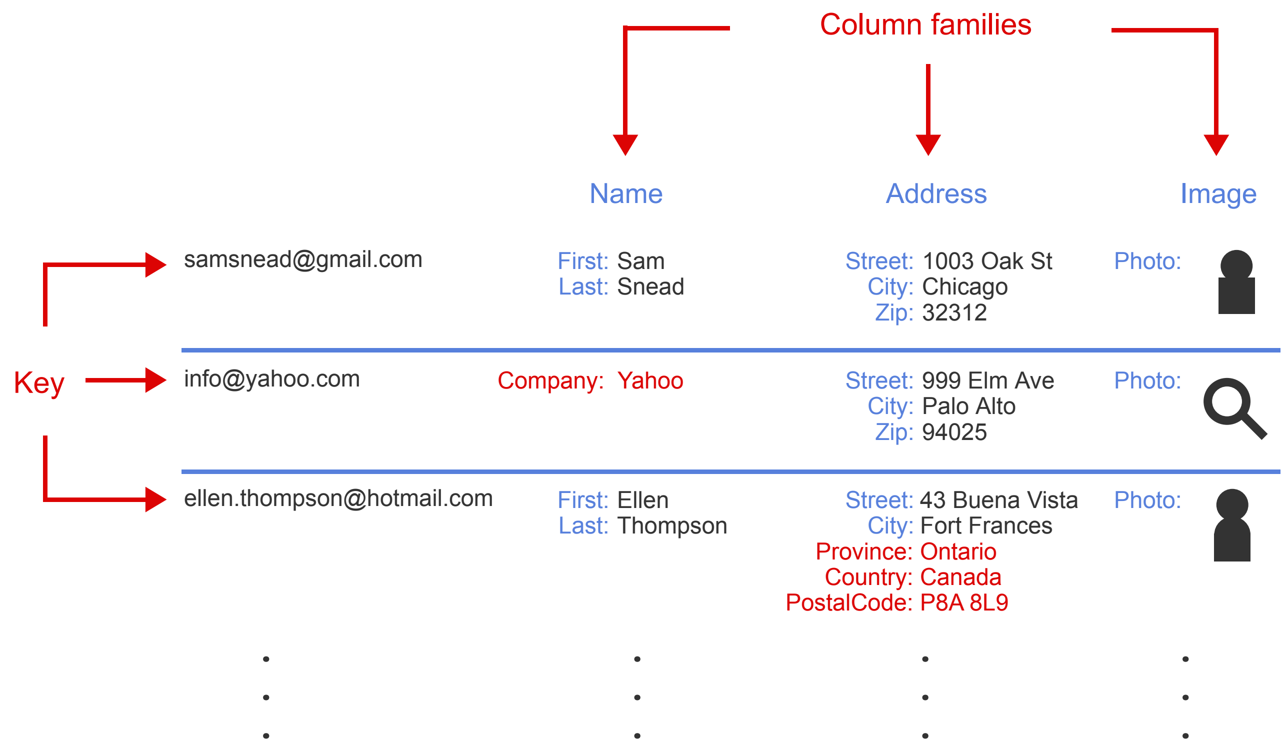
Wide column databases store multiple versions of each value. Each version is marked with the date and time the version is created, called a ***timestamp***. To access older values, the timestamp must be specified in a query. If a query does not specify a timestamp, the database selects the most recent version.

In a wide column database, a specific value is accessed with a combination of table name, key, column family name, column name, and optional timestamp.

Table 23.3.1: Database models and keys.

Database model	Required to access value
Relational	Table name Primary key Column name
Key-value	Key
Wide column	Table name Key Column family name Column name Timestamp (optional)

PARTICIPATION ACTIVITY 23.3.1: Wide column logical structure.



Wide column logical structure

Animation content:

Static figure:
The diagram contains three rows of wide column data. Each row has a key and three column families. The key is an email address. The column families are Name, Address, and Image. Each column family of each row consists of name:value pairs.

In the first row, the key is samsnead@gmail.com.
The name:value pairs for Name are
First: Sam
Last: Snead
The name:value pairs for Address are:
Street: 1003 Oak St
City: Chicago
Zip: 32312
The name:value pairs for Image are:
Photo: [a person icon]

In the second row, the key is info@yahoo.com.

The name:value pairs for Name are
Company: Yahoo
The name:value pairs for Address are:
Street: 999 Elm Ave
City: Palo Alto
Zip: 94025
The name:value pairs for Image are:
Photo: [a logo icon]

In the third row, the key is ellenthompson@hotmail.com.
The name:value pairs for Name are
First: Ellen
Last: Thompson
The name:value pairs for Address are:
Street: 43 Buena Vista
City: Fort Frances
Province: Ontario
Country: Canada
PostalCode: P8A 8L9
The name:value pairs for Image are:
Photo: [a person icon]

Variations in name:value pairs within one column family are highlighted. In the second row, Company: Yahoo is highlighted. In the third row, Province: Ontario, Country: Canada, and PostalCode: P8A 8L9 are highlighted.

Animation captions:

1. The key identifies rows, as in a key-value database.
2. Column families are fixed across all rows.
3. Each column family contains one or many columns. Each column has a name and a value.
4. Columns can vary across rows.

PARTICIPATION ACTIVITY | 23.3.2: Wide column logical structure.

The following questions refer to the animation above.

1) What information is required to access Sam Snead's ZIP code?

○ Key 'samsnead@gmail.com'

- Table name 'People'
  Key 'samsnead@gmail.com'
  Column name 'Zip'
- Table name 'People'
  Key 'samsnead@gmail.com'
  Column family name 'Address'
  Column name 'Zip'

2) A new row can contain:

- A new column family 'Employment' with a new column 'CompanyName'.
- A new column 'Suffix' in the 'Name' column family.
- No new columns or column families.

3) When Yahoo changes its corporate address:

- The old address is deleted and only the new address is stored.
- Both old and new addresses are stored along with timestamps.
- Both old and new addresses are stored with a new column named 'Current', indicating the most recent address.

4) In comparison to key-value databases, the wide column data structure is:

- Simpler.
- More complex.
- About the same complexity.

Physical structure

Wide column and key-value databases are similar in several respects:

Wide column and key-value databases are similar in several respects:

- *Fast read and write*. Rather than update an existing value, a new version is saved.
- *High volume*. Both key-value and wide column databases support very large numbers of rows.

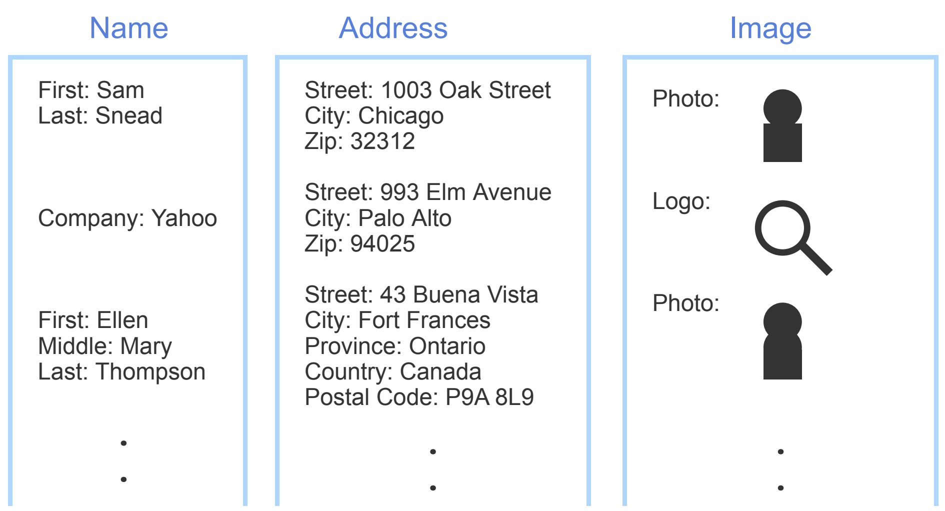
Wide column databases differ from key-value databases in other respects:

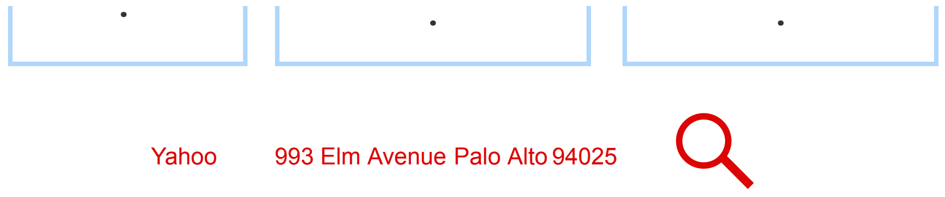
- *Complex data model*. In a key-value database, data is stored as a value. In a wide column database, data is structured in multiple columns, grouped in column families.
- *Flexible schema*. Column families are fixed, but columns within each family are not. Consequently, wide column schema are more flexible than relational schema but less flexible than key-value schema.
- *Data access*. In a key-value database, the key is sufficient to identify and locate a value. A wide column database requires table name, key, column family name, column name, and optional timestamp.

Like most NoSQL databases, wide column databases utilize replication with eventual consistency, sharding, and horizontal scaling. Rows are physically sorted on the key. Each shard contains rows with a range of key values. Ex: Rows with keys beginning with the letter 'a' might be stored on one shard, rows with key beginning with 'b' might be on another shard, and so on.

Within each row, all columns of a family are stored contiguously. This organization is optimal for reading and writing all columns within a family and all rows within a range of key values. Wide column databases are not optimized for reading several column families in the same query, since different column families are not stored together.

PARTICIPATION ACTIVITY | 23.3.3: Wide column physical structure.





Wide column physical structure

Animation content:

Static figure:
A diagram illustrates the physical organization of wide column data. The key is an email address. The column families are Name, Address, and Image. The data is the same as the prior animation. The diagram has three rectangles with captions Name, Address, and Image. Each rectangle contains column family data for all rows.

The Name rectangle contains these name:value pairs:
First: Sam Last: Snead
Company: Yahoo
First: Ellen Last: Thompson

The Address rectangle contains these name:value pairs:
Street: 1003 Oak St City: Chicago Zip: 32312
Street: 999 Elm Ave City: Palo Alto Zip: 94025
Street: 43 Buena Vista City: Fort Frances Province: Ontario Country: Canada PostalCode: P8A 8L9

The Image rectangle contains these name:value pairs:
Photo: [a person icon]
Photo: [a logo icon]
Photo: [a person icon]

Animation captions:

1. All columns of a family are stored together for fast access via the key.
2. Different column families are physically separated.
3. Wide column databases are not optimized to access multiple column families within one query.

PARTICIPATION ACTIVITY | 23.3.4: Wide column physical structure.

1) Wide column databases are suitable for more complex applications than key-value databases.

- True
- False

2) Referring to the schema in the animation above, the wide column database is optimized to query the names and images of all people.

- True
- False

3) Wide column databases are optimized for vertical scaling.

- True
- False

Database systems

Leading wide-column databases include:

- *BigTable*. The wide column model was developed at Google and published in a 2006 article. Consequently, Google BigTable was the first wide column database. BigTable now supports many Google services, including search, maps, and email. The wide column model is ideal for these services, since web pages, maps, and email have massive data volumes yet more structure than supported by key-value database.
- *HBase*. HBase is an open source database sponsored by Apache. HBase is based on BigTable and has a similar structure. However, BigTable is available as a Google cloud service only, while HBase is available as open source. Unlike most NoSQL databases, HBase has a master component that coordinates updates and ensures immediate consistency across data replicas.
- *Cassandra*. Like HBase, Cassandra is sponsored by Apache and is available as open source. Cassandra is similar to HBase, with one major difference: Cassandra has no master component to coordinate updates to replicas. Consequently, Cassandra has no single point of failure and is always available. In exchange for high availability, Cassandra supports eventual rather than immediate consistency.

Key-value databases commonly support additional database models, since the key-value model is limited. Because the wide column model has more structure, most wide column databases do not support additional models.

Table 23.3.2: Wide column database systems.

	Developer	Initial release	Database models	License	DB-Engines rank (May 2020)
Cassandra	Apache Software Foundation	2008	Wide column	Open source	11
HBase	Apache Software Foundation	2008	Wide column	Open source	22
BigTable	Google	early 2000s (internal) 2015 (commercial)	Wide column	Cloud	103

PARTICIPATION ACTIVITY 23.3.5: Wide column database systems.

If unable to drag and drop, refresh the page.

Cassandra **HBase** **BigTable**

	The first wide column database system, based on a research paper published in 2006.
	An open source database based on BigTable.
	According to db-engines.com, the highest-ranked wide column database in 2019.

Reset

CHALLENGE ACTIVITY | 23.3.1: Wide column databases.

544874.3500394.qx3zqy7

Start

Table: Contact

Key	Column family Name	Column family Address	
ajf@acm.org	First: Arnold Middle: J Last: Fourier	POBox: 2243 City: Boise State: ID	Categ Statu
info@neaq.org	Organization: NE Aquarium	Location: Central Wharf State: MA City: Boston	Notes Categ
sales@corp.com	CompanyName: Corp, Inc.	Mailstop: 3A Street: 900 4th St. City: Denver State: CO	Categ Produ

What information is needed to get the 'Category' of 'NE Aquarium'?

Table name	Pick
Key	Pick
Column family name	Pick
Column name	Pick

1 | 2

Check   Next

Exploring further:

- Google BigTable research paper

- Google BigTable
- Apache HBase
- Apache Cassandra

23.4 Document databases

Logical structure

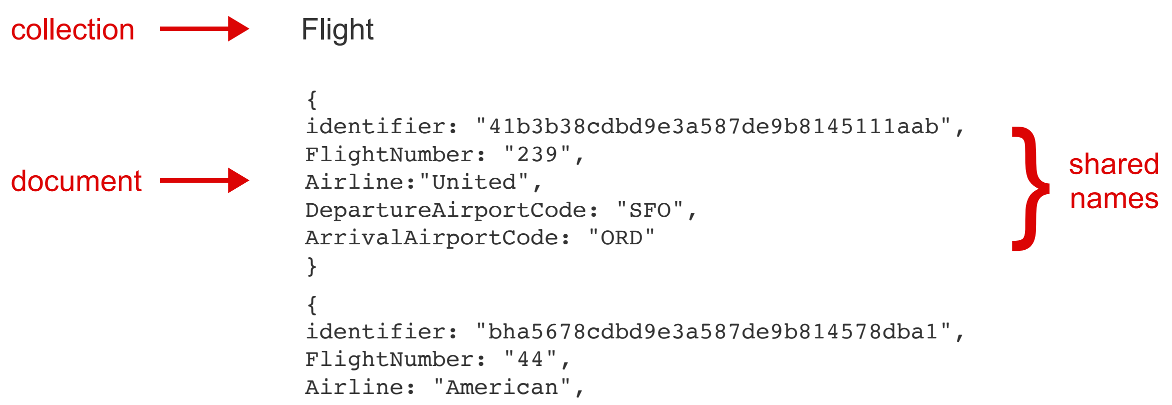
A document database stores data as a ***collection*** of documents. A document database may contain multiple collections, just as a relational database may contain multiple tables.

The term ***document*** is misleading, since a document is not a traditional text document. Each document is represented in XML, JSON, or a similar format. The specific format is fixed by the database system and is the same for all documents. Each document contains one or more values. Each value is associated with a name, specified explicitly in the document. In addition to the values, each document contains a unique identifier, similar to a key in other database models.

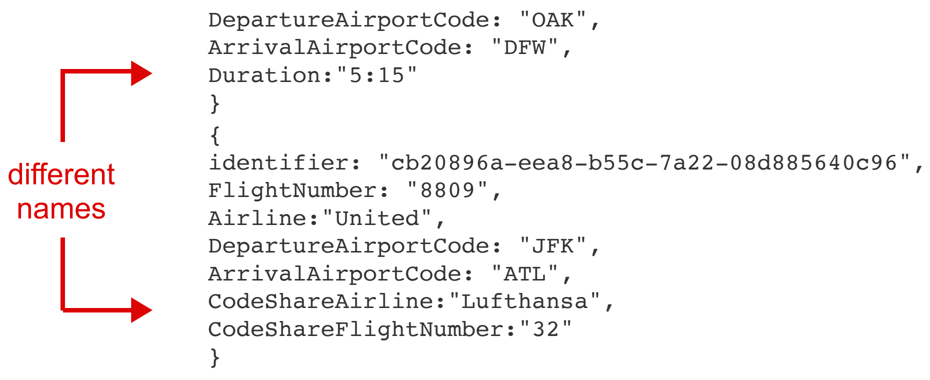
The number and names of values may vary across documents within a collection. Consequently, the document database schema is flexible. Typically, all documents in a collection share several common names to facilitate queries across the collection.

Queries can access documents via an identifier, as in other database models. In addition, queries can access documents based on other values, as in the WHERE clause of an SQL statement. In fact, some document databases support SQL or a similar query language.

PARTICIPATION ACTIVITY | 23.4.1: Document database logical structure.



```
{
identifier: "41b3b38cdbd9e3a587de9b8145111aab",
FlightNumber: "239",
Airline:"United",
DepartureAirportCode: "SFO",
ArrivalAirportCode: "ORD"
}
{
identifier: "bha5678cdbd9e3a587de9b814578dba1",
FlightNumber: "44",
Airline: "American",
```



```
DepartureAirportCode: "OAK",
ArrivalAirportCode: "DFW",
Duration:"5:15"
}
{
identifier: "cb20896a-eea8-b55c-7a22-08d885640c96",
FlightNumber: "8809",
Airline:"United",
DepartureAirportCode: "JFK",
ArrivalAirportCode: "ATL",
CodeShareAirline:"Lufthansa",
CodeShareFlightNumber:"32"
}
```

Document database logical structure

Animation content:

Static figure:
A collection of documents is named Flight. Flight appears in JSON format:
Begin JSON code:
{
identifier: "41b3b38cdbd9e3a587de9b8145111aab",
FlightNumber: "239",
Airline:"United",
DepartureAirportCode: "SFO",
ArrivalAirportCode: "ORD"
}
{
identifier: "bha5678cdbd9e3a587de9b814578dba1",
FlightNumber: "44",
Airline: "American",
DepartureAirportCode: "OAK",
ArrivalAirportCode: "DFW",
Duration:"5:15"
}
{
identifier: "cb20896a-eea8-b55c-7a22-08d885640c96",
FlightNumber: "8809",
Airline:"United",
DepartureAirportCode: "JFK",
ArrivalAirportCode: "ATL",
CodeShareAirline:"Lufthansa",
CodeShareFlightNumber:"32"
}

End JSON code.

Each group of name:value pairs within curly braces is a document, so Flight contains three documents. A caption indicates that names FlightNumber, Airline, DepartureAirportCode, and ArrivalAirportCode appear in all three documents. Another caption indicates that Duration appears only in the second document, and CodeShareAirline, and CodeShareFlightNumber appear only in the third document.

Animation captions:

1. The Flight collection consists of documents describing scheduled airline flights, in JSON format.
2. Documents may have a different number of values with different names.
3. Usually all documents in a collection share common value names to facilitate queries.

PARTICIPATION ACTIVITY | 23.4.2: Document database logical structure.

1) In the animation above, what is the information within the curly braces {} called?

- ○ row
- ○ document
- ○ object

2) What data type is commonly used to represent documents?

- ○ Binary
- ○ JSON
- ○ ASCII

3) In the example above, 'FlightNumber' is

- ○ the document identifier.
- ○ a document value.
- ○ the name of a document value.

4) Which of the following expressions select all three documents in the

above animation?

- FlightNumber > 30
- CodeShareFlightNumber > 30
- Fare > 0

Physical structure

Like other NoSQL databases, document databases support replication, sharding, horizontal scaling, and eventual consistency. Unlike many NoSQL databases, document databases support indexes on non-key values, as long as the value is present in all documents of a collection.

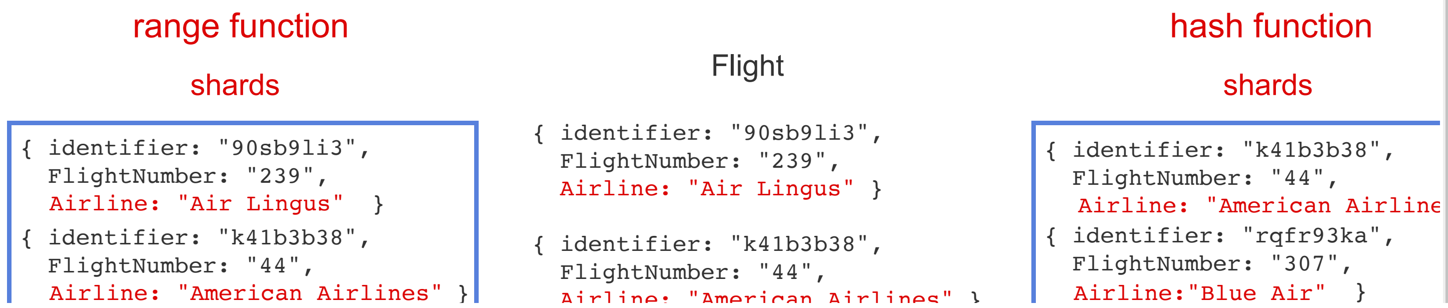
Documents are assigned to a shard based on a ***shard key***. The shard key is either the document identifier or some other value. To quickly locate the shard containing a document, a hash or a range function is applied to the document's shard key. Since hash and range functions are fast, and shards are relatively small, the shard key enables rapid insertion and retrieval of documents.

Hash functions are described elsewhere in this material. With a ***range function***, each shard contains a contiguous range of shard key values. Ex: If the shard key for the Flight collection is Airline, documents for airlines beginning with 'A' might be in one shard, 'B' in another, and so on.

Some document databases support strict consistency. All updates are first applied to a ***primary replica*** and subsequently propagated to ***secondary replicas***. Strict consistency is achieved by directing all read queries to the primary replica, ensuring the latest document version is returned. A secondary replica becomes active only if the primary replica fails.

Compared to eventual consistency, strict consistency provides slower response time and reduced availability. Consequently, document databases that support strict consistency can also be configured for eventual consistency. With eventual consistency, no primary replica is specified. Updates are directed to any replica and propagated asynchronously to all other replicas. Reads are directed to any replica and potentially return out-of-date results.

PARTICIPATION ACTIVITY | 23.4.3: Assigning documents to shards.

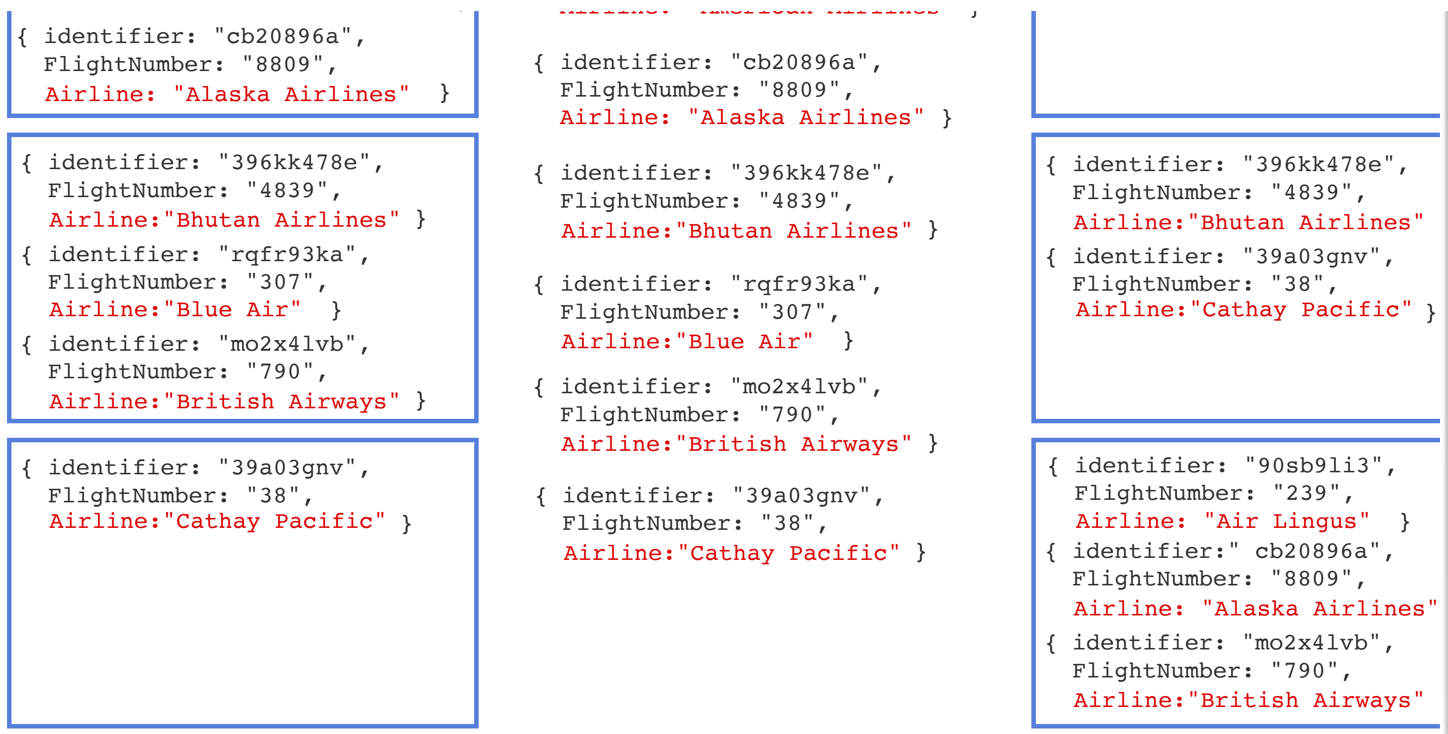


```
{ identifier: "90sb9li3",
  FlightNumber: "239",
  Airline: "Air Lingus"  }
{ identifier: "k41b3b38",
  FlightNumber: "44",
  Airline: "American Airlines" }
```

```
{ identifier: "90sb9li3",
  FlightNumber: "239",
  Airline: "Air Lingus" }

{ identifier: "k41b3b38",
  FlightNumber: "44",
  Airline: "American Airlines" }
```

```
{ identifier: "k41b3b38",
  FlightNumber: "44",
   Airline: "American Airline
{ identifier: "rqfr93ka",
  FlightNumber: "307",
  Airline:"Blue Air"  }
```



```
{ identifier: "cb20896a",
  FlightNumber: "8809",
  Airline: "Alaska Airlines"  }

{ identifier: "396kk478e",
  FlightNumber: "4839",
  Airline:"Bhutan Airlines" }
{ identifier: "rqfr93ka",
  FlightNumber: "307",
  Airline:"Blue Air"  }
{ identifier: "mo2x4lvb",
  FlightNumber: "790",
  Airline:"British Airways" }

{ identifier: "39a03gnv",
  FlightNumber: "38",
  Airline:"Cathay Pacific" }
```

```
{ identifier: "cb20896a",
  FlightNumber: "8809",
  Airline: "Alaska Airlines" }

{ identifier: "396kk478e",
  FlightNumber: "4839",
  Airline:"Bhutan Airlines" }

{ identifier: "rqfr93ka",
  FlightNumber: "307",
  Airline:"Blue Air"  }

{ identifier: "mo2x4lvb",
  FlightNumber: "790",
  Airline:"British Airways" }

{ identifier: "39a03gnv",
  FlightNumber: "38",
  Airline:"Cathay Pacific" }
```

```
{ identifier: "396kk478e",
  FlightNumber: "4839",
  Airline:"Bhutan Airlines"
{ identifier: "39a03gnv",
  FlightNumber: "38",
  Airline:"Cathay Pacific" }

{ identifier: "90sb9li3",
  FlightNumber: "239",
  Airline: "Air Lingus"  }
{ identifier:" cb20896a",
  FlightNumber: "8809",
  Airline: "Alaska Airlines"
{ identifier: "mo2x4lvb",
  FlightNumber: "790",
  Airline:"British Airways"
```

Document database physical structure.

Animation content:

Static figure:
The Flight collection contains seven documents in JSON format:
Begin JSON code:

```
{ identifier: "90sb9li3",
  FlightNumber: "239",
  Airline:"Air Lingus"}
{ identifier: "k41b3b38",
  FlightNumber: "44",
  Airline:"American Airlines"}
{ identifier: "cb20896a",
  FlightNumber: "8809",
  Airline:"Alaska Airlines"}
{ identifier: "396kk478e",
  FlightNumber: "4839",
  Airline:"Bhutan Airlines"}
{ identifier: "rqfr93ka",
  FlightNumber: "307",
  Airline:"Blue Air"}
{ identifier: "mo2x4lvb",
  FlightNumber: "790",
```

Airline:"British Airways"}
{ identifier: "39a03gnv",
FlightNumber: "38",
Airline:"Cathay Pacific"}
End JSON code.

Three shards appear to the right of Flight with caption hash function. Shard 1 contains the American Airlines and Blue Air documents. Shard 2 contains the Bhutan Airlines and Cathay Pacific documents. Shard 3 contains the Air Lingus, Alaska Airlines, and British Airways documents.

Three more shards appear to the left of Flight with caption range function. Shard 1 contains the Air Lingus, American Airlines, and Alaska Airlines documents. Shard 2 contains the Bhutan Airlines, Blue Air, and British Airways documents. Shard 3 contains the Cathay Pacific document.

Step 1: The database designer selects either the identifier or an indexed value as the shard key. Airline is chosen as the shard key. The Flight document appears. The Airline name:value pairs are highlighted with caption shard key.

Step 2: Documents can be assigned to a shard based on a hash function on the shard key. The hash function shards appear. Documents appear in shards based on a hash function of the airline name.

Step 3: Alternatively, documents can be assigned to a shard based on a range function. The range function shards appear. Documents for airline names beginning with A appear in shard 1. Documents for airline names beginning with B appear in shard 2. The document for the airline name beginning with C appears in shard 3.

Animation captions:

1. The database designer selects either the identifier or an indexed value as the shard key. Airline is chosen as the shard key.
2. Documents can be assigned to a shard based on a hash function on the shard key.
3. Alternatively, documents can be assigned to a shard based on a range function.

PARTICIPATION ACTIVITY 23.4.4: Document database physical structure.

1) In a document database, the _______ can be either the document identifier or another

value.

Check Show answer

2) A _______ assigns contiguous ranges of shard key values to each shard.

Check Show answer

3) If the shard key is a value other than the document identifier, an _______ must be created on the value.

Check Show answer

Database systems

Leading document databases include:

- *MongoDB* is the most widely-used document database, ranked first among document databases by DB-Engines. MongoDB stores data in a BSON format, which stands for binary JSON. Unlike JSON, BSON is optimized for fast performance and is not human-readable. MongoDB can be configured for either strict or eventual consistency. MongoDB supports SQL read queries, but not SQL create, update, or delete.
- *CouchDB* is an open source database sponsored by Apache. CouchDB was the first document database, initially released in 2005. CouchDB stores data in JSON format and supports eventual but not strict consistency. CouchDB was designed to support mobile devices, which occasionally go offline. Revision information, such as date and time, is stored with updated data. When an offline device comes back online, the revision information is used to merge updated data with other replicas.
- *Couchbase* was derived from CouchDB and Membase, an open-source database released in 2010. Since CouchDB is a document database and Membase was a key-value database,

Couchbase supports both document and key-value models. Couchbase supports SQL++, an extension of SQL that supports JSON data, and both eventual and strict consistency.

According to the DB-Engines 2019 ranking, document database is the most popular NoSQL category, and MongoDB is the most popular NoSQL database, by a wide margin. However, the same ranking shows MongoDB well behind relational databases such as Oracle, SQL Server, and MySQL.

Table 23.4.1: Document database systems.

	Developer	Initial release	Database models	License	DB-Engines rank (May 2020)
MongoDB	MongoDB	2009	Document	Cloud Commercial Open source	5
CouchDB	Apache Software Foundation	2005	Document	Open source	35
Couchbase	Couchbase	2011	Document Key-value	Cloud Commercial Open source	25

PARTICIPATION ACTIVITY 23.4.5: Document database systems.

If unable to drag and drop, refresh the page.

Couchbase **CouchDB** **MongoDB**

Designed to support mobile devices and offline updates with eventual consistency.

	Supports both key-value and document database models.
	Stores documents in BSON format, a variation of JSON optimized for fast read and write.

Reset

CHALLENGE ACTIVITY 23.4.1: Document databases.

544874.3500394.qx3zqy7

Start

Select each document that matches the expression: Class > 2019

Documents in collection Contact

☐
```
{ identifier: "crf8gwj9",
  Class: 2023,
  FirstName: "Abe",
  LastName: "Cruz"    }
```

☐
```
{ identifier: "dk615cy2",
  FirstName: "Ron",
  LastName: "Roy",
  Major: "Computer Science",
  Degree: "BSc",
  Class: 2004    }
```

☐
```
{ identifier: "etrj1wg4",
  FirstName: "Mai",
  LastName: "Ford",
  Class: 2002,
  Major: "Mathematics",
  Degree: "BA"    }
```

☐
```
{ identifier: "a9djqp11",
  Class: 2023,
  FirstName: "Eli",
  LastName: "King",
  Major: "Philosophy",
  Minor: "Mathematics",
  State: "MI",
  GPA: 3.1    }
```

☐
```
{ identifier: "bjndx712",
  FirstName: "Ada",
  LastName: "Page",
  Class: 2024,
  Major: "Mathematics",
  State: "PA",
  GPA: 3.72    }
```

☐ No documents are selected

1	2	3

Check Next

Exploring further:

- MongoDB
- Apache CouchDB
- Couchbase

23.5 Graph databases

Graphs

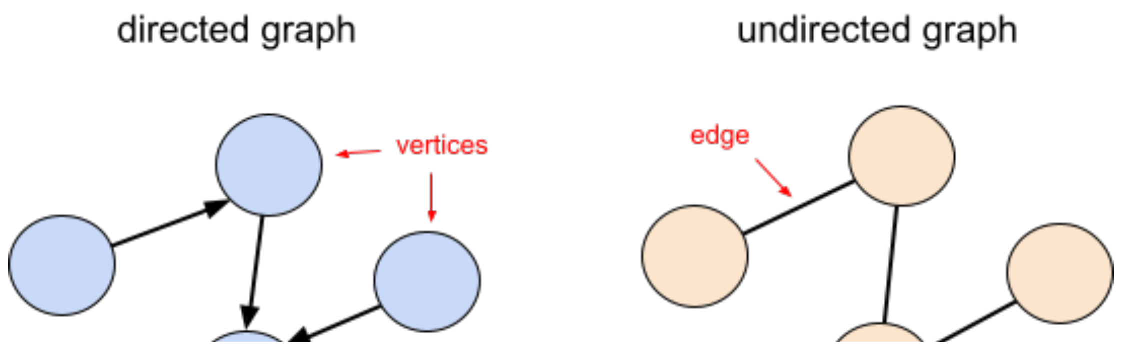
The graph database model is based on graph theory. In graph theory, a graph is a network of vertices and edges:

- A ***vertex***, also called a ***node***, is a hub where network lines converge.
- An ***edge***, also called a ***link***, is a connection between two vertices.
- A ***property*** is descriptive information associated with vertices and edges.

Graphs are depicted as drawings, with vertices as points, circles, or rectangles, and edges as lines or arrows.

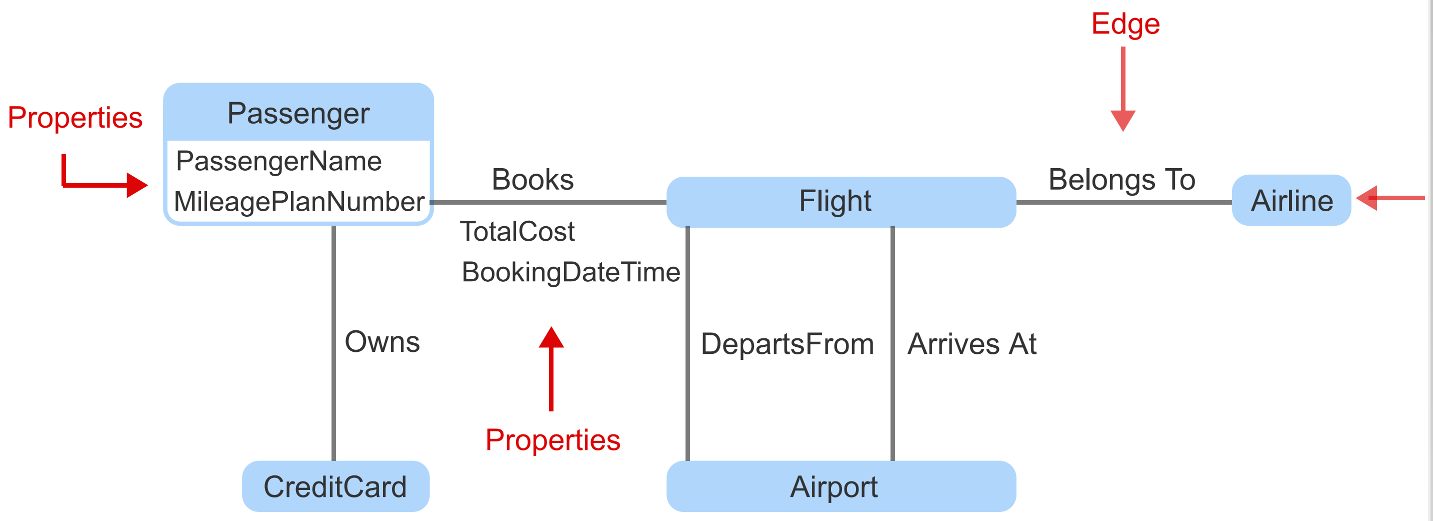
Graphs are either directed or undirected. In a ***directed graph***, edges have a starting and ending vertex and are depicted as arrows. In an ***undirected graph***, edges have no direction and are depicted as lines.

Figure 23.5.1: Vertices and edges in a directed and undirected graph.



An entity-relationship model, described elsewhere in this material, is an undirected graph. Entities are vertices, relationships are edges, and attributes are properties.

PARTICIPATION ACTIVITY | 23.5.1: Graphs are similar to entity-relationship models.



Animation content:

Static figure:
An undirected graph has vertices Passenger, Flight, Airline, CreditCard, and Airport. The graph has five undirected edges:
Passenger-Books-Flight
Flight-BelongsTo-Airline
Passenger-Owns-CreditCard
Flight-DepartsFrom-Airport
Flight-ArrivesAt-Airport

The Passenger vertex has properties PassengerName and MileagePlanNumber. The Books edge has properties TotalCost and BookingDateTime.

Animation captions:

1. A vertex is like an entity. An edge is like a relationship.

2. In an undirected graph, edges have no direction and are drawn as lines.
3. In a directed graph, edges have a start and end vertex and are drawn as arrows.
4. Vertices and edges can have properties.

PARTICIPATION ACTIVITY | 23.5.2: Graphs.

Refer to the graph in the animation above.

1) ' CreditCard' is a(n) _______.

Check Show answer

2) ' Owns' is a(n) _______.

Check Show answer

3) 'TotalCost' is a(n) _______.

Check Show answer

4) This is a(n) _______ graph.

Check Show answer

Logical structure

Relational databases implement a standard relational model, with limited variations and extensions. Since the graph database model is not standardized, logical structure and terminology varies from one database to the next.

The most common type of graph database is a ***property graph***. The property graph logical structure is a directed graph.

- Vertices, sometimes called nodes, are the main data items, similar to rows in a relational table.
- Edges, sometimes called relationships, have a start and end vertex.
- Properties are name-value pairs. The name is like a relational column name, and the value is like data in the column. Properties may be associated with both vertices and edges.
- Vertices and edges have zero, one, or many labels. A ***vertex label*** is a collection of similar objects, similar to a relational table. An ***edge label*** is a collection of relationships between vertices.

Like other NoSQL databases, graph databases have a flexible schema. New vertices, edges, and properties can be added at any time. Property names are stored with property values, so vertices and edges with the same label can have different property names.

Table 23.5.1: Property graph terminology.

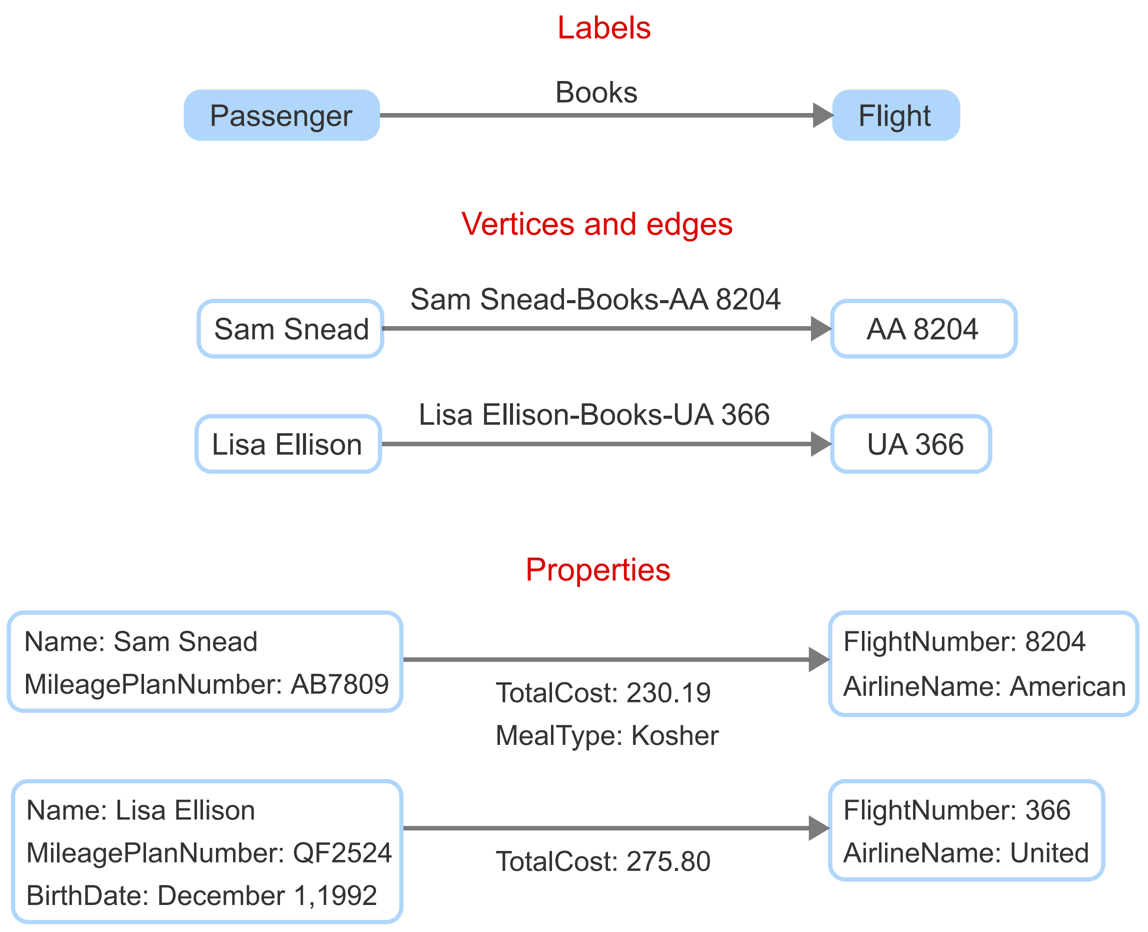
Entity-relationship term	Property graph term
Entity type	Label
Entity instance	Vertex or node
Relationship type	Label
Relationship instance	Edge or relationship
Attribute type	Property name
Attribute instance	Property value

A less common type of graph database is a ***triple store***, also known as a ***resource description framework*** (RDF) database. In a triple store, every property is represented as a separate vertex. Triple stores have a simpler logical structure than property graphs but more vertices, since every property is a vertex.

Property graph databases are commonly used for transactional applications. Triple store databases are commonly used for analytic applications.

PARTICIPATION

ACTIVITY 23.5.3: Property graph logical structure.



Animation content:

Static figure:
Three diagrams appear, with captions labels, vertices and edges, and properties.

The labels diagram has two vertex labels, Passenger and Flight, connected by a directed edge label, Books. Labels appear in rectangles with rounded corners and solid fill.

The vertices and edges diagram shows vertices and edges for these labels. The diagram has two vertices for vertex label Passenger:
Sam Snead
LIsa Ellison
The diagram has two vertices for vertex label Flight:
AA 8204
UA 366.
The diagram has two directed edges for directed edge label Books:

Sam Snead-Books-AA8204
Lisa Ellison-Books-UA 366
Vertices appear in rectangles with rounded corners and no fill. Directed edges appear as arrows.

The properties diagram shows properties for each of these vertices and edges.
Vertex Sam Snead has properties:
Name: Sam Snead
MileagePlanNumber: AB7809
Vertex Lisa Ellison has properties:
Name: Lisa Ellison
MileagePlanNumber: QF2524
BirthDate: December 1, 1992
Vertex AA 8204 has properties:
FlightNumber: 8204
AirlineName: American
Vertex UA 366 has properties:
FlightNumber: 366
AirlineName: United
Edge Sam Snead-Books-AA8204 has properties:
TotalCost: 230.19
MealType: Kosher
Edge Lisa Ellison-Books-UA 366 has property:
TotalCost: 275.80
Properties appear inside vertex rectangles and below edge lines.

Animation captions:

1. Vertex labels are collections of objects, like entity types.
2. A vertex is an individual object, like an entity instance. An edge is a relationship between individual objects.
3. Properties of vertices and edges are name-value pairs.
4. Property graphs have flexible schema. Different vertices and edges can have different properties.

PARTICIPATION ACTIVITY 23.5.4: Graph database logical structure.

Refer to the animation above.

1) How many vertices does the database contain?

- ○ 2
- ○ 4
- ○ 6

2) Can a vertex without the 'MileagePlanNumber' property be added to the database?

- ○ Yes
- ○ No

3) Name an end vertex of the Books edge.

- ○ Lisa Ellison
- ○ Flight
- ○ UA 366

Query languages

All relational databases support SQL, with limited variations and extensions. Graph databases support a variety of query languages:

- *Cypher* was developed by the vendor of Neo4j, the leading property graph database, and is widely supported by other property graph databases.
- *Gremlin* is sponsored by the Apache Software Foundation. Like Cypher, Gremlin supports property graph databases and is implemented in many database products.
- *SPARQL*, pronounced 'sparkle', is primarily used by triple store (RDF) databases.

In addition to the established graph query languages above, standards organizations are developing two new languages:

- *SQL/Property Graph Query* (SQL/PQG) is an extension to SQL, sponsored by the same standards organization as SQL.
- *Graph Query Language* (GQL) is heavily influenced by Cypher. GQL is sometimes confused with GraphQL, a different query language initially developed by Facebook and released as open source in 2015.

Both languages are designed for property graph databases. Both languages are new and evolving, with limited database support as of 2020.

PARTICIPATION ACTIVITY

23.5.5: Gremlin query language.

Queries

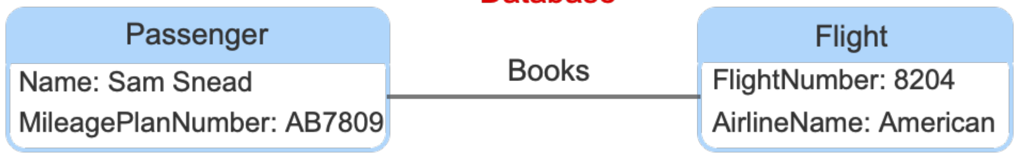
```
g.addV('Passenger').property('Name', 'Sam Snead')
                   .property('MileagePlanNumber', 'AB7809')

g.addV('Flight').property('FlightNumber', '8204')
                .property('AirlineName', 'American')

g.V('Sam Snead').addE('Books').to(g.V('8204'))

g.V('Sam Snead').out('Books')
```

Database



Result

Sam Snead AB7809 8204 American

Animation content:

Static figure:
A Gremlin code fragment has caption queries:
Begin Gremlin code:
g.addV('Passenger').property('Name', 'Sam Snead').property('MileagePlanNumber', 'AB7809')
g.addV('Flight').property('FlightNumber', '8204').property('AirlineName', 'American')
g.V('Sam Snead').addE('Books').to(g.V('8204'))
g.V('Sam Snead').out('Books')
End Gremlin code.

A graph diagram appears with label database. The diagram has vertex labels Passenger and Flight, and edge label Passenger-Books-Flight.
Passenger has properties:
Name: Sam Snead
MileagePlanNumber: AB7809

Flight has properties:
FlightNumber: 8204
AirlineName: American

Text with caption result shows the result of the Gremlin code:
Sam Snead AB7809 8204 American

Step 1: g.addV().property() adds a vertex with label 'Passenger' to graph g. The first Gremlin statement appears:
g.addV('Passenger').property('Name', 'Sam Snead').property('MileagePlanNumber', 'AB7809')
The Passenger vertex appears with properties in the database diagram.

Step 2: g.addV().property() adds a vertex with label 'Flight' to graph g. The second Gremlin statement appears:
g.addV('Flight').property('FlightNumber', '8204').property('AirlineName', 'American')
The Flight vertex appears with properties in the database diagram.

Step 3: g.V().addE().to() adds an edge between two vertices. The third Gremlin statement appears:
g.V('Sam Snead').addE('Books').to(g.V('8204'))
The Books edge appears in the database diagram.

Step 4: out() traverses edges from start to end vertex, like a relational join. The fourth Gremlin statement appears:
g.V('Sam Snead').out('Books')
The Passenger and Flight properties are highlighted. The result appears:
Sam Snead AB7809 8204 American

Animation captions:

1. g.addV().property() adds a vertex with label 'Passenger' to graph g.
2. g.addV().property() adds a vertex with label 'Flight' to graph g.
3. g.V().addE().to() adds an edge between two vertices.
4. out() traverses edges from start to end vertex, like a relational join.

PARTICIPATION ACTIVITY | 23.5.6: Query languages.

1) In Gremlin, what operation is similar to a relational join?

- ○ addV()
- ○ hasLabel()
- ○ out()

2) In Gremlin, what operation is similar to a relational INSERT?
   - ○ addV()
   - ○ hasLabel()
   - ○ out()

3) In Gremlin, what is the first component of a query?
   - ○ Vertex label
   - ○ Graph name
   - ○ addV()

4) Which language is designed for triple store databases?
   - ○ Cypher
   - ○ Gremlin
   - ○ SPARQL

Physical structure

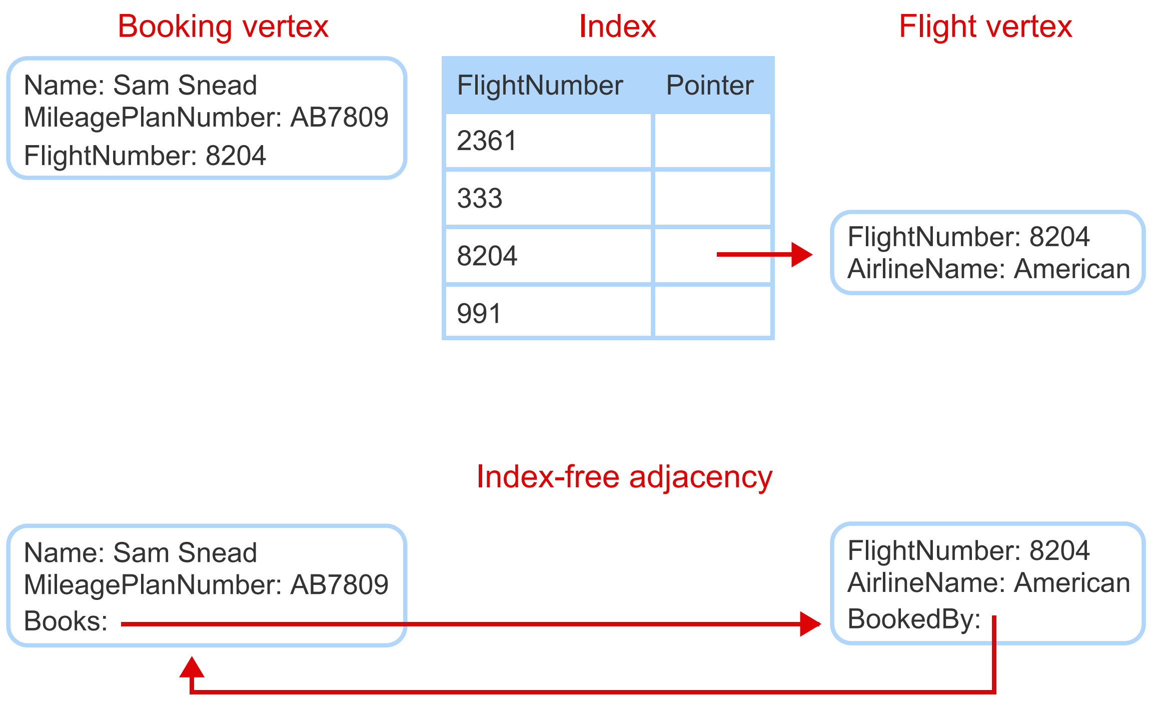
Physical structures vary greatly across graph databases but fall into two major categories:

- *Native*. Native physical structures are optimized for graph models and typically support index-free adjacency. ***Index-free adjacency*** means that each vertex contains a pointer, or physical address, to all connected vertices. When a vertex is accessed, the location of related vertices is immediately available. Consequently, queries that traverse related vertices do not use indexes and are faster than relational joins.
- *Layered*. Some graph databases layer a graph model on an alternative database model. Ex: A property graph might be implemented on top of a relational database, with vertices stored as rows and edges stored as foreign keys. A triple store might be implemented on a relational database as a three-column table, with the first and third column storing the related vertices and the second column storing the edge between the vertices. Alternatively, property graphs and triple stores might be implemented on top of document or key-value databases.

A layered physical structure is not optimized for graph query languages like Cypher and Gremlin.

Layered structure is often implemented in multi-model databases, which support several NoSQL models on top of a single physical structure.

PARTICIPATION ACTIVITY 23.5.7: Index-free adjacency.



Animation content:

Static figure:
Two diagrams appear, with captions index and index-free adjacency.

The index diagram shows an index between a Booking vertex and a Flight vertex. The Booking vertex has three properties, including FlightNumber: 8204. The Flight vertex has two properties, including FlightNumber: 8204. The index contains flight numbers and pointers. Index entry 8204 points to the Flight vertex with FlightNumber: 8204.

The index-free adjacency diagram shows the same Booking and Flight vertices, but no index. Instead of property FlightNumber: 8204, the Booking vertex has a property with name Books and a pointer to the Flight vertex. In addition to FlightNumber and AirlineName properties, the Flight vertex has a property with name BookedBy and a pointer to the Flight vertex.

Static figure:

Step 1: With an index, a relationship is stored as a value along with a pointer to the related vertex. In the index diagram, property FlightNumber: 8204 of Booking is highlighted. The index entry for flight number 8204 is highlighted. The pointer for this index entry is highlighted, and points to the Flight vertex with property FlightNumber: 8204.

Step 2: With index-free adjacency, a pointer is stored in the start vertex. Queries that traverse edges require fewer reads. The index-free adjacency diagram appears. Books: [pointer] replaces the FlightNumber: 8204 property. The pointer points to the Flight vertex.

Step 3: A pointer is also stored in the end vertex to enable traversal in any direction. The BookedBy: [pointer] property is added to the Flight vertex. The pointer points to the Booking vertex.

Animation captions:

1. With an index, a relationship is stored as a value along with a pointer to the related vertex.
2. With index-free adjacency, a pointer is stored in the start vertex. Queries that traverse edges require fewer reads.
3. A pointer is also stored in the end vertex to enable traversal in any direction.

PARTICIPATION ACTIVITY 23.5.8: Graph database physical structure.

1) Can graph databases have indexes?

- ○ Yes
- ○ No

2) What is a disadvantage of index-free adjacency as compared to indexes?

- ○ Index-free adjacency has no disadvantages.
- ○ Restructuring data storage may take longer.
- ○ Programmers must hard-code the physical location of vertices.

3) Relational databases typically

implement index-free adjacency.

- ○ Yes
- ○ No

Database systems

Leading graph database systems include:

- *Neo4j*, from the company of the same name, is by many measures the most popular property graph database. The Cypher query language was developed by Neo4j and has been adopted by other property graph databases. Neo4j has a native physical structure with index-free adjacency. Neo4j is positioned as an alternative to relational database for transaction processing.
- *Azure Cosmos DB* is a multi-model database from Microsoft, supporting all four NoSQL models. Azure Cosmos DB is available as a cloud service only, running on the Microsoft Azure cloud. Although relatively new (first released in 2014), Azure Cosmos DB is rapidly gaining popularity in the NoSQL community. Azure Cosmos DB offers a range of consistency models, including immediate, eventual, and 'bounded staleness'. With bounded staleness, reads lag behind the latest write by at most a fixed number of versions or a fixed time interval.
- *GraphDB*, from Ontotext, is the leading triple store database. GraphDB is one of the oldest graph databases, first released in 2000. GraphDB supports the SPARQL query language and an extension for geospatial queries called GeoSPARQL. GraphDB has not achieved the popularity of Neo4j or Cosmos DB, according to DB-Engines.com, in part because the triple store model is more specialized than the property graph model.

Graph databases comprise a small but steadily growing NoSQL category. Graph databases benefit from similarities to entity-relationship models and natural language but are challenged by the lack of a standard graph model and query language.

Table 23.5.2: Database systems.

	Developer	Initial release	Database models	License	DB-Engines rank (May 2020)
Neo4j	Neo4j	2007	Graph (property)	Cloud Comercial Open source	21

Azure Cosmos DB	Microsoft	2014	Key-value Wide column Document Graph (property)	Cloud	24
GraphDB	Ontotext	2000	Graph (triple store)	Cloud Commercial	136

PARTICIPATION ACTIVITY 23.5.9: Graph database systems.

If unable to drag and drop, refresh the page.

Azure Cosmos DB   Neo4j   GraphDB

	Supports the triple store database model.
	The most widely used graph database.
	A multi-model database, supporting all four NoSQL database models.

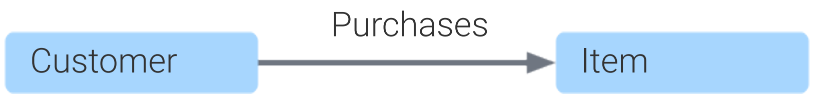
Reset

CHALLENGE ACTIVITY 23.5.1: Graph databases.

544874.3500394.qx3zqy7

Start

Consider the following property graph structure.



Select the most appropriate vertex for each label.

Customer	Select
Item	Select

1 2 3 4

Check Next

Exploring further:

- Neo4j
- Microsoft Azure Cosmos DB
- Ontotext GraphDB

23.6 MongoDB

MongoDB document database

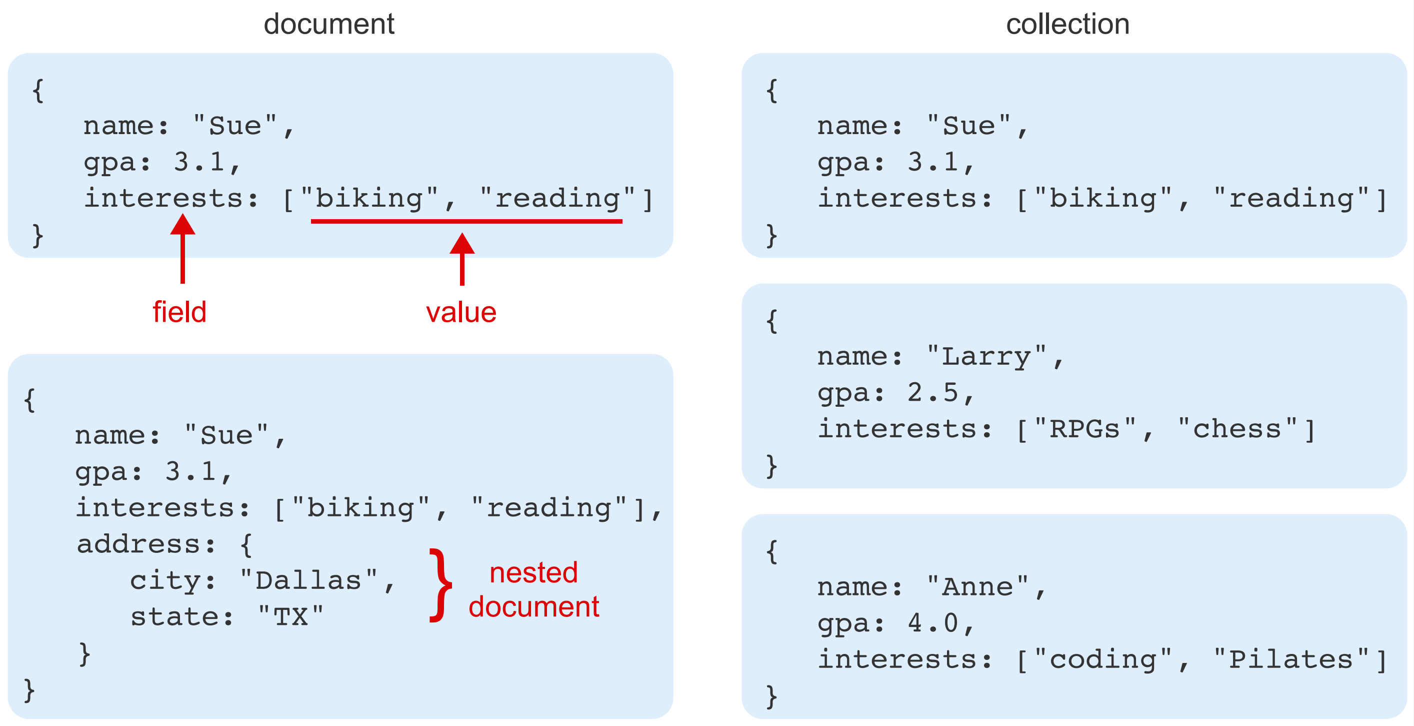
This section presents a detailed example of a NoSQL database.

MongoDB is a document database and, according to DB-Engines rankings, the leading NoSQL database of any type. As of May 2022, MongoDB was ranked fifth overall, behind only relational databases and well ahead of all other NoSQL databases.

MongoDB stores data objects as documents inside a collection. A document is a single data object in a MongoDB database that is composed of field/value pairs, similar to JSON property/value pairs. A collection is a group of related documents in a MongoDB database.

MongoDB stores documents internally as BSON documents. A ***BSON document*** (Binary JSON) is a binary representation of JSON with additional type information. BSON types include string, integer, double, date, boolean, null, and others. A BSON document may not exceed 16 MB in size.

PARTICIPATION ACTIVITY | 23.6.1: Documents and collections.



```
{
   name: "Sue",
   gpa: 3.1,
   interests: ["biking", "reading"]
}
```

```
{
   name: "Sue",
   gpa: 3.1,
   interests: ["biking", "reading"],
   address: {
      city: "Dallas",
      state: "TX"
   }
}
```

```
{
   name: "Sue",
   gpa: 3.1,
   interests: ["biking", "reading"]
}
```

```
{
   name: "Larry",
   gpa: 2.5,
   interests: ["RPGs", "chess"]
}
```

```
{
   name: "Anne",
   gpa: 4.0,
   interests: ["coding", "Pilates"]
}
```

Animation content:

Static figure:
Two diagrams appear, with captions document and collection.

The document diagram has two BSON documents. The first document is:
{ name: "Sue", gpa: 3.1, interests: ["biking", "reading"] }
The second document is:
{ name: "Sue", gpa: 3.1, interests: ["biking", "reading"], address: { city: "Dallas", state: "TX" } }
In the first document, interests has caption field, and ["biking", "reading"] has caption value. In the second document, {city: "Dallas", state: "TX"} has caption nested documents.

The collection diagram has three BSON documents:
{ name: "Sue", gpa: 3.1, interests: ["biking", "reading"] }
{ name: "Larry", gpa: 2.5, interests: ["RPGs", "chess"] }
{ name: "Anne", gpa: 4.0, interests: ["coding", "Pilates"] }

Step 1: A single student is represented as a document with field:value pairs. The name field is assigned a BSON string, gpa is a double, and interests is an array. The first document of the document diagram appears with captions field and value.

Step 2: Documents may be nested. The student document contains a nested address document. The second document of the document diagram appears with caption nested document.

Step 3: MongoDB organizes documents into collections. A group of students is stored in a single collection. The three documents of the collection diagram appear.

Animation captions:

1. A single student is represented as a document with field:value pairs. The name field is assigned a BSON string, gpa is a double, and interests is an array.
2. Documents may be nested. The student document contains a nested address document.
3. MongoDB organizes documents into collections. A group of students is stored in a single collection.

PARTICIPATION ACTIVITY | 23.6.2: MongoDB concepts.

1) All MongoDB documents must be stored in a collection.

- ○ True
- ○ False

2) MongoDB documents may store nested documents.

- ○ True
- ○ False

3) Document fields do not require quotes, but all values do.

- ○ True
- ○ False

4) MongoDB stores documents in a binary-encoded format.

- ○ True
- ○ False

5) A size limit exists for a BSON document.

- ○ True
- ○ False

Installing MongoDB

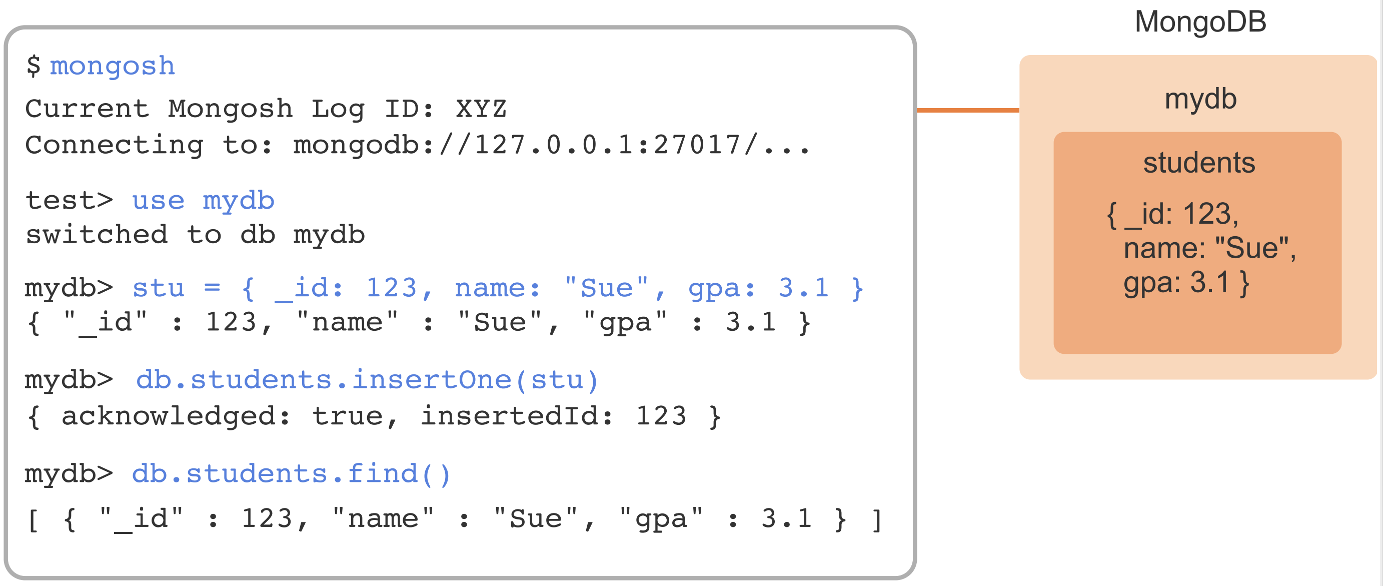
MongoDB runs on a wide range of operating systems. Instructions for installing MongoDB Community Edition are provided on the MongoDB website.

MongoDB Shell is a program for interacting with MongoDB. Instructions for installing MongoDB Shell are also available on the MongoDB website.

MongoDB Shell

The ***MongoDB Shell*** is a command-line interface for creating and deleting documents, querying, creating user accounts, and performing many other operations in MongoDB. The `mongosh` command starts the MongoDB shell.

PARTICIPATION ACTIVITY 23.6.3: Running MongoDB Shell.



```
$ mongosh
Current Mongosh Log ID: XYZ
Connecting to: mongodb://127.0.0.1:27017/...

test> use mydb
switched to db mydb

mydb> stu = { _id: 123, name: "Sue", gpa: 3.1 }
{ "_id" : 123, "name" : "Sue", "gpa" : 3.1 }

mydb> db.students.insertOne(stu)
{ acknowledged: true, insertedId: 123 }

mydb> db.students.find()
[ { "_id" : 123, "name" : "Sue", "gpa" : 3.1 } ]
```

Animation content:

Static figure:
The following console is shown:
$ mongosh
Current Mongosh Log ID: XYZ
Connecting to: mongodb://127.0.0.1:27017/...
test> use mydb
switched to db mydb

mydb> stu = { _id: 123, name: "Sue", gpa: 3.1 }
{ "_id" : 123, "name" : "Sue", "gpa" : 3.1 }

mydb>db.students.insertOne(stu)
{ acknowledged: true, insertedId: 123 }

mydb>db.students.find()
[{ "_id" : 123, "name" : "Sue", "gpa" : 3.1 }]

The MongoDB shell is displayed next which holds the mydb database. mydb database contains the student collection with the following student information:
{ _id: 123,
 name: "Sue",
 gpa: 3.1 }

Animation captions:

1. Entering "mongosh" at the command line starts the MongoDB Shell and connects to the MongoDB instance running on the local computer.
2. The use command creates a new database called "mydb" since mydb does not exist.
3. stu is assigned a document and inserted into the students collection with insertOne().
4. The find() method retrieves the one student in the students collection.

PARTICIPATION ACTIVITY | 23.6.4: MongoDB Shell commands.

Refer to the commands entered into the MongoDB Shell below.

```
test> use dealer
switched to db dealer

dealer> car = { _id: 200, make: "Ford", model: "Mustang" }
```

```
{ "_id" : 200, "make" : "Ford", "model" : "Mustang" }

dealer> db.autos.insertOne(car)
{ acknowledged: true, insertedId: 200 }

dealer> db.autos.find()
[ { "_id" : 200, "make" : "Ford", "model" : "Mustang" } ]

dealer> show dbs
admin     41 kB
config   111 kB
dealer  73.7 kB
local     41 kB
mydb    73.7 kB

dealer> help
  Shell Help:

    use                 Set current database
    show                'show databases'/'show dbs': Print a list of all available
databases.
                        'show collections'/'show tables': Print a list of all
collections for...
    ...

dealer> show collections
autos

dealer> db.autos.help()
  Collection Class:

    aggregate           Calculates aggregate values for the data in a collection or a
view.
    bulkWrite           Performs multiple write operations with controls for order of
execution.
    count               Returns the count of documents that would match a find() query
for...
    ...

dealer> exit
```

If unable to drag and drop, refresh the page.

db.autos.insertOne(car) help show dbs show collections use dealer

exit db.autos.help()

	Displays a summary of all available shell commands.
	Displays a summary of all collection methods.
	Creates or selects a database called "dealer"

	called "dealer".
	Quits the MongoDB Shell.
	Displays a list of all databases.
	Displays all collections in the current database.
	Inserts a single document into the "autos" collection.

Reset

Inserting documents

The ***insertOne()*** collection method inserts a single document into a collection. The ***insertMany()*** collection method inserts multiple documents into a collection.

In the figure below, Sue is inserted into the students collection, then three students in the `students` array are inserted.

Figure 23.6.1: Inserting multiple students in bulk.

```
mydb> db.students.insertOne({ name: "Sue", gpa: 3.1 })
{
  acknowledged: true,
  insertedId: ObjectId("62794229fc4ebd4933877a9d")
}

mydb> students = [
... { name: "Maria", gpa: 4.0 },
... { name: "Xiu", gpa: 3.8 },
... { name: "Braden", gpa: 2.5 } ]

mydb> db.students.insertMany(students)
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("627942f6fc4ebd4933877a9e"),
    '1': ObjectId("627942f6fc4ebd4933877a9f"),
    '2': ObjectId("627942f6fc4ebd4933877aa0")
  }
}

mydb> db.students.find()
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa: 3.1
},
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 4
},
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 3.8
},
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa:
2.5 }
]
```

The ***_id*** field is automatically created for every document and is always the first field in the document. The _id acts as a primary key. A ***primary key*** is a field that uniquely identifies each document in a collection.

The _id may be assigned a unique value like a student ID number or use an auto-incrementing value. In the figure above, no _id field was assigned, so MongoDB automatically assigned an ObjectId to _id. An ***ObjectId*** is a 12-byte BSON type that contains a unique value. An ObjectId is displayed as hexadecimal numbers. Ex: 56e0a0a501c0b1ea806cadcb.

PARTICIPATION ACTIVITY 23.6.5: MongoDB _id field.

1) The command below assigns _id an ObjectId.

```
db.students.insertOne({ _id:
123, "Ebony", gpa: 3.2 })
```

- True
- False

2) In the figure above, the inserted documents received similar ObjectIds.

- True
- False

3) The ObjectId is 12 characters long.

- True
- False

4) The _id field may use duplicate values.

- True
- False

Finding documents

The ***find()*** collection method returns all documents by default or documents that match an optional query parameter. The ***findOne()*** collection method returns only the first document matching the query. Both methods return null if the query matches no documents.

Figure 23.6.2: Find 'Sue' and students with GPA ≥ 3.0.

```
mydb> db.students.find({ name: 'Sue' })
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa:
3.1 }
]

mydb> db.students.find({ gpa: { $gte: 3.0 } })
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa:
3.1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa:
4 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa:
3.8 }
]
```

The operators used in queries are summarized in the table below, but many more exist and are documented in the MongoDB manual.

Table 23.6.1: Common MongoDB query operators.

Operator	Description	Example

`field:value`	Matches documents with fields that are equal to the given value.	`// Matches Sue`
		`{ "_id" : ObjectId("62794229fc4ebd4933877a9d") }`

`$eq   $ne`	Matches values = or ≠ to the given value.	`// Matches all docs except Sue`
		`{ name: { $ne: "Sue" } }`

`$gt   $gte`	Matches values > or ≥ to the given value.	`// Matches students with gpa > 3.5`
		`{ gpa: { $gt: 3.5 } }`

`$lt   $lte`	Matches values < or ≤ to the given value.	`// Matches students with gpa <= 3.0`
		`{ gpa: { $lte: 3.0 } }`

	Matches values in or	`// Matches  Sue, Susan, or Susie`

`$in` `$nin`	Matches values in or not in a given array.	`{ name: { $in: ["Sue", "Susan", "Susie"] } }`

`$and`	Joins query clauses with a logical AND, returns documents that match both clauses.	`// Matches student with gpa >= 3.0 and gpa <= 3.5`
		`{ $and: [{ gpa: { $gte: 3.0 } },`
		`{ gpa: { $lte: 3.5 } }] }`

`$or`	Joins query clauses with a logical OR, returns documents that match either clauses.	`// Matches students with gpa >= 3.9 or gpa <= 3.0`
		`{ $or: [{ gpa: { $gte: 3.9 } },`
		`{ gpa: { $lte: 3.0 } }] }`

PARTICIPATION ACTIVITY | 23.6.6: Querying the 'autos' collection.

Refer to the "autos" collection below, and choose the documents returned by each query.

```
[
   {
      "_id" : 100,
      "make" : "Ford",
      "model" : "Fusion",
      "year" : 2014,
      "options" : [ "engine start", "moon roof" ],
      "price" : 13500
   },
   {
      "_id" : 200,
      "make" : "Honda",
      "model" : "Accord",
      "year" : 2013,
      "options" : [ "spoiler", "alloy wheels", "sunroof" ],
      "price" : 16900
   },
   {
      "_id" : 300,
      "make" : "Dodge",
      "model" : "Avenger",
      "year" : 2012,
      "options" : [ "leather seats" ],
      "price" : 10800
   },
   {
      "_id" : 400,
      "make" : "Toyota",
      "model" : "Corolla",
      "year" : 2013,
      "options" : [ "antitheft" ],
      "price" : 13400
    }
]
```

1) `db.autos.find({})`

- ○ _id 100
- ○ All documents
- ○ null

2) `db.autos.find({ year: { $gte: 2013 } })`

- ○ _id 100
- ○ _id 100, 200, 400
- ○ _id 300

3) `db.autos.findOne({ year: { $gte: 2013 } })`

- ○ _id 100
- ○ _id 100, 200, 400
- ○ _id 300

4) `db.autos.findOne({ year: { $gte: 2016 } })`

- ○ _id 100
- ○ Run-time error
- ○ null

5)
```
db.autos.find({ $and: [
{price: { $lte: 15000 } },
   { options: { $in:
["sunroof", "antitheft", "moon
roof"]
   } } ] })
```

- ○ _id 100
- ○ _id 100, 400
- ○ _id 100, 200, 400

6)
```
db.autos.find({ $or: [
{"make":"Honda"},
 { year: {$ne: 2013} } ] })
```

○ All documents

○ _id 100, 300

○ _id 100, 200, 300

Updating documents

The ***updateOne()*** collection method modifies a single document in a collection. The ***updateMany()*** collection method modifies multiple documents in a collection. The methods have two required parameters:

1. `query` - The query to find the document(s) to update. An empty query `{}` matches all documents.
2. `update` - The modification to perform on matched documents using an update operator like $inc, $set, and $unset.

In the example below, the calls to updateOne() and updateMany() return the matchedCount property indicating how many documents matched the query, and the modifiedCount property indicating the number of documents modified.

Figure 23.6.3: Change Sue's GPA to 3.3 and set students with GPA > 3 to 1.

```
mydb> db.students.updateOne({ name: 'Sue' }, { $set: { gpa: 3.3 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

mydb> db.students.find({ name: 'Sue' })
{ "_id" : ObjectId("5e600d18bbd10ee972f6ed9a"), "name" : "Sue", "gpa" :
3.3 }

mydb> db.students.updateMany({ gpa: { $gt: 3 } }, { $set: { gpa: 1 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}

mydb> db.students.find()
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: "Braden", gpa: 2.5 }
]
```

Table 23.6.2: Common MongoDB update operators.

Operator	Description	Example
`$currentDate`	Sets a field's value to the current date/time	`// Sue's "birthDate" : ISODate("2022-06-12T16:39:00.121Z")` `db.students.updateOne({ name: 'Sue' }, { $currentDate: { birthDate: true } })`
`$inc`	Increments a field's value by the specified amount	`// Sue's "gpa" incremented from 3.1 to 3.2` `db.students.updateOne({ name: 'Sue' }, { $inc: { gpa: 0.1 } })`
`$rename`	Renames a field	`// Sue's "name" is now "firstName"` `db.students.updateOne({ name: 'Sue' }, { $rename: { name: 'firstName' } })`
`$set`	Sets a field's value	`// Sue's "gpa" : 4.0` `db.students.updateOne({ name: 'Sue' }, { $set: { gpa: 4.0 } })`
`$unset`	Removes a field	`// Removes Sue's "gpa" and "birthDate" fields` `db.students.updateOne({ name: 'Sue' }, { $unset: { gpa: "", birthDate: "" } })`

PARTICIPATION ACTIVITY 23.6.7: Updating documents in the 'autos' collection.

Refer to the "autos" collection below, and choose the result of each command.

```
[
   {
      "_id" : 100,
      "make" : "Ford",
      "model" : "Fusion",
      "year" : 2014,
      "options" : [ "engine start", "moon roof" ],
      "price" : 13500
   },
   {
      "_id" : 200,
      "make" : "Honda",
      "model" : "Accord",
      "year" : 2013,
      "options" : [ "spoiler", "alloy wheels", "sunroof" ],
      "price" : 16900
   }
]
```

1)
```
db.autos.updateOne({ price: {
$gt: 10000} },
   { $set: { year: 2000,
options: [] }})
```

- ○ Only auto with _id 100 has year set to 2000 and options removed.
- ○ Both autos have year set to 2000 and options removed.
- ○ No autos are updated.

2)
```
db.autos.updateMany({ price: {
$gt: 10000} },
   { $set: { year: 2000,
options: [] }})
```

- ○ Only auto with _id 100 has year set to 2000 and options removed.
- ○ Both autos have year set to 2000 and options removed.
- ○ No autos are updated.

3)
```
db.autos.updateOne({ price: {
$gt: 10000} },
   { $set: { sold: true }})
```

- ○ No autos are updated because the autos do not have a "sold" field

- ○ Auto with _id 100 has new field "sold" set to true.
- ○ Both autos have a new field "sold" set to true.

4)
```
db.autos.updateOne({ _id: 100
},
   { $currentDate: { soldDate:
true }})
```

- ○ Auto with _id 100 has new field "soldDate" set to true.
- ○ Auto with _id 100 has new field "soldDate" set to the Unix epoch (January 1, 1970).
- ○ Auto with _id 100 has new field "soldDate" set to the current date and time.

5)
```
db.autos.updateOne({ _id: 100
},
   { $inc: { price: -500,
year: 2 }})
```

- ○ Auto with _id 100 has price reduced by 500 and year increased by 2.
- ○ Auto with _id 100 has price set to -500 and year set to 2.
- ○ Auto with _id 100 has price and year fields removed.

Deleting documents

The ***deleteOne()*** collection method deletes a single document from a collection. The ***deleteMany()*** collection method deletes multiple document from a collection. The methods have a required query parameter that matches documents to delete.

In the figure below, the calls to deleteOne() and deleteMany() return a deletedCount property indicating how many documents were deleted.

Figure 23.6.4: Delete the first student with GPA < 3.5 (Sue), and delete all

students with GPA > 3.5 (Maria and Xiu).

```
mydb> db.students.deleteOne({ gpa: { $lt:3.5 } })
{ acknowledged: true, deletedCount: 1 }

mydb> db.students.find()
[
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 4
},
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 3.8
},
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa:
2.5 }
]

mydb> db.students.deleteMany({ gpa: { $gt:3.5 } })
{ acknowledged: true, deletedCount: 2 }

mydb> db.students.find()
[
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa:
2.5 }
]
```

PARTICIPATION ACTIVITY 23.6.8: Removing documents.

1) Remove all documents from the "autos" collection.

`db.autos.`

Check Show answer

2) Remove all documents with a price more than $10,000.

`db.autos.`

Check Show answer

3) Remove only the first document with the year 2020.

`db.autos.`

Check Show answer

Exploring further:

- MongoDB manual
- Installing MongoDB
- Query and Projection Operators
- SQL to MongoDB Mapping Chart