**UNIT II NOSQL DATA MANAGEMENT**

**2.1.1 Introduction to** **NoSQL**

The NoSQL Database is a non-relational data management system that does not require a schema. It doesn't require any joins and is simple to scale. A NoSQL database is mostly used for distributed data stores with massive data storage requirements. NoSQL is used in both big data and real-time web apps. Every day, firms like Twitter, Facebook, and Google, for example, amass gigabytes of user data.

A database that is not SQL is referred to as "Not Only SQL" or "Not SQL". NoSQL has become popular, despite the fact that a better term would be "NoREL." In 1998, Carl Strozz coined the term "NoSQL.” Traditional RDBMS employ SQL syntax to store and retrieve data for further processing. A NoSQL database system, on the other hand, refers to a collection of database systems that may hold structured, semi-structured, unstructured, and polymorphic data

NoSQL, also referred to as “not only SQL”, “non-SQL”, is an approach to database design that enables the storage and querying of data outside the traditional structures found in relational databases. While it can still store data found within relational database management systems (RDBMS), it just stores it differently compared to an RDBMS. The decision to use a relational database versus a non-relational database is largely contextual, and it varies depending on the use case.

Instead of the typical tabular structure of a relational database, NoSQL databases, house data within one data structure, such as JSON document. Since this non-relational database design does not require a schema, it offers rapid scalability to manage large and typically unstructured data sets.

NoSQL is also type of distributed database, which means that information is copied and stored on various servers, which can be remote or local. This ensures availability and reliability of data. If some of the data goes offline, the rest of the database can continue to run.

Today, companies need to manage large data volumes at high speeds with the ability to scale up quickly to run modern web applications in nearly every industry. In this era of growth within cloud, big data, and mobile and web applications, NoSQL databases provide that speed and scalability, making it a popular choice for their performance and ease of use.

**NoSQL vs. SQL**

Structured query language (SQL) is commonly referenced in relation to NoSQL. To better understand the difference between NoSQL and SQL, it may help to understand the history of SQL, a programming language used for retrieving specific information from a database.

Before relational databases, companies used a hierarchical database system with a tree-like structure for the data tables. These early database management systems (DBMS) enabled users to organize large quantities of data. However, they were complex, often proprietary to a particular application, and limited in the ways in which they could uncover within the data. These limitations eventually led to the development of relational database management systems, which arranged data in tables. SQL provided an interface to interact with relational data, allowing analysts to connect tables by merging on common fields.

As time passed, the demands for faster and more disparate use of large data sets became increasingly more important for emerging technology, such as e-commerce applications. Programmers needed something more flexible than SQL databases (i.e. relational databases). NoSQL became that alternative.

While NoSQL provided an alternative to SQL, this advancement by no means replaced SQL databases. For example, let's say that you are managing retail orders at a company. In a relational model, individual tables would manage customer data, order data and product data separately, and they would be joined together through a unique, common key, such as a Customer ID or an Order ID. While this is great for storing and retrieving data quickly, it requires significant memory. When you want to add more memory, SQL databases can only scale vertically, not horizontally, which means your ability to add more memory is limited to the hardware you have. The result is that vertical scaling ultimately limits your company’s data storage and retrieval.

In comparison, NoSQL databases are non-relational, which eliminates the need for connecting tables. Their built-in sharding and high availability capabilities ease horizontal scaling. If a single database server is not enough to store all your data or handle all the queries, the workload can be divided across two or more servers, allowing companies to scale their data horizontally.

**Key Features of NoSQL Database**

Some of the main features of the NoSQL Database are listed below:

* **Horizontal Scaling**: NoSQL Databases can scale horizontally by adding nodes to share loads. As the data grows the hardware can be added and scalability features could be preserved for NoSQL.
* **Performance**: Users can increase the performance of the NoSQL Database by adding a different server.
* **Flexible Schema**: NoSQL Databases do not require the same schema as compared to SQL Databases. The document in the same collection does not need to have the same set of fields and data type.
* **High Availability**: Unlike Relational Databases that use primary and secondary nodes for fetching data. NoSQL Databases use master place architecture.

**2.1.2 Aggregate Data Models**

**What are Aggregate Data Models in NoSQL?**

* Aggregate means a collection of objects that are treated as a unit. In NoSQL Databases, an aggregate is a collection of data that interact as a unit. Moreover, these units of data or aggregates of data form the boundaries for the ACID operations.
* Aggregate Data Models in NoSQL make it easier for the Databases to manage data storage over the clusters as the aggregate data or unit can now reside on any of the machines. Whenever data is retrieved from the Database all the data comes along with the Aggregate Data Models in NoSQL.
* Aggregate Data Models in NoSQL don’t support ACID transactions and sacrifice one of the ACID properties. With the help of Aggregate Data Models in NoSQL, you can easily perform OLAP operations on the Database.
* You can achieve high efficiency of the Aggregate Data Models in the NoSQL Database if the data transactions and interactions take place within the same aggregate.

**Types of Aggregate Data Models in NoSQL Databases**

The Aggregate Data Models in NoSQL are majorly classified into 4 Data Models listed below:



Fig.2.1.2.1 Different Aggregate Data Models

**(i) Key Value Model**

The Key-Value Data Model contains the key or an ID used to access or fetch the data of the aggregates corresponding to the key. In this Aggregate Data Models in NoSQL, the data of the aggregates are secure and encrypted and can be decrypted with a Key.

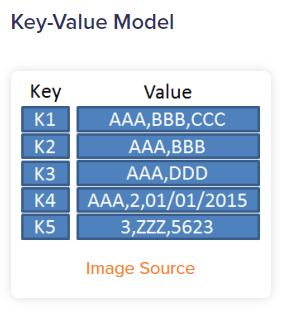


Fig.2.1.2.2 Key Value Model

**Use Cases:**

* These Aggregate Data Models in NoSQL Database are used for storing the user session data.
* Key Value-based Data Models are used for maintaining schema-less user profiles.
* It is used for storing user preferences and shopping cart data.

**(ii) Document Model**

The Document Data Model allows access to the parts of aggregates. In this Aggregate Data Models in NoSQL, the data can be accessed in an inflexible way. The Database stores and retrieves documents, which can be XML, JSON, BSON, etc. There are some restrictions on data structure and data types of the data aggregates that are to be used in this Aggregate Data Models in NoSQL Database.

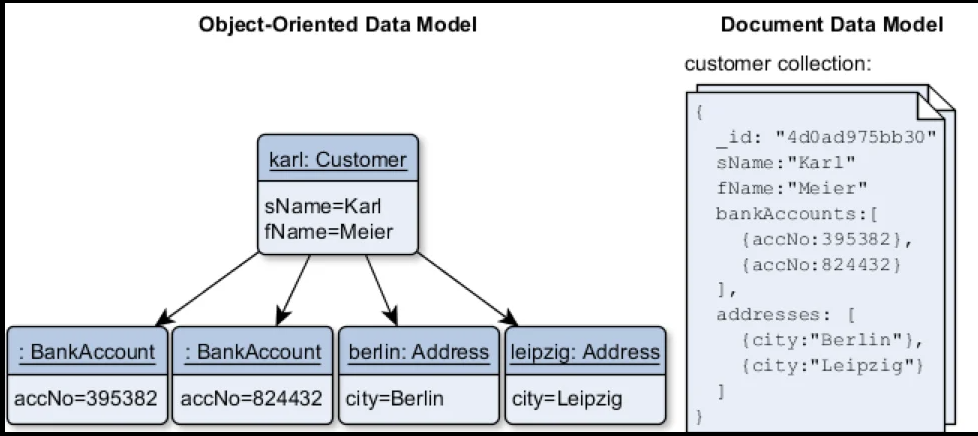


Fig.2.1.2.3 Document Model

**Use Cases:**

* Document Data Models are widely used in E-Commerce platforms
* It is used for storing data from content management systems.
* Document Data Models are well suited for Blogging and Analytics platforms.

**(iii) Column Family Model**

Column family is an Aggregate Data Models in NoSQL Database usually with big-table style Data Models that are referred to as column stores. It is also called a two-level map as it offers a two-level aggregate structure. In this Aggregate Data Models in NoSQL, the first level of the Column family contains the keys that act as a row identifier that is used to select the aggregate data. Whereas the second level values are referred to as columns.

**Use Cases:**

* Column Family Data Models are used in systems that maintain counters.
* These Aggregate Data Models in NoSQL are used for services that have expiring usage.
* It is used in systems that have heavy write requests.

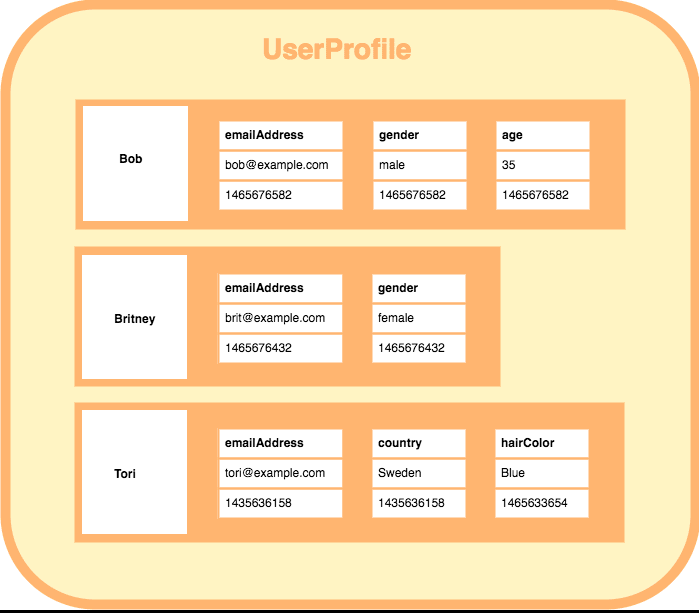


Fig.2.1.2.4 Column Family Model

**(iv)** **Graph-Based Model**

Graph-based data models store data in nodes that are connected by edges. These Aggregate Data Models in NoSQL are widely used for storing the huge volumes of complex aggregates and multidimensional data having many interconnections between them.

**Use Cases:**

* Graph-based Data Models are used in social networking sites to store interconnections.
* It is used in fraud detection systems.
* This Data Model is also widely used in Networks and IT operations.

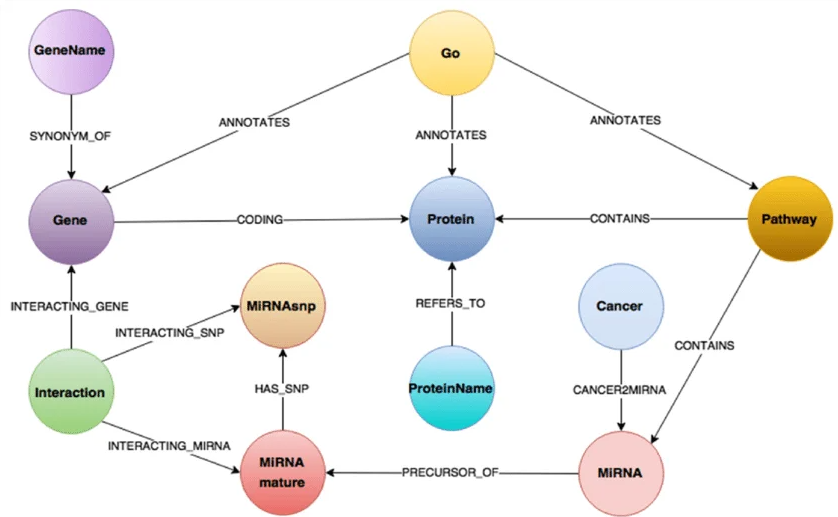


Fig 2.1.2.5 Graph Based Model

**STEPS TO BUILD AGGREGATE DATA MODELS IN NOSQL DATABASES**

For this, a Data Model of an E-Commerce website will be used to explain Aggregate Data Models in NoSQL.

This example of the E-Commerce Data Model has two main aggregates – customer and order. The customer contains data related to billing addresses while the order aggregate consists of ordered items, shipping addresses, and payments. The payment also contains the billing address.

If you notice a single logical address record appears 3 times in the data, but its value is copied each time wherever used. The whole address can be copied into an aggregate as needed. There is no pre-defined format to draw the aggregate boundaries. It solely depends on whether you want to manipulate the data as per your requirements.

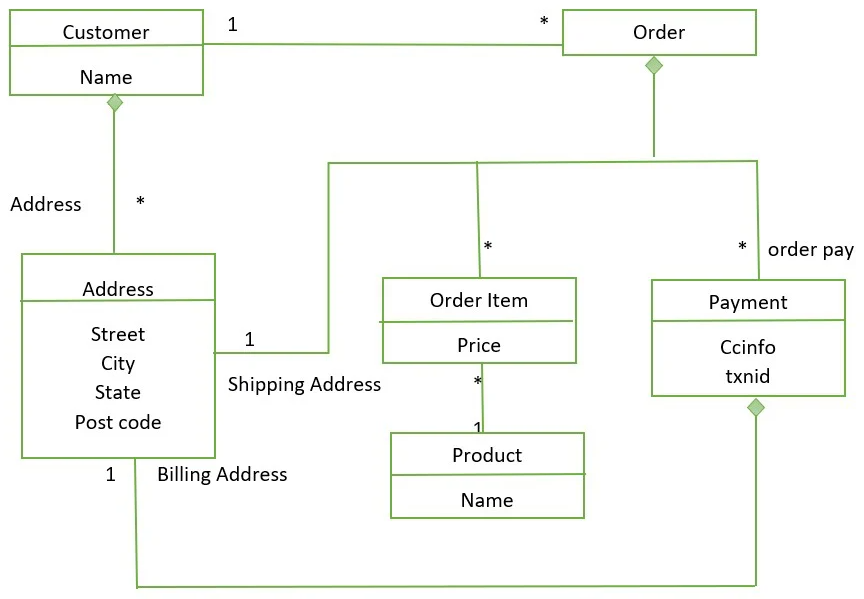


Fig 2.1.2.6 UML Diagram for E-Commerce Site - Aggregate Data Models in NoSQL

The Data Model for customer and order would look like this.

// in customers

{

"customer": {

"id": 1,

"name": "Martin",

"billingAddress": [{"city": "Chicago"}],

"orders": [

{

"id":99,

"customerId":1,

"orderItems":[

{

"productId":27,

"price": 32.45,

"productName": "NoSQL Distilled"

}

],

"shippingAddress":[{"city":"Chicago"}]

"orderPayment":[

{

"ccinfo":"1000-1000-1000-1000",

"txnId":"abelif879rft",

"billingAddress": {"city": "Chicago"}

}],

}]

}

}

In these Aggregate Data Models in NoSQL, if you want to access a customer along with all customer’s orders at once. Then designing a single aggregate is preferable. But if you want to access a single order at a time, then you should have separate aggregates for each order. It is very content-specific.

**2.2 KEY VALUE AND DOCUMENT DATA MODELS**

**2.2.1 KEY VALUE MODEL**

The Key-Value Data Model contains the key or an ID used to access or fetch the data of the aggregates corresponding to the key. In this Aggregate Data Models in NoSQL, the data of the aggregates are secure and encrypted and can be decrypted with a Key.

A key-value database is a type of nonrelational database that uses a simple key-value method to store data. A key-value database stores data as a collection of key-value pairs in which a key serves as a unique identifier. Both keys and values can be anything, ranging from simple objects to complex compound objects. Key-value databases are highly partitionable and allow horizontal scaling at scales that other types of databases cannot achieve. For example, Amazon DynamoDB allocates additional partitions to a table if an existing partition fills to capacity and more storage space is required.

The following diagram shows an example of data stored as key-value pairs in DynamoDB.

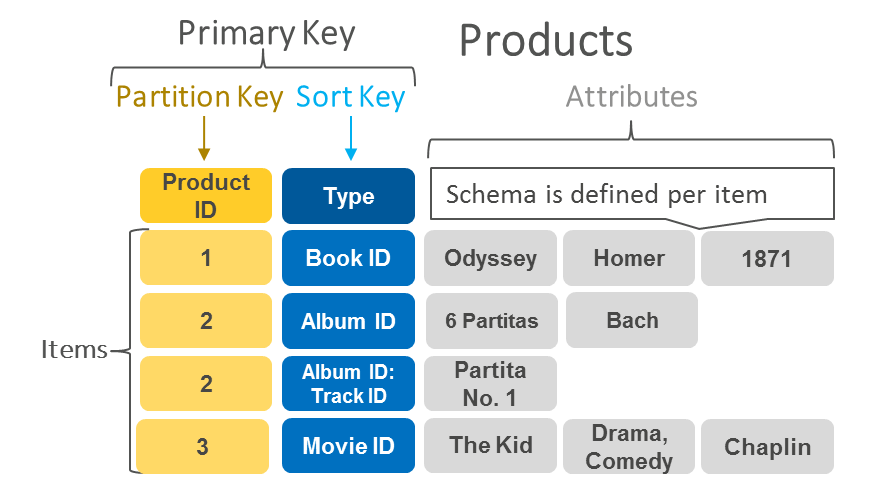


Fig.2.2.1.1 Key-value pairs in Dynamo DB.

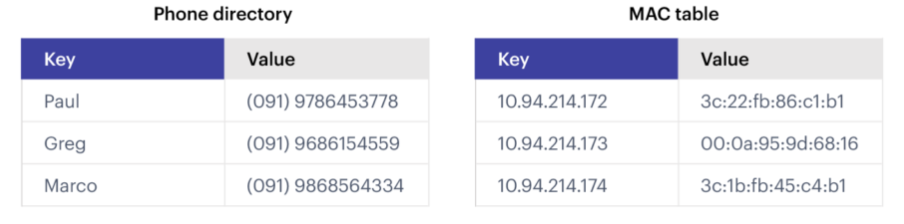


Fig.2.2.1.2 An Example of Key-value database

**Understanding Key-Value Databases**

As the name suggests, this type of NoSQL database implements a hash table to store unique keys along with the pointers to the corresponding data values. The values can be of scalar data types such as integers or complex structures such as JSON, lists, BLOB, and so on. A value can be stored as an integer, a string, JSON document, or an array—with a key used to reference that value. It typically offers excellent performance and can be optimized to fit an organization’s needs. Key-value stores have no query language but they do provide a way to add and remove key-value pairs, some vendors being quite sophisticated. Values cannot be queried or searched upon. Only the key can be queried.

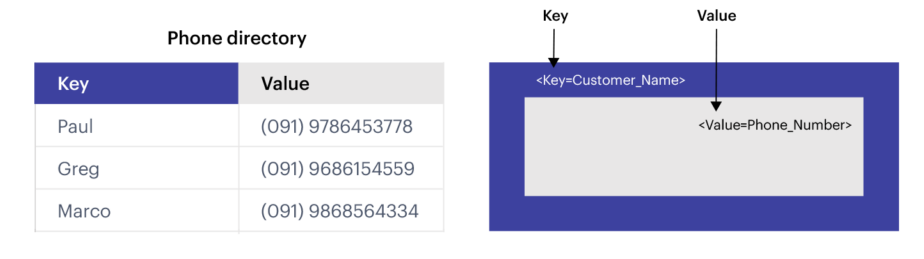


Fig.2.2.1.3 A simple example of key-value data store.

**Features of a key-value database**

A key-value database is defined by the fact that it allows programs or users of programs to retrieve data by keys, which are essentially names, or identifiers, that point to some stored value. Because key-value databases are defined so simply, but can be extended and optimized in numerous ways, there is no global list of features, but there are a few common ones:

* Retrieving a value (if there is one) stored and associated with a given key
* Deleting the value (if there is one) stored and associated with a given key
* Setting, updating, and replacing the value (if there is one) associated with a given key

Modern applications will probably require more than the above, but this is the bare minimum for a key-value store.

**WHEN TO USE A KEY-VALUE DATABASE**

**1. Handling Large Volume of Small and Continuous Reads and Writes**

Key-value databases are particularly suitable when your application requires handling a large volume of small and continuous reads and writes. These databases are designed for efficient and fast access to data stored as key-value pairs. Whether the data is volatile or frequently changing, key-value databases can provide high-performance in-memory access, making them ideal for use cases that demand quick retrieval and modification of data.

**2. Storing Basic Information**

Key-value databases are well-suited for storing basic information, such as customer details, user profiles, or simple configurations. In these scenarios, each piece of information can be associated with a unique key, allowing for easy retrieval and updates based on the key value. For example, a key-value database can be used to store webpages with the URL as the key and the webpage content as the value. Similarly, storing shopping-cart contents, product categories, or e-commerce product details can be efficiently managed using key-value databases.

**3. Applications with Infrequent Updates and Simple Queries**

Key-value databases are a good choice for applications that don’t require frequent updates or complex queries. If your application primarily focuses on data retrievaland simpleCRUD operations, key-value databases provide an efficient and straightforward solution. These databases prioritize simplicity and high-performance data access, making them suitable for applications that require quick lookups and modifications without the need for complex query capabilities or extensive data manipulation.

**4. Key-Value Databases for Volatile Data**

When your application needs to handle lots of small continuous reads and writes, that may be volatile, key-value databases offer fast in-memory access.

**USE CASES FOR KEY-VALUE DATABASES**

**1. Session Management on a Large Scale**

Key-value databases are well-suited for managing session data in applications that require handling a large number of concurrent users. These databases can efficiently store and retrieve session information, such as user authentication tokens, user preferences, or temporary data. With their fast in-memory access and ability to handle high volumes of small reads and writes, key-value databases provide an optimal solution for session management in applications with a large user base.

**2. Using Cache to Accelerate Application Responses**

Key-value databases are often employed as cache layers to accelerate application responses. By caching frequently accessed data in a key-value store, applications can reduce the need for expensive and time-consuming operations, such as database queries or complex computations. This caching strategy allows for faster data retrieval, leading to improved application performance and responsiveness.

**3. Storing Personal Data on Specific Users**

Key-value databases can efficiently store personal data on specific users. For example, they can be used to store user profile information, user preferences, or other user-specific data. With their simple key-value storage model, these databases allow for quick and efficient access to user data, making them suitable for applications that need to handle a large volume of user-specific data.

**4. Product Recommendations and Personalized Lists**

Key-value databases can be used to generate and store product recommendations and personalized lists. They provide a quick and efficient way to store and retrieve user preferences and other user-specific data, which can be used to personalize product recommendations and lists. This can lead to a more engaging and personalized user experience, improving user satisfaction and potentially driving increased revenue for businesses.

**5. Managing Player Sessions in Massive Multiplayer Online Games**

Key-value databases are excellent for managing player sessions in massive multiplayer online games (MMOGs). These games require real-time management of a large number of simultaneous player sessions, and key-value databases can provide the necessary performance and scalability to handle this challenge.

**2.2.2 DOCUMENT DATA MODELS**

A document is a record in a document database. A document typically stores information about one object and any of its related metadata.

Documents store data in field-value pairs. The values can be a variety of types and structures, including strings, numbers, dates, arrays, or objects. Documents can be stored in formats like JSON, [BSON](https://www.mongodb.com/json-and-bson), and XML.

Below is a JSON document that stores information about a user named Tom.

{

"\_id": 1,

"first\_name": "Tom",

"email": "tom@example.com",

"cell": "765-555-5555",

"likes": [

"fashion",

"spas",

"shopping"

],

"businesses": [

{

"name": "Entertainment 1080",

"partner": "Jean",

"status": "Bankrupt",

"date\_founded": {

"$date": "2012-05-19T04:00:00Z"

}

},

**Collections**

A collection is a group of documents. Collections typically store documents that have similar contents.

Not all documents in a collection are required to have the same fields, because document databases have a flexible schema. Note that some document databases provide schema validation, so the schema can optionally be locked down when needed.

Continuing with the example above, the document with information about Tom could be stored in a collection named users. More documents could be added to the users collection in order to store information about other users. For example, the document below that stores information about Donna could be added to the users collection.

**CRUD operations**

Document databases typically have an API or query language that allows developers to execute the CRUD (create, read, update, and delete) operations.

* **Create**: Documents can be created in the database. Each document has a unique identifier.
* **Read**: Documents can be read from the database. The API or query language allows developers to query for documents using their unique identifiers or field values. Indexes can be added to the database in order to increase read performance.
* **Update**: Existing documents can be updated — either in whole or in part.
* **Delete**: Documents can be deleted from the database.

**Key features of document databases**

Document databases have the following key features:

* **Document model:** Data is stored in documents (unlike other databases that store data in structures like tables or graphs). Documents map to objects in most popular programming languages, which allows developers to rapidly develop their applications.
* **Flexible schema:** Document databases have a flexible schema, meaning that not all documents in a collection need to have the same fields. Note that some document databases support schema validation, so the schema can be optionally locked down.
* **Distributed and resilient:** Document databases are distributed, which allows for horizontal scaling (typically cheaper than vertical scaling) and data distribution. Document databases provide resiliency through replication.
* **Querying through an API or query language:** Document databases have an API or query language that allows developers to execute the CRUD operations on the database. Developers have the ability to query for documents based on unique identifiers or field values.

**Document databases different from relational databases**

Three key factors differentiate document databases from relational databases:  
  
1. **The intuitiveness of the data model:** Documents map to the objects in code, so they are much more natural to work with. There is no need to decompose data across tables, run expensive joins, or integrate a separate Object Relational Mapping (ORM) layer. Data that is accessed together is stored together, so developers have less code to write and end users get higher performance.

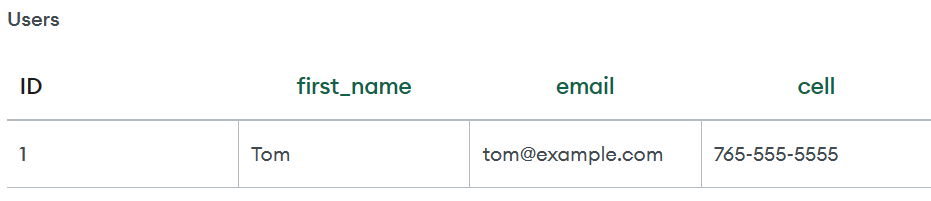
2. **The ubiquity of JSON documents:** JSON has become an established standard for data interchange and storage. JSON documents are lightweight, language-independent, and human-readable. Documents are a superset of all other data models so developers can structure data in the way their applications need — rich objects, key-value pairs, tables, geospatial and time-series data, or the nodes and edges of a graph.

3. **The flexibility of the schema:** A document’s schema is dynamic and self-describing, so developers don’t need to first pre-define it in the database. Fields can vary from document to document. Developers can modify the structure at any time, avoiding disruptive schema migrations. Some document databases offer schema validation so you can optionally enforce rules governing document structures.

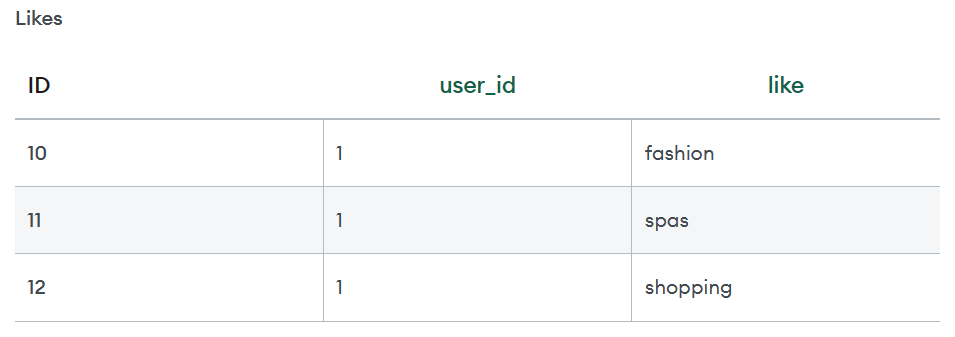
**Documents to work with than tables**

Developers commonly find working with data in documents to be easier and more intuitive than working with data in tables. Documents map to data structures in most popular programming languages. Developers don't have to worry about manually splitting related data across multiple tables when storing it or joining it back together when retrieving it. They also don't need to use an ORM to handle manipulating the data for them. Instead, they can easily work with the data directly in their applications.

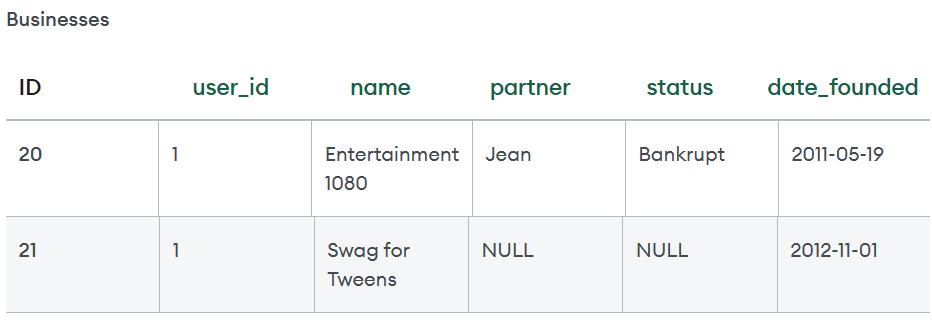
Now let's consider how we can store that same information in a relational database. We'll begin by creating a table that stores the basic information about the user.



A user can like many things (meaning there is a one-to-many relationship between a user and likes), so we will create a new table named "Likes" to store a user’s likes. The Likes table will have a foreign key that references the ID column in the Users table.



Similarly, a user can run many businesses, so we will create a new table named "Businesses" to store business information. The Businesses table will have a foreign key that references the ID column in the Users table.

****

**Strengths and weaknesses of document databases Document databases have many strengths:**

* The document model is ubiquitous, intuitive, and enables rapid software development.
* The flexible schema allows for the data model to change as an application's requirements change.
* Document databases have rich APIs and query languages that allow developers to easily interact with their data.
* Document databases are distributed (allowing for horizontal scaling as well as global data distribution) and resilient.

These strengths make document databases an excellent choice for a general-purpose database.

A common weakness that people cite about document databases is that many do not support multi-document ACID transactions. We estimate that 80%-90% of applications that leverage the document model will not need to use multi-document transactions.

Note that some document databases like MongoDB support multi-document ACID transactions.

relationships – graph databases – schema less databases- materialized views – distribution models - master-slave replication – consistency - Cassandra – Cassandra data model – Cassandra examples – Cassandra client- Lumify

While a relational database stores data in rows and reads data row by row, a column store is organized as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster. Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). Use cases include analytics.

The keys and the column names of this type of database are not fixed. Columns within the same column family, or cluster of columns, can have a different number of rows and can accommodate different types of data and names. These databases are most often utilized when there is a need for a large data model. They are very useful for data warehouses, or when there is a need for high performance or handling intensive querying.

**Column-oriented databases work flow**

Relational databases have a set schema and they function as tables of rows and columns. Wide-column databases have a similar, but different schema. They also have rows and columns. However, they are not fixed within a table, but have a dynamic schema. Each column is stored separately. If there are similar (related) columns, they are joined into column families and then the column families are stored separately from other column families.

The row key is the first column in each column family, and it serves as an identifier of a row. Furthermore, each column after that has a column key (name). It identifies columns within rows and thus enables the querying of the columns. The value and the timestamp come after the column key, leaving a trace of when the data was entered or modified.

The number of columns pertaining to each row, or their name, can vary. In other words, not every column of a column family, and thus a database, has the same number of rows. In fact, even though they might share their name, each column is contained within one row and does not run across all rows.

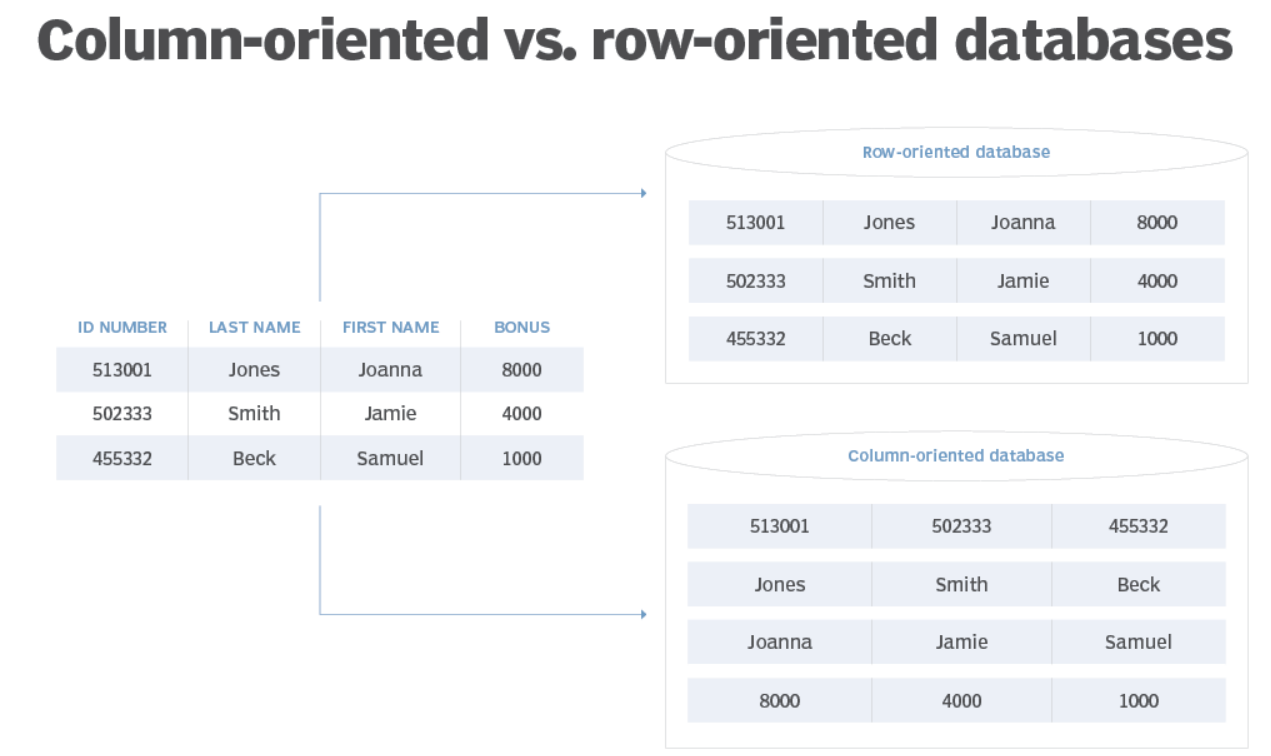
****

Fig. Column-oriented databases use vertical organization as opposed to the horizontal layout of row databases.

Those who have encountered relational databases know that each column of a relational database has the same number of rows, but it happens that some of the fields have a null value, or they appear to be empty. With wide-column databases, rather than being empty, these rows simply do not exist for a particular column.

The column families are located in a keyspace. Each keyspace holds an entire NoSQL data store and it has a similar role or importance that a schema has for a relational database. However, as NoSQL datastores have no set structure, keyspaces represent a schemaless database containing the design of a data store and its own set of attributes.

One of the most popular columnar databases available is MariaDB. It was created as a fork of MySQL intended to be robust and scalable, handle many different purposes and a large volume of queries. Apache Cassandra is another example of a columnar database handling heavy data loads across numerous servers, making the data highly available. Some of the other names on this list include Apache HBase, Hypertable and Druid specially designed for analytics. These databases support certain features of platforms such as Outbrain, Spotify and Facebook.

**Column family types**

* **Standard column family.**This column family type is similar to a table; it contains a key-value pair where the key is the row key, and the values are stored in columns using their names as their identifiers.
* **Super column family.** A super column represents an array of columns. Each super column has a name and a value mapping the super column out to several different columns. Related super columns are joined under a single row into super column families. Compared to a relational database, this is like a view of several different tables within a database. Imagine you had the view of the columns and values available for a single row -- that is a single identifier across many different tables -- and were able to store them all in one place: That is the super column family.

**Advantages of column-oriented databases**

* **Scalability.** This is a major advantage and one of the main reasons this type of database is used to store big data. With the ability to be spread over hundreds of different machines depending on the scale of the database, it supports massively parallel processing. This means it can employ many processors to work on the same set of computations simultaneously.
* **Compression.** Not only are they infinitely scalable, but they are also good at compressing data and thus saving storage.
* **Very responsive.** The load time is minimal, and queries are performed fast, which is expected given that they are designed to hold big data and be practical for analytics.

**Disadvantages of column-oriented databases**

* **Online transactional processing.** These databases are not very efficient with online transactional processing as much as they are for online analytical processing. This means they are not very good with updating transactions but are designed to analyze them. This is why they can be found holding data required for business analysis with a relational database storing data in the back end.
* **Incremental data loading.**As mentioned above, typically column-oriented databases are used for analysis and are quick to retrieve data, even when processing complex queries, as it is kept close together in columns. While incremental data loads are not impossible, columnar databases do not perform them in the most efficient way. The columns first need to be scanned to identify the right rows and scanned further to locate the modified data which requires overwriting.
* **Row-specific queries.**Like the potential downfalls mentioned above, it all boils down to the same issue, which is using the right type of database for the right purposes. With row-specific queries, you are introducing an extra step of scanning the columns to identify the rows and then locating the data to retrieve. It takes more time to get to individual records scattered in multiple columns, rather than accessing grouped records in a single column. Frequent row-specific queries might cause performance issues by slowing down a column-oriented database, which is particularly designed to help you get to required pieces of information quickly, thus defeating its purpose.

NoSQL databases are mostly designed to fit specific purposes and are not expected to work as a general type of storage. Wide-column databases are column-oriented rather than row-oriented and are intended to store and query big data.

**Graph databases**

A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships.

A graph database is optimized to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.Very few real-world business systems can survive solely on graph queries. As a result, graph databases are usually run alongside other more traditional databases. Use cases include fraud detection, social networks, and knowledge graphs.

**Graph Based Data Model**

The semantic graph database is a type of NoSQL graph database that is capable of integrating heterogeneous data from many sources and making links between datasets.

The semantic graph database, also referred to as an [RDF triplestore](https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/), focuses on the relationships between entities and is able to infer new knowledge out of existing information. It is a powerful tool to use in relationship-centered analytics and knowledge discovery.

In addition, the capability to handle massive datasets and the schema-less approach support the NoSQL semantic graph database usage in real-time big data analytics.

* In relational databases, the need to have the schemas defined before adding new information restricts data integration from new sources because the whole schema needs to be changed anew.
* With the schema-less NoSQL semantic graph database with no need to change schemas every time a new data source is about to be added, enterprises integrate data with less effort and cost.

The semantic graph database stands out from the other types of graph databases with its ability to additionally support rich semantic data schema, the so-called ontologies.

The semantic NoSQL graph database gets the best of both worlds: on the one hand, data is flexible because it does not depend on the schema. On the other hand, ontologies give the semantic graph database the freedom and ability to build logical models any way organizations find it useful for their applications, without having to change the data.

**Advantages of Graph Data Model:**

* **Structure:**The structures are very agile and workable too.
* **Explicit Representation:** The portrayal of relationships between entities is explicit.
* **Real-time O/P Results:**Query gives us real-time output results.

**Disadvantages of Graph Data Model:**

* **No standard query language:** Since the language depends on the platform that is used so there is no certain standard query language.
* **Unprofessional Graphs:**Graphs are very unprofessionalfor transactional-based systems.
* **Small User Base:**The user base is small which makes it very difficult to get support when running into a system.

**Applications of Graph Data Model:**

* Graph data models are very much used in fraud detection which itself is very much useful and important.
* It is used in Digital asset management which provides a scalable database model to keep track of digital assets.
* It is used in Network management whichalerts a network administrator about problems in a network.
* It is used in Context-aware services by giving traffic updates and many more.
* It is used in Real-Time Recommendation Engines which provide a better user experience

**SCHEMA LESS DATABASES**

Traditional relational databases are well-defined, using a schema to describe every functional element, including tables, rows views, indexes, and relationships. By exerting a high degree of control, the database administrator can improve performance and prevent capture of low-quality, incomplete, or malformed data. In a SQL database, the schema is enforced by the Relational Database Management System (RDBMS) whenever data is written to disk.

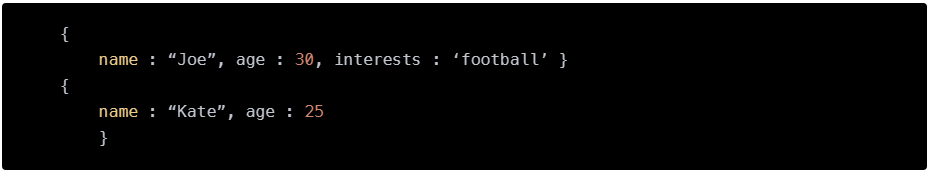
But in order to work, data needs to be heavily formatted and shaped to fit into the table structure. This means sacrificing any undefined details during the save, or storing valuable information outside the database entirely.

A schemaless database, like MongoDB, does not have these up-front constraints, mapping to a more ‘natural’ database. Even when sitting on top of a data lake, each document is created with a partial schema to aid retrieval. Any formal schema is applied in the code of your applications; this layer of abstraction protects the raw data in the NoSQL database and allows for rapid transformation as your needs change.

Any data, formatted or not, can be stored in a non-tabular NoSQL type of database. At the same time, using the right tools in the form of a schemaless database can unlock the value of all of your structured and unstructured data types.

**How does a schemaless database work?**

In schemaless databases, information is stored in JSON-style documents which can have varying sets of fields with different data types for each field. So, a collection could look like this:



With the schemaless MongoDB database, there is some additional structure — the system namespace contains an explicit list of collections and indexes. Collections may be implicitly or explicitly created — indexes must be explicitly declared.

benefits of using a schemaless database?

* Greater flexibility over data types

By operating without a schema, schemaless databases can store, retrieve, and query any data type — perfect for big data analytics and similar operations that are powered by unstructured data. Relational databases apply rigid schema rules to data, limiting what can be stored.

* **No pre-defined database schemas**

The lack of schema means that your NoSQL database can accept any data type — including those that you do not yet use. This future-proofs your database, allowing it to grow and change as your data-driven operations change and mature.

* **No data truncation**

A schemaless database makes almost no changes to your data; each item is saved in its own document with a partial schema, leaving the raw information untouched. This means that every detail is always available and nothing is stripped to match the current schema. This is particularly valuable if your analytics needs to change at some point in the future.

* **Suitable for real-time analytics functions**

With the ability to process unstructured data, applications built on NoSQL databases are better able to process real-time data, such as readings and measurements from IoT sensors. Schemaless databases are also ideal for use with machine learning and artificial intelligence operations, helping to accelerate automated actions in your business.

* **Enhanced scalability and flexibility**

With NoSQL, you can use whichever data model is best suited to the job. Graph databases allow you to view relationships between data points, or you can use traditional wide table views with an exceptionally large number of columns. You can query, report, and model information however you choose. And as your requirements grow, you can keep adding nodes to increase capacity and power.

When a record is saved to a relational database, anything (particularly metadata) that does not match the schema is truncated or removed. Deleted at write, these details cannot be recovered at a later point in time.

**2.4.1 MATERIALIZED VIEW**

A materialized view is a particular type of database object that contains any results derived from a query. Think of this like a replica of a target master from a particular moment in time. Materialized views are precomputed. They will periodically cache query results to improve a database’s performance.

Depending on the situation it could be a local copy of data that is stored somewhere remotely, or it could be the product of a join result, or it could even be the summary of said data that was created using an aggregate function.

Materialized views were first implemented by the Oracle Database and have been available in every version from 8i on. Additional environments that support materialized views include PostgreSQL, SQL Server, Sybase SQL Anywhere, Big Query, and more.  
A View is a virtual relation that acts as an actual relation. It is not a part of logical relational model of the database system. Tuples of the view are not stored in the database system and tuples of the view are generated every time the view is accessed. Query expression of the view is stored in the databases system.

Views can be used everywhere were we can use the actual relation. Views can be used to create custom virtual relations according to the needs of a specific user. We can create as many views as we want in a databases system.  
When the results of a view expression are stored in a database system, they are called materialized views. SQL does not provide any standard way of defining materialized view, however some database management system provides custom extensions to use materialized views. The process of keeping the materialized views updated is known as view maintenance.

Database system uses one of the three ways to keep the materialized view updated:

* Update the materialized view as soon as the relation on which it is defined is updated.
* Update the materialized view every time the view is accessed.
* Update the materialized view periodically.

Materialized view is useful when the view is accessed frequently, as it saves the computation time, as the result is stored in the database beforehand. Materialized view can also be helpful in case where the relation on which view is defined is very large and the resulting relation of the view is very small. Materialized view has storage cost and updation overheads associated with it.

**Materialized View Examples**

For example, let’s say you have a database with two tables: one contains the number of employees in your business, and the other contains the number of departments in your business.

Using a materialized view, you could query the database to retrieve all the employees who are associated with a particular department.

Or, say you have a database with two tables: one for the total number of sales you’ve made and one for the total amount of revenue you’re generating. You could use a materialized view to see how much revenue each sale brings with it in real-time.

**Why Use Materialized Views**

Every single time you query a database, you always accrue some type of cost. Even something as seemingly straightforward as a query still means that you are parsing, validating, planning, optimizing, and executing that query, which equates to CPU time, memory usage, opportunity costs, and more.

As your application continues to grow and become more demanding, naturally you’re looking for opportunities to reduce those costs as much as possible.

**This is where materialized views come into play.**

Because the results of a materialized view are *maintained in memory*, they are only updated when it is expressly necessary to do so. Because of that, they can significantly reduce your overall costs as opposed to querying tables or using logical views.

**Pros of Materialized Views**

* Improve performance by precomputing expensive operations
* Increase the speed of queries on very large databases
* Efficiently execute expensive queries or expensive parts of your queries

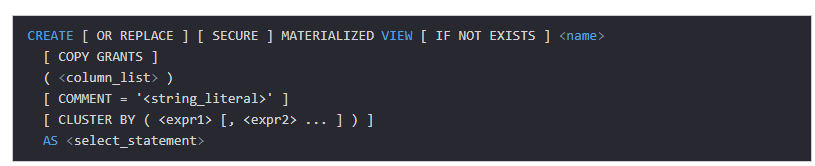
**Cons of Materialized Views**

* Not every database type supports materialized views
* Materialized views are read-only
* You cannot create keys, constraints, triggers, or articles

**How to create a Materialized View**

To create a materialized view in the tool you’re working with, you can use a DML statement to create a basic table, to load data into it, and to create a materialized view as a result.

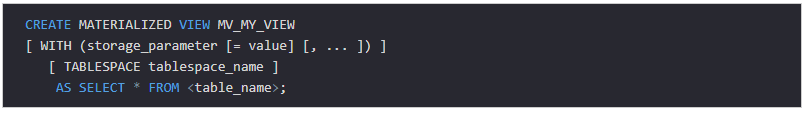
An example of an SQL command that could be used to create a materialized view is as follows:



An example of a command in **Oracle** that could be used to create a materialized view is as follows:



An example of a command for **PostgreSQL** that could be used to create a materialized view is as follows:



Materialized views are supported in the following features:

**Forward Engineering**

When you forward-engineer your data model to generate a database, the SQL code to define the materialized view is generated to the database.

**Reverse Engineering**

When you reverse-engineer an existing database that includes one or more materialized views, each materialized view is imported, the syntax is parsed, and where possible the relationships to the tables referenced by the materialized view are created.

**Complete Compare**

When you use complete compare and update a materialized view either in the model or in the database, you can keep the model materialized view specification in sync with the database table specification.

When you add a materialized view to a model, the materialized view is represented as a box with rounded corners. A relationship between a table and a materialized view indicates that the materialized view table references one or more of the columns from that table.

A materialized view column can be a reference to a table column or a user-defined expression.

* For materialized view columns that are references to a table column, the following naming convention is used:  
  **<MaterializedViewColumnName>: <SourceTableName>.<SourceColumnName>**
* For materialized view columns that are user-defined expressions, the following naming convention is used:  
  **<MaterializedViewColumnName >: <expression>**

After you draw a view relationship between a table and a materialized view, you must use on-diagram editing or the Views editor to migrate columns in the table to the materialized view. By default, the <MaterializedViewColumnName> is the same as the <SourceColumnName> until you edit the materialized view column name.

You can edit materialized views directly in the diagram window using on-diagram editing. When you drag a column from a table into a materialized view, the materialized view column and the relationship are created.

If you delete a table column that is referenced by a materialized view, the corresponding materialized view column is deleted. If you delete a table that is referenced by a materialized view, the corresponding materialized view columns are deleted.

**2.4.2 DISTRIBUTION MODEL**

The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on. A more appealing option is to scale out—run the database on a cluster of servers. Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.

Depending on your distribution model, you can get a data store that will give you the ability to handle larger quantities of data, the ability to process a greater read or write traffic, or more availability in the face of network slowdowns or breakages. These are often important benefits, but they come at a cost. Running over a cluster introduces complexity—so it’s not something to do unless the benefits are compelling.

Broadly, there are two paths to data distribution: replication and sharding. Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes.

Replication and sharding are orthogonal techniques: You can use either or both of them. Replication comes into two forms: master-slave and peer-to-peer. We will now discuss these techniques starting at the simplest and working up to the more complex: first single-server, then master-slave replication, then sharding, and finally peer-to-peer replication.

* + - 1. **Single Server**

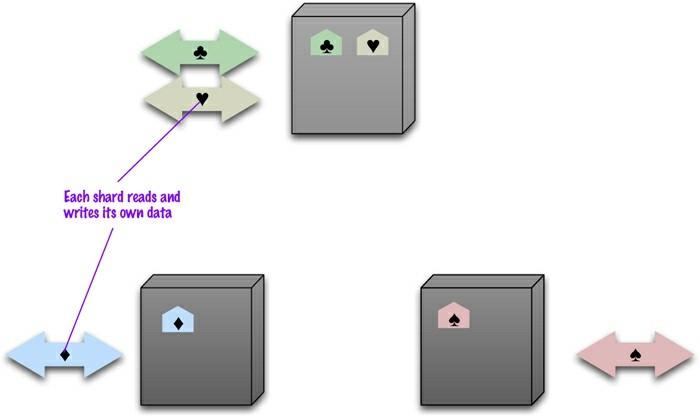
The first and the simplest distribution option is the one we would most often recommend—no distribution at all. Run the database on a single machine that handles all the reads and writes to the data store. We prefer this option because it eliminates all the complexities that the other options introduce; it’s easy for operations people to manage and easy for application developers to reason about.

Although a lot of NoSQL databases are designed around the idea of running on a cluster, it can make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application. Graph databases are the obvious category here—these work best in a single-server configuration. If your data usage is mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it’s easier on application developers.

For the rest of this chapter we’ll be wading through the advantages and complications of more sophisticated distribution schemes. Don’t let the volume of words fool you into thinking that we would prefer these options. If we can get away without distributing our data, we will always choose a single-server approach.

* + - 1. **Sharding**

Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers—a technique that’s called **sharding.**



**Figure 2.4.2.2. Sharding puts different data on separate nodes, each of which does its own reads and writes.**

In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server. The load is balanced out nicely between servers—for example, if we have ten servers, each one only has to handle 10% of the load.

In order to get close to it we have to ensure that data that’s accessed together is clumped together on the same node and that these clumps are arranged on the nodes to provide the best data access. The first part of this question is how to clump the data up so that one user mostly gets her data from a single server. This is where aggregate orientation comes in really handy. The whole point of aggregates is that we design them to combine data that’s commonly accessed together—so aggregates leap out as an obvious unit of distribution.

When it comes to arranging the data on the nodes, there are several factors that can help improve performance. If you know that most accesses of certain aggregates are based on a physical location, you can place the data close to where it’s being accessed. If you have orders for someone who lives in Boston, you can place that data in your eastern US data center. Another factor is trying to keep the load even. This means that you should try to arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load. This may vary over time, for example if some data tends to be accessed on certain days of the week—so there may be domain-specific rules you’d like to use.

In some cases, it’s useful to put aggregates together if you think they may be read in sequence. The Bigtable paper [Chang etc.] described keeping its rows in lexicographic order and sorting web addresses based on reversed domain names (e.g., com.martinfowler). This way data for multiple pages could be accessed together to improve processing efficiency. Historically most people have done sharding as part of application logic.

Many NoSQL databases offer **auto-sharding**, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard. This can make it much easier to use sharding in an application.

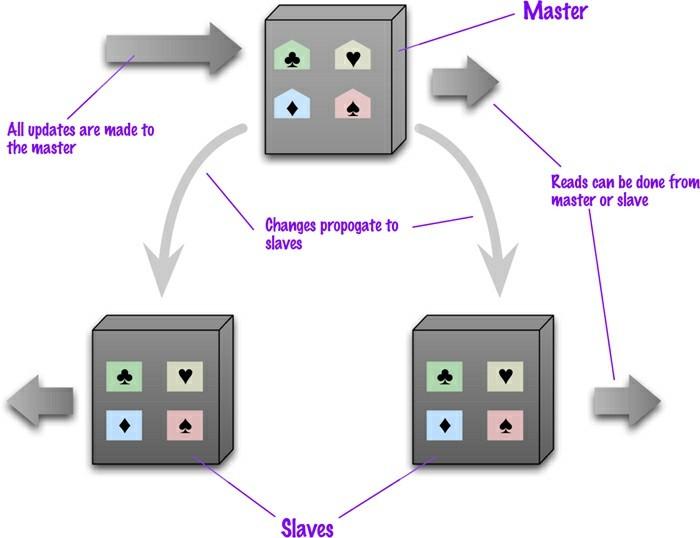
Sharding is particularly valuable for performance because it can improve both read and write performance. Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.

Sharding does little to improve resilience when used alone. Although the data is on different nodes, a node failure makes that shard’s data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it’s not good to have a database with part of its data missing. With a single server it’s easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you’re more likely to get a node failure. So in practice, sharding alone is likely to decrease resilience.

Despite the fact that sharding is made much easier with aggregates Some databases are intended from the beginning to use sharding, in which case it’s wise to run them on a cluster from the very beginning of development, and certainly in production. Other databases use sharding as a deliberate step up from a single-server configuration, in which case it’s best to start single-server and only use sharding once your load projections clearly indicate that you are running out of headroom.

**Master-Slave Replication**

With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master.



**Fig. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.**

Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves. You are still, however, limited by the ability of the master to process updates and its ability to pass those updates on. Consequently it isn’t such a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a bit with handling the write load.

A second advantage of master-slave replication is read resilience: Should the master fail, the slaves can still handle read requests. Again, this is useful if most of your data access is reads. The failure of the master does eliminate the ability to handle writes until either the master is restored or a new master is appointed. However, having slaves as replicates of the master does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.

The ability to appoint a slave to replace a failed master means that master-slave replication is useful even if you don’t need to scale out. All read and write traffic can go to the master while the slave acts as a hot backup. In this case it’s easiest to think of the system as a single-server store with a hot backup. You get the convenience of the single-server configuration but with greater resilience which is particularly handy if you want to be able to handle server failures gracefully.

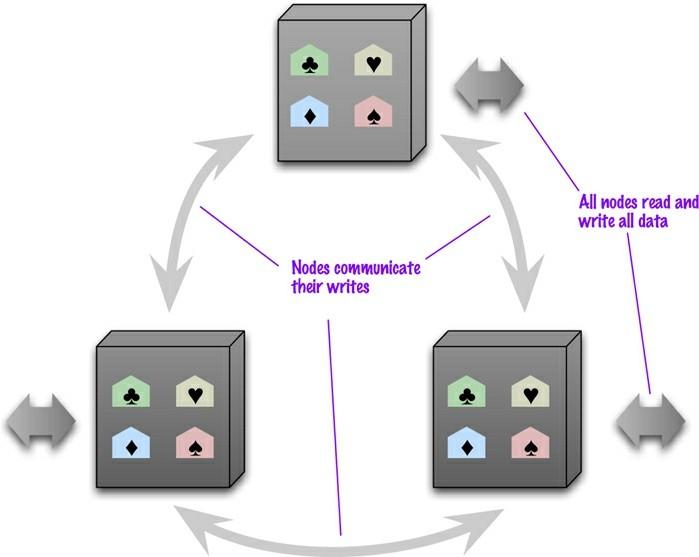
Masters can be appointed manually or automatically. Manual appointing typically means that when you configure your cluster, you configure one node as the master. With automatic appointment, you create a cluster of nodes and they elect one of themselves to be the master. Apart from simpler configuration, automatic appointment means that the cluster can automatically appoint a new master when a master fails, reducing downtime.

In order to get read resilience, you need to ensure that the read and write paths into your application are different, so that you can handle a failure in the write path and still read. This includes such things as putting the reads and writes through separate database connections a facility that is not often supported by database interaction libraries. As with any feature, you cannot be sure you have read resilience without good tests that disable the writes and check that reads still occur.

Replication comes with some alluring benefits, but it also comes with an inevitable dark side inconsistency. You have the danger that different clients, reading different slaves, will see different values because the changes haven’t all propagated to the slaves. In the worst case, that can mean that a client cannot read a write it just made. Even if you use master-slave replication just for hot backup this can be a concern, because if the master fails, any updates not passed on to the backup are lost.

**Peer to Peer Replication**

Master-slave replication helps with read scalability but doesn’t help with scalability of writes. It provides resilience against failure of a slave, but not of a master. Essentially, the master is still a bottleneck and a single point of failure. Peer-to-peer replication attack these problems that does not have master. All the replicas have equal weight, they can all accept writes and the loss of any of them doesn’t prevent access to the data stores.

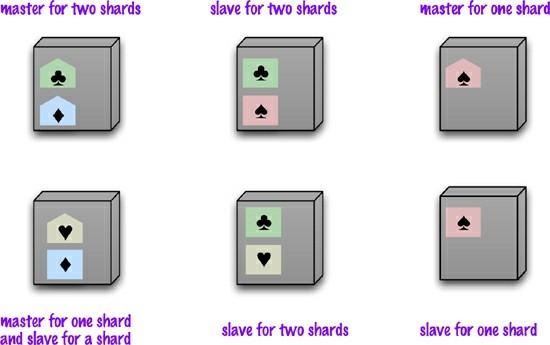


**Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.**

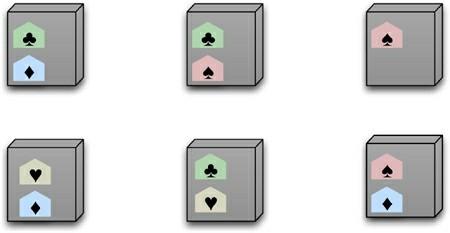
With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance. There’s much to like here—but there are complications.

The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient. Inconsistent writes are forever.

**Combining Sharding and Replication**

Replication and sharding are strategies that can be combined. If we use both master-slave replication and sharding this means that we have multiple masters, but each data item only has a single master. Depending on your configuration, you may choose a node to be a master for some data and slaves for others, or you may dedicate nodes for master or slave duties.

**Figure 4.4. Using master-slave replication together with sharding**

Using peer-to-peer replication and sharding is a common strategy for column-family databases. In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them. A good starting point for peer-to-peer replication is to have a replication factor of 3, so each shard is present on three nodes. Should a node fail, then the shards on that node will be built on the other nodes

**Figure. Using peer-to-peer replication together with sharding**

**CONSISTENCY**

One of the biggest changes from a centralized relational database to a cluster-oriented NoSQL database is in how you think about consistency. Relational databases try to exhibit strong consistency by avoiding all the various inconsistencies that we’ll shortly be discussing. Once you start looking at the NoSQL world, phrases such as “CAP theorem” and “eventual consistency” appear, and as soon as you start building something you have to think about what sort of consistency you need for your system.

Consistency in NoSQL databases refers to the level of data consistency maintained by the database system in the presence of concurrent read and write operations. Unlike traditional relational databases that typically adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties, NoSQL databases often prioritize other characteristics like scalability, availability, and partition tolerance, as defined by the CAP theorem (Consistency, Availability, Partition Tolerance).

The choice of consistency model depends on the specific requirements of the application and the trade-offs the developers are willing to make. Different NoSQL databases offer different consistency guarantees, and it's essential to select the appropriate database based on your application's needs.

It's worth noting that many NoSQL databases allow developers to adjust the consistency levels based on their requirements. For example, some databases offer tunable consistency levels that allow developers to configure the level of consistency on a per-operation basis. This flexibility allows developers to strike a balance between data consistency and system performance according to their application's demands.

**Update Consistency**

We’ll begin by considering updating a telephone number. Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. Implausibly, they both have update access, so they both go in at the same time to update the number. To make the example interesting, we’ll assume they update it slightly differently, because each uses a slightly different format. This issue is called a write-write conflict: two people updating the same data item at the same time.

When the writes reach the server, the server will serialize them—decide to apply one, then the other. Let’s assume it uses alphabetical order and picks Martin’s update first, then Pramod’s. Without any concurrency control, Martin’s update would be applied and immediately overwritten by Pramod’s. In this case Martin’s is a lost update. Here the lost update is not a big problem, but often it is. We see this as a failure of consistency because Pramod’s update was based on the state before Martin’s update, yet was applied after it.

Approaches for maintaining consistency in the face of concurrency are often described as pessimistic or optimistic. A pessimistic approach works by preventing conflicts from occurring; an optimistic approach lets conflicts occur, but detects them and takes action to sort them out. For update conflicts, the most common pessimistic approach is to have write locks, so that in order to change a value you need to acquire a lock, and the system ensures that only one client can get a lock at a time.

So Martin and Pramod would both attempt to acquire the write lock, but only Martin (the first one) would succeed. Pramod would then see the result of Martin’s write before deciding whether to make his own update.

A common optimistic approach is a conditional update where any client that does an update tests the value just before updating it to see if it’s changed since his last read. In this case, Martin’s update would succeed but Pramod’s would fail. The error would let Pramod know that he should look at the value again and decide whether to attempt a further update.

Both the pessimistic and optimistic approaches rely on a consistent serialization of the updates. With a single server, this is obvious—it has to choose one, then the other. But if there’s more than one server, such as with peer-to-peer replication, then two nodes might apply the updates in a different order, resulting in a different value for the telephone number on each peer.

**Read Consistency**

Read Consistency in NoSQL databases refers to the guarantee that a read operation will always return the most recent and up-to-date data from the database. In distributed systems, such as NoSQL databases, achieving strong consistency (as in traditional ACID-compliant databases) across all nodes can be challenging due to factors like network latency, node failures, and the need for horizontal scalability. As a result, NoSQL databases often adopt a more relaxed form of consistency known as eventual consistency.

Eventual Consistency: In an eventually consistent system, after a write operation is performed, the data may not be immediately propagated to all replicas or nodes in the cluster. Instead, the system allows for a delay, during which the data will eventually be propagated and reconciled across all nodes. This means that if a read operation occurs shortly after a write operation, it might return a slightly outdated value until the data reaches all replicas and becomes fully consistent.

In practical terms, eventual consistency implies that, given enough time and assuming no further writes, all replicas will eventually converge to the same value. The time it takes to achieve consistency depends on various factors, such as network conditions, system load, and the specific NoSQL database's implementation.

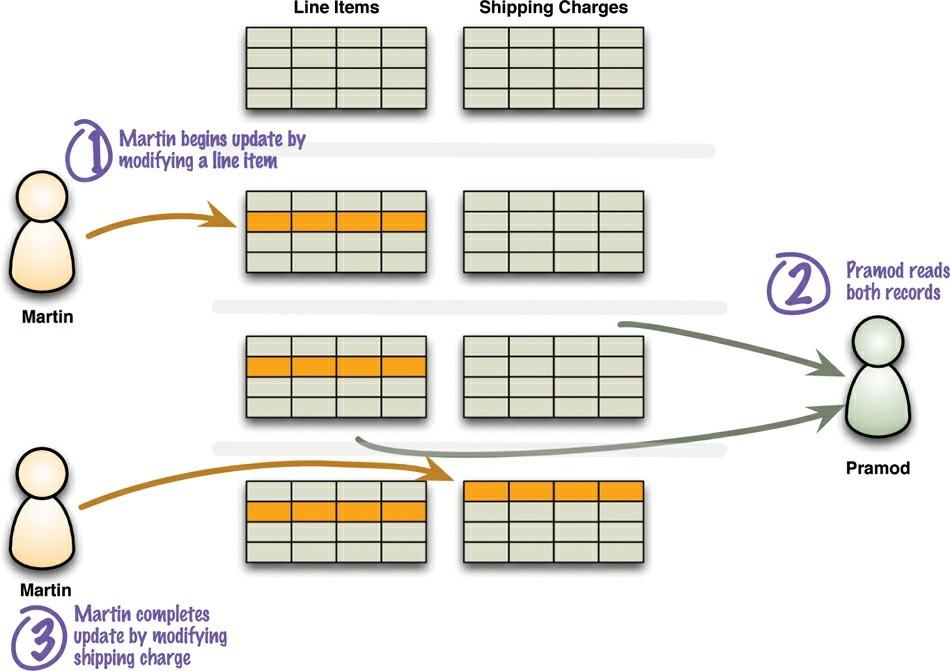
Use Cases for Eventual Consistency: Eventual consistency is appropriate for certain types of applications where the immediate consistency of data across all replicas is not critical, and some level of temporary inconsistency is acceptable. For example:

1. Social media platforms, where the order of likes, comments, and updates might not be immediately reflected on all users' devices but eventually converge to the correct state.
2. Analytical applications that perform big data processing, where slight delays in propagating data are acceptable for handling large volumes of data efficiently.
3. Collaborative applications where users can edit the same document concurrently, and the system resolves conflicts in an eventual manner.

Tunable Consistency Levels: Many NoSQL databases allow developers to choose their preferred consistency levels based on the specific use case. This means that developers can select stronger consistency models (e.g., strong or causal consistency) when needed or opt for weaker consistency (e.g., eventual consistency) for better availability and performance in other scenarios. This flexibility allows developers to tailor the database's behavior to the application's requirements.

It's essential for developers to understand the consistency guarantees provided by their chosen NoSQL database and design their applications to handle potential inconsistencies gracefully when working with distributed data systems.

Having a data store that maintains update consistency is one thing, but it doesn’t guarantee that readers of that data store will always get consistent responses to their requests.

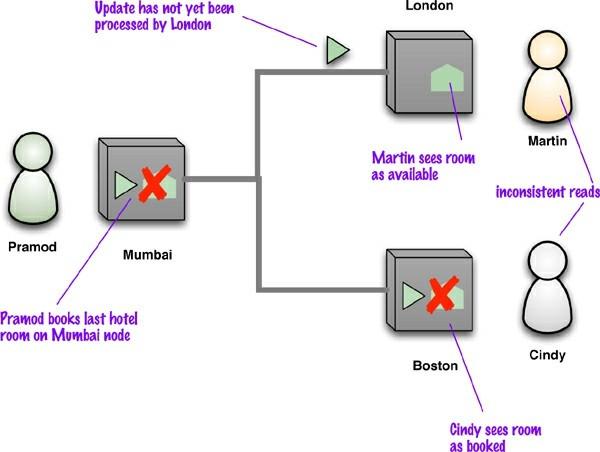


**Fig. Inconsistent read or read-write conflict**

**Logical consistency:** ensuring that different data items make sense together. To avoid a logically inconsistent read-write conflict, relational databases support the notion of transactions. A common claim we hear is that NoSQL databases don’t support transactions and thus can’t be consistent. Such claim is mostly wrong because it glosses over lots of important details. Our first clarification is that any statement about lack of transactions usually only applies to some NoSQL databases, in particular the aggregate-oriented ones. In contrast, graph databases tend to support ACID transactions just the same as relational databases.

Secondly, aggregate-oriented databases do support atomic updates, but only within a single aggregate. This means that you will have logical consistency within an aggregate but not between aggregates. So in the example, you could avoid running into that inconsistency if the order, the delivery charge, and the line items are all part of a single order aggregate.

Of course not all data can be put in the same aggregate, so any update that affects multiple aggregates leaves open a time when clients could perform an inconsistent read. The length of time an inconsistency is present is called the **inconsistency window**. A NoSQL system may have a quite short inconsistency window: As one data point, Amazon’s documentation says that the inconsistency window for its SimpleDB service is usually less than a second.

**Replication consistency:** ensuring that the same data item has the same value when read from different replicas

**Fig An example of replication inconsistency**

Replication consistency in NoSQL databases refers to the consistency guarantees provided when data is replicated across multiple nodes or replicas in a distributed database system. Replication is a fundamental technique used in NoSQL databases to achieve high availability, fault tolerance, and scalability. The consistency level in replication determines how and when data changes are propagated to replicas and how concurrent read and write operations are handled.

There are several common replication consistency models in NoSQL databases:

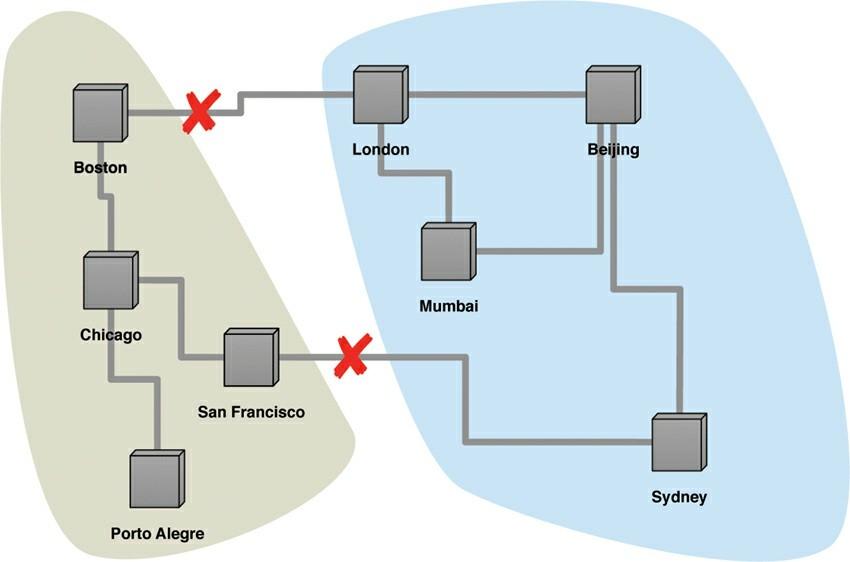
1. **Strong Consistency:** In a strongly consistent replication model, all replicas are updated synchronously, and each read operation from any replica will always return the most recent write. This level of consistency is similar to what traditional ACID-compliant databases offer. While strong consistency ensures data integrity and accuracy, it can result in higher latency and reduced availability during network partitions or node failures.
2. **Eventual Consistency:** In an eventually consistent replication model, replicas are allowed to be out of sync temporarily. After a write operation, replicas asynchronously synchronize with each other, and it may take some time for all replicas to converge to the same state. Consequently, read operations might return slightly outdated data until eventual convergence is achieved. Eventual consistency provides better availability and performance but allows for temporary data inconsistency.
3. **Causal Consistency:** Causal consistency is a middle ground between strong and eventual consistency. It ensures that if one operation causally affects another, the database system maintains this causality relationship across all replicas. This means that causally related operations will be observed in the same order on all replicas, but there may be temporary inconsistencies between unrelated operations.
4. **Session Consistency:** Session consistency is a consistency level that guarantees all read and write operations performed within the same session will observe consistent data. However, data might be inconsistent between different sessions or clients.
5. **Read-your-writes Consistency:** Read-your-writes consistency ensures that if a client performs a write operation, any subsequent read operation from the same client will reflect the write's effects. This consistency level is often desirable in scenarios where strong consistency is not required, but clients expect to observe their own writes immediately.

**Relaxing Consistency**

Relaxing consistency, in the context of NoSQL databases, refers to adopting a less strict or less stringent consistency model to achieve higher scalability, availability, and performance at the cost of relaxing the guarantees of strong consistency. This relaxation allows for certain forms of temporary data inconsistency, but it enhances the overall performance and fault tolerance of the system.

As mentioned earlier, NoSQL databases often prioritize the CAP theorem (Consistency, Availability, and Partition Tolerance) over ACID properties (Atomicity, Consistency, Isolation, Durability). The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three guarantees in the presence of network partitions, and therefore, designers of NoSQL databases often choose to sacrifice strong consistency in favor of better availability and partition tolerance.

**Availability** has a particular meaning in the context of CAP—it means that if you can talk to a node in the cluster, it can read and write data. That’s subtly different from the usual meaning, which we’ll explore later. **Partition tolerance** means that the cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other (situation known as a **split brain**)



**Figure With two breaks in the communication lines, the network partitions into two groups.**

A single-server system is the obvious example of a CA system—a system that has Consistency and Availability but not Partition tolerance. A single machine can’t partition, so it does not have to worry about partition tolerance. There’s only one node—so if it’s up, it’s available. Being up and keeping consistency is reasonable

**Relaxing Durability**

As it turns out, there are cases where you may want to trade off some durability for higher performance. If a database can run mostly in memory, apply updates to its in-memory representation, and periodically flush changes to disk, then it may be able to provide substantially higher responsiveness to requests. The cost is that, should the server crash, any updates since the last flush will be lost.

**Quorums**

In the context of NoSQL databases, "quorums" refer to the minimum number of nodes or replicas that must participate in read and write operations to achieve a specific level of data consistency and availability. Quorums are used in distributed databases to ensure that a sufficient number of replicas acknowledge and agree on an operation to guarantee certain consistency guarantees while tolerating node failures or network partitions.

Quorums play a crucial role in maintaining data consistency and fault tolerance in distributed systems, especially in databases that use replication for high availability and data redundancy. Different NoSQL databases use different quorum strategies based on their underlying architecture and consistency models.

There are two primary types of quorums used in NoSQL databases:

1. Read Quorum: In a read quorum, a certain number of replicas must participate in a read operation before a response is considered valid. The read quorum size determines how consistent the data will be for read operations. There are typically two types of read quorums:

a. Strong Read Quorum: Requires all replicas to participate in the read operation. This ensures strong consistency because the read operation will return the most recent data available in the system. However, this approach might lead to higher latency, especially in the presence of network partitions or node failures.

b. Eventual Read Quorum: Requires only a subset of replicas to participate in the read operation. This allows for eventual consistency, where the read might return slightly outdated data until all replicas converge. Eventual read quorums provide better read availability and lower latency.

1. Write Quorum: In a write quorum, a certain number of replicas must participate in a write operation before it is considered successful. The write quorum size determines how many replicas need to acknowledge a write before it is considered durable. There are also two main types of write quorums:

a. Strict or Synchronous Write Quorum: Requires all replicas to acknowledge the write operation before it is considered successful. This ensures that the write is committed to all replicas before acknowledging the client, providing strong consistency but potentially increasing write latency.

b. Sloppy or Asynchronous Write Quorum: Requires only a subset of replicas to acknowledge the write operation before it is considered successful. This allows for higher write availability and lower write latency at the cost of eventual consistency.

The choice of read and write quorum sizes depends on the consistency model desired, the desired level of fault tolerance, and the trade-offs between consistency, availability, and performance for the specific application.

It's important to note that not all NoSQL databases use quorums, as they are primarily associated with distributed databases that use replication strategies to ensure data availability and consistency.

**CASSANDRA**

Cassandra is a popular NoSQL distributed database known for its scalability, fault tolerance, and high availability. Its data model is designed to handle large amounts of data across multiple nodes while providing a flexible schema and excellent read and write performance. Cassandra follows a "wide-column" data model, which is also often referred to as a "distributed multi-dimensional map."

**Cassandra** is a distributed database management system which is open source with wide column store, NoSQL database to handle large amount of data across many commodity servers which provides high availability with no single point of failure. It is written in Java and developed by Apache Software Foundation.

The design goal of a Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes of the cluster.

All the nodes of Cassandra in a cluster play the same role. Each node is independent, at the same time interconnected to other nodes. Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster. When a node goes down, read/write request can be served from other nodes in the network.

Key features of the Cassandra data model include:

1. Distributed Architecture: Cassandra is designed to operate in a distributed manner across a cluster of nodes. Each node can hold a subset of the data, and the data is partitioned across nodes using a hash function on the primary key. This allows Cassandra to scale horizontally by adding more nodes to the cluster.
2. Column Families (Tables): In Cassandra, data is organized into "column families," which are analogous to tables in a relational database. Each column family can have different columns, and rows are identified by a primary key. Unlike traditional relational databases, Cassandra does not enforce a fixed schema for each row, allowing great flexibility in the data structure.
3. Composite Keys: The primary key in Cassandra can be a composite key consisting of multiple columns. This allows for efficient querying on multiple dimensions and offers flexibility in data modeling.
4. Columns and Rows: Each row in a column family consists of multiple columns. Columns can have different names and data types, even within the same column family. Rows are identified by their unique primary keys and are distributed across the cluster based on the hash of the primary key.
5. Wide Rows: Cassandra allows rows to have an unlimited number of columns, which means they can store a large amount of data. This design is particularly useful for applications that require high-speed read and write access to large datasets.
6. Consistency Levels: Cassandra provides tunable consistency levels, allowing developers to choose the desired level of data consistency for read and write operations. This allows developers to strike a balance between data consistency and system performance based on the application's requirements.
7. No Single Point of Failure: Cassandra is designed to be fault-tolerant. Data is replicated across multiple nodes, ensuring that the system can withstand node failures without losing data or compromising availability.
8. Secondary Indexes: Cassandra supports secondary indexes, allowing queries on non-primary key columns for efficient data retrieval based on different criteria.

The data model in Cassandra makes it well-suited for use cases that require massive scale, high availability, and low-latency data access. It is commonly used in applications that handle large volumes of time-series data, real-time analytics, and other scenarios where traditional relational databases may not be able to meet the performance and scalability requirements.

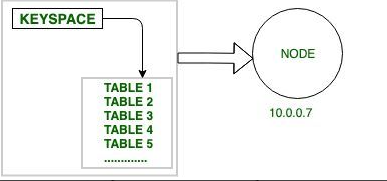
Apache Cassandra is used to manage very large amounts of structure data spread out across the world. It provides highly available service with no single point of failure. Listed below are some points of Apache Cassandra:

* It is scalable, fault-tolerant, and consistent.
* It is column-oriented database.
* Its distributed design is based on Amazon’s Dynamo and its data model on Google’s Big table.
* It is created at Facebook and it differs sharply from relational database management systems.

Cassandra implements a Dynamo-style replication model with no single point of failure but it’s add a more powerful “column family” data model. Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, eBay, Netflix, and more.

**Basic Terminology:**

1. **Node:**Node is the basic component in Apache Cassandra. It is the place where actually data is stored. For Example:As shown in diagram node which has IP address 10.0.0.7 contain data (keyspace which contain one or more tables).



**Fig Node**

1. **Data Centre:**Data Centre is a collection of nodes.

For example:

DC – N1 + N2 + N3 ….

DC: Data Centre

N1: Node 1

N2: Node 2

N3: Node 3

**3. Cluster:**

It is the collection of many data centers.

For example:

**C = DC1 + DC2 + DC3….**

C: Cluster

DC1: Data Center 1

DC2: Data Center 2

DC3: Data Center 3

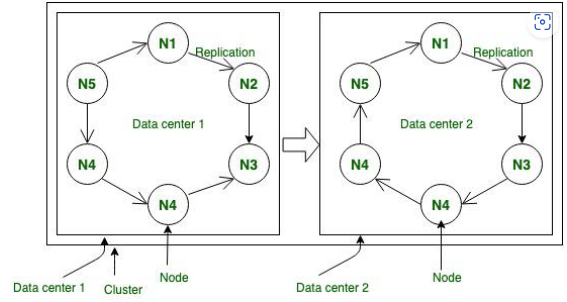
**4. Cluster:**   
It is the collection of many data centers.   
For example:

**C = DC1 + DC2 + DC3….**

C: Cluster

DC1: Data Center 1

DC2: Data Center 2

DC3: Data Center 3 

**Figure –** Node, Data center, Cluster

**Operations:**

**1. Read Operation:**

In Read Operation there are three types of read requests that a coordinator can send to a replica. The node that accepts the write requests called coordinator for that particular operation.

**Step-1: Direct Request:**

In this operation coordinator node sends the read request to one of the replicas.

Step-2: Digest Request:

In this operation coordinator will contact to replicas specified by the consistency level. For Example: CONSISTENCY TWO; It simply means that any two nodes in data center will acknowledge.

Step-3: Read Repair Request:

If there is any case in which data is not consistent across the node then background Read Repair Request initiated that makes sure that the most recent data is available across the nodes.

**2. Write Operation:**

Step-1:

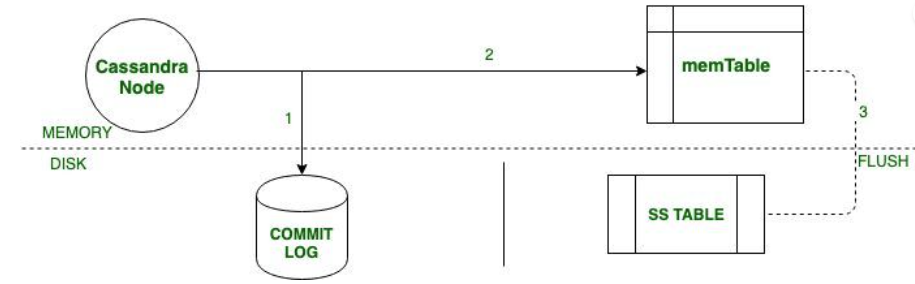
In Write Operation as soon as we receives request then it is first dumped into commit log to make sure that data is saved.

Step-2:

Insertion of data into table that is also written in MemTable that holds the data till its get full.

Step-3:

If MemTable reaches its threshold then data is flushed to SS Table.



**Figure –** Write Operation in Cassandra

**Application of Apache Cassandra:**   
Some of the application use cases that Cassandra excels in include:

* Real-time, big data workloads
* Time series data management
* High-velocity device data consumption and analysis
* Media streaming management (e.g., music, movies)
* Social media (i.e., unstructured data) input and analysis
* Online web retail (e.g., shopping carts, user transactions)
* Real-time data analytics
* Online gaming (e.g., real-time messaging)
* Software as a Service (SaaS) applications that utilize web services
* Online portals (e.g., healthcare provider/patient interactions)
* Most write-intensive systems

**CASSANDRA DATA MODELING**

Data modeling is the process of identifying entities and their relationships. In relational databases, data is placed in normalized tables with foreign keys used to reference related data in other tables. Queries that the application will make are driven by the structure of the tables and related data are queried as table joins. In Cassandra, data modeling is query-driven. The data access patterns and application queries determine the structure and organization of data which then used to design the database tables.

**Data Model Analysis**

The data model is a conceptual model that must be analyzed and optimized based on storage, capacity, redundancy and consistency. A data model may need to be modified as a result of the analysis. Considerations or limitations that are used in data model analysis include:

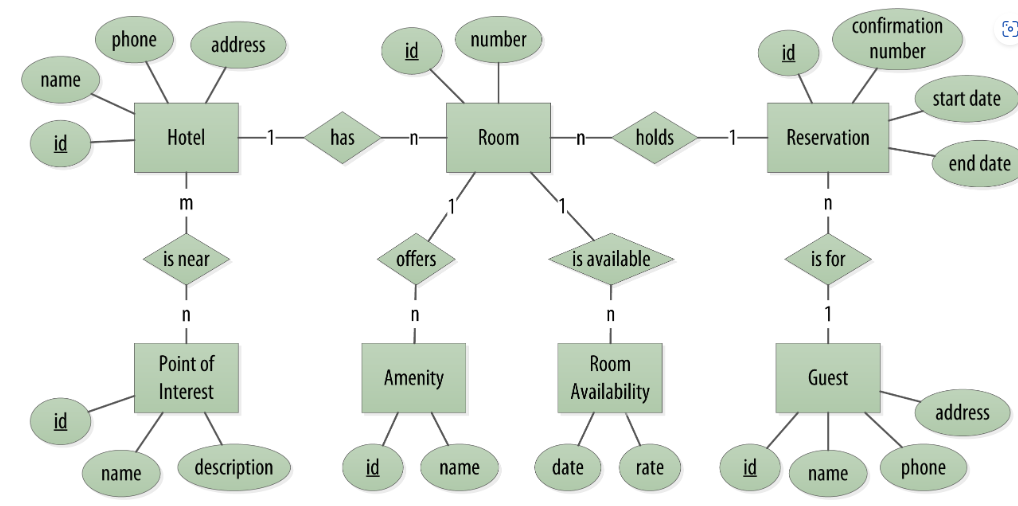
* Partition Size
* Data Redundancy
* Disk space
* Lightweight Transactions (LWT)

**Conceptual Data Modeling**

A simple domain model that is easy to understand in the relational world, and then see how you might map it from a relational to a distributed hash table model in Cassandra.

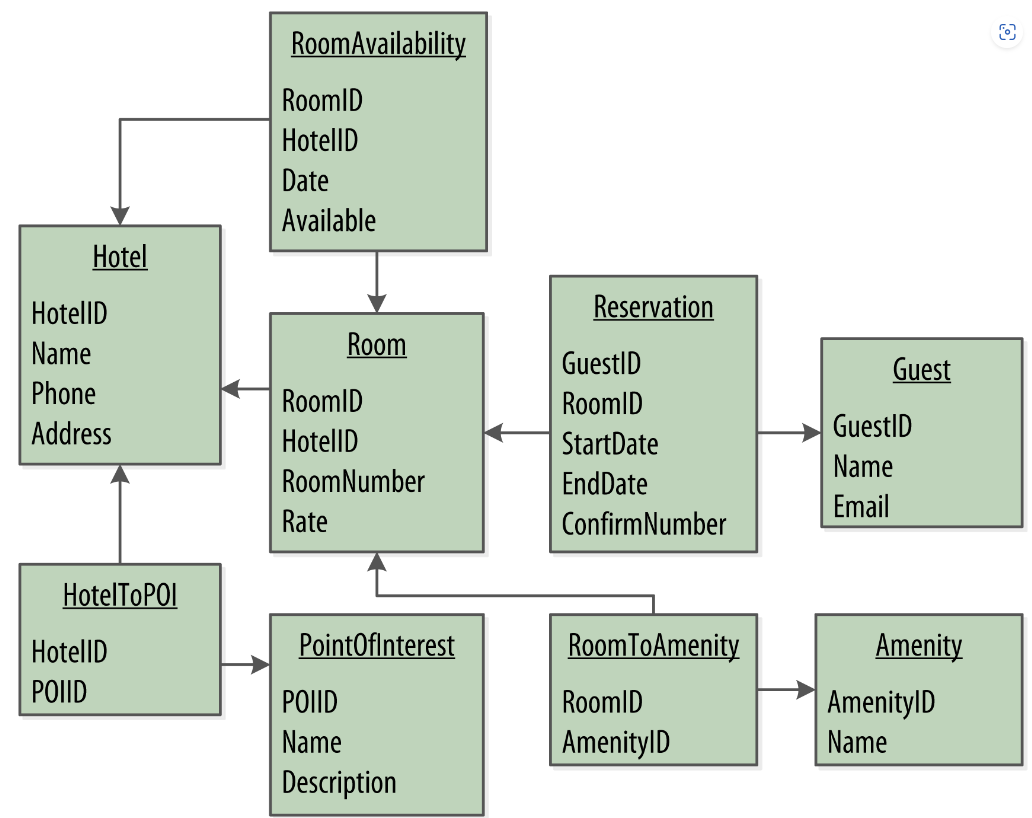
For example, let’s use a domain that is easily understood and that everyone can relate to: making hotel reservations.

The conceptual domain includes hotels, guests that stay in the hotels, a collection of rooms for each hotel, the rates and availability of those rooms, and a record of reservations booked for guests. Hotels typically also maintain a collection of “points of interest,” which are parks, museums, shopping galleries, monuments, or other places near the hotel that guests might want to visit during their stay. Both hotels and points of interest need to maintain geolocation data so that they can be found on maps for mashups, and to calculate distances.



**RDBMS Design**

When you set out to build a new data-driven application that will use a relational database, you might start by modeling the domain as a set of properly normalized tables and use foreign keys to reference related data in other tables.

****

The figure below shows how you might represent the data storage for your application using a relational database model. The relational model includes a couple of “join” tables in order to realize the many-to-many relationships from the conceptual model of hotels-to-points of interest, rooms-to-amenities, rooms-to-availability, and guests-to-rooms (via a reservation).

**Defining Application Queries**

The query-first approach to start designing the data model for a hotel application. The user interface design for the application is often a great artifact to use to begin identifying queries. Let’s assume that you’ve talked with the project stakeholders and your UX designers have produced user interface designs or wireframes for the key use cases. You’ll likely have a list of shopping queries like the following:

* Q1. Find hotels near a given point of interest.
* Q2. Find information about a given hotel, such as its name and location.
* Q3. Find points of interest near a given hotel.
* Q4. Find an available room in a given date range.
* Q5. Find the rate and amenities for a room.

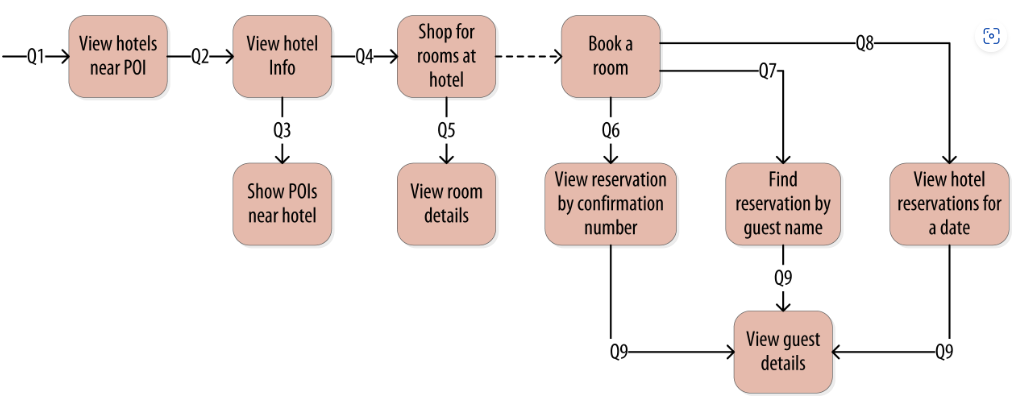
It is often helpful to be able to refer to queries by a shorthand number rather that explaining them in full. The queries listed here are numbered Q1, Q2, and so on, which is how they are referenced in diagrams throughout the example.

Now if the application is to be a success, you’ll certainly want customers to be able to book reservations at hotels. This includes steps such as selecting an available room and entering their guest information. So clearly you will also need some queries that address the reservation and guest entities from the conceptual data model. Even here, however, you’ll want to think not only from the customer perspective in terms of how the data is written, but also in terms of how the data will be queried by downstream use cases.

Your natural tendency might be to focus first on designing the tables to store reservation and guest records, and only then start thinking about the queries that would access them. You may have felt a similar tension already when discussing the shopping queries before, thinking “but where did the hotel and point of interest data come from?” Don’t worry, you will see soon enough. Here are some queries that describe how users will access reservations:

* Q6. Lookup a reservation by confirmation number.
* Q7. Lookup a reservation by hotel, date, and guest name.
* Q8. Lookup all reservations by guest name.
* Q9. View guest details.

All of the queries are shown in the context of the workflow of the application in the figure below. Each box on the diagram represents a step in the application workflow, with arrows indicating the flows between steps and the associated query. If you’ve modelled the application well, each step of the workflow accomplishes a task that “unlocks” subsequent steps. For example, the “View hotels near POI” task helps the application learn about several hotels, including their unique keys. The key for a selected hotel may be used as part of Q2, in order to obtain detailed description of the hotel. The act of booking a room creates a reservation record that may be accessed by the guest and hotel staff at a later time through various additional queries.



**Logical Data Modeling**

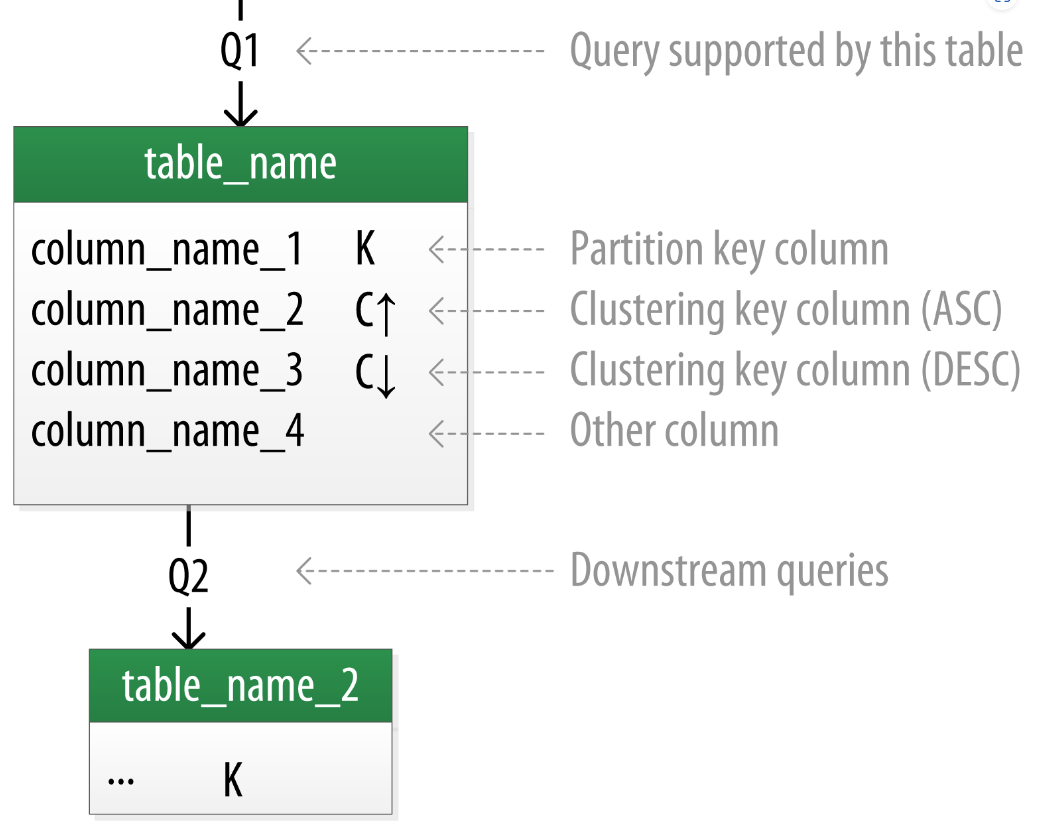
Create a logical model containing a table for each query, capturing entities and relationships from the conceptual model.

Step 1: To name each table, you’ll identify the primary entity type for which you are querying and use that to start the entity name. If you are querying by attributes of other related entities, append those to the table name, separated with *by*. For example, hotels\_by\_poi.

Step 2:  Identify the primary key for the table, adding partition key columns based on the required query attributes, and clustering columns in order to guarantee uniqueness and support desired sort ordering.

The design of the primary key is extremely important, as it will determine how much data will be stored in each partition and how that data is organized on disk, which in turn will affect how quickly Cassandra processes reads.

Complete each table by adding any additional attributes identified by the query. If any of these additional attributes are the same for every instance of the partition key, mark the column as static.



Each table is shown with its title and a list of columns. Primary key columns are identified via symbols such as K for partition key columns and C↑ or C↓ to represent clustering columns. Lines are shown entering tables or between tables to indicate the queries that each table is designed to support.

**Physical Data Modeling**

Each of the logical model tables, assigning types to each item. Use any valid CQL data type <data-types>, including the basic types, collections, and user-defined types. Identify additional user-defined types that can be created to simplify your design. After assigned data types, analyze the model by performing size calculations and testing out how the model works. Make some adjustments based on findings.

The figure includes a designation of the keyspace containing each table and visual cues for columns represented using collections and user-defined types. Note the designation of static columns and secondary index columns. There is no restriction on assigning these as part of a logical model, but they are typically more of a physical data modeling concern.

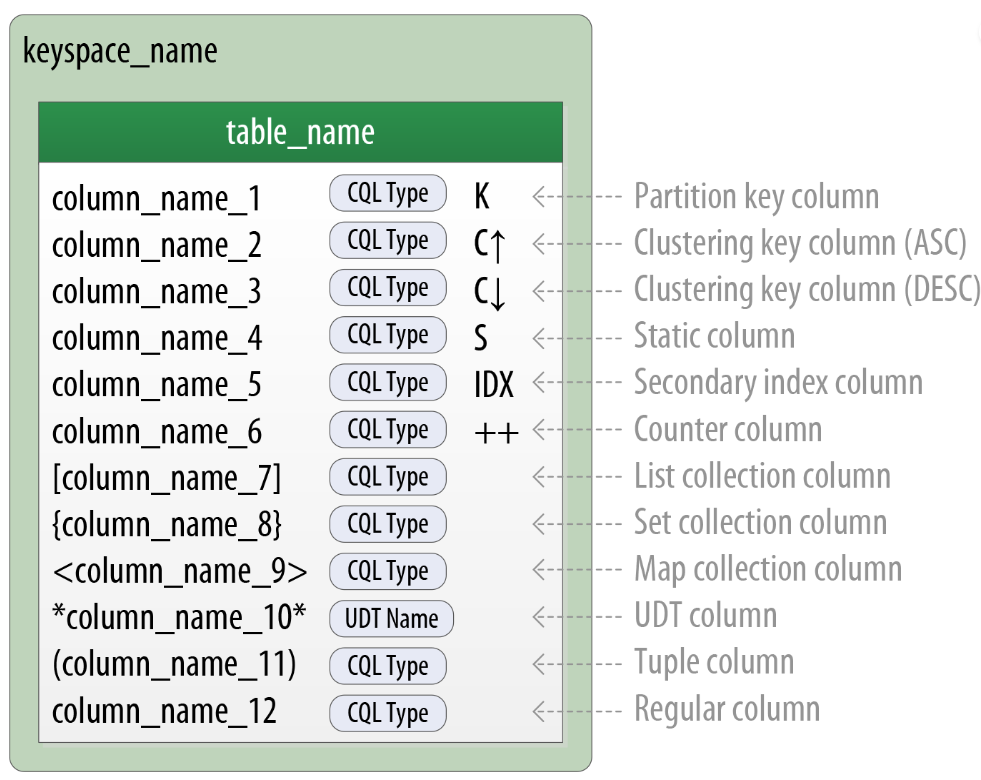
**Evaluating and Refining Data Models**

Evaluate and refine table designs to help ensure optimal performance.

**Calculating Partition Size**

Partition size is measured by the number of cells (values) that are stored in the partition.

In order to calculate the size of partitions, use the following formula:

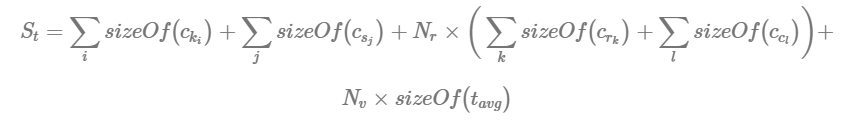


Nv=Nr(Nc−Npk−Ns)+Ns

The number of values (or cells) in the partition (Nv) is equal to the number of static columns (Ns) plus the product of the number of rows (Nr) and the number of of values per row. The number of values per row is defined as the number of columns (Nc) minus the number of primary key columns (Npk) and static columns (Ns).

**Calculating Size on Disk**

In addition to calculating the size of a partition, it is also an excellent idea to estimate the amount of disk space that will be required for each table you plan to store in the cluster. In order to determine the size, use the following formula to determine the size St of a partition:



This is a bit more complex than the previous formula, but let’s break it down a bit at a time. Let’s take a look at the notation first:

* In this formula, ck refers to partition key columns, cs to static columns, cr to regular columns, and cc to clustering columns.
* The term tavg refers to the average number of bytes of metadata stored per cell, such as timestamps. It is typical to use an estimate of 8 bytes for this value.
* You’ll recognize the number of rows Nr and number of values Nv from previous calculations.
* The **sizeOf()** function refers to the size in bytes of the CQL data type of each referenced column.

The first term asks you to sum the size of the partition key columns.

The second term asks you to sum the size of the static columns. This table has no static columns, so the size is 0 bytes.

The third term is the most involved, and for good reason—it is calculating the size of the cells in the partition. Sum the size of the clustering columns and regular columns. The two clustering columns are the date, which is 4 bytes, and the room\_number, which is a 2-byte short integer, giving a sum of 6 bytes. There is only a single regular column, the boolean is\_available, which is 1 byte in size. o finish up the term, multiply this value by the number of rows (73,000), giving a result of 511,000 bytes (0.51 MB).

The fourth term is simply counting the metadata that that Cassandra stores for each cell. In the storage format used by Cassandra 3.0 and later, the amount of metadata for a given cell varies based on the type of data being stored, and whether or not custom timestamp or TTL values are specified for individual cells. For this table, reuse the number of values from the previous calculation (73,000) and multiply by 8, which gives 0.58 MB. Adding these terms together, you get a final estimate:



This formula is an approximation of the actual size of a partition on disk, but is accurate enough to be quite useful. Remembering that the partition must be able to fit on a single node, it looks like the table design will not put a lot of strain on disk storage.

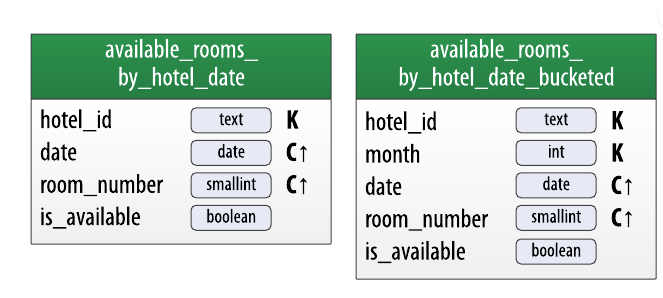
**Breaking up Large Partitions**

The goal is to design tables that can provide the data you need with queries that touch a single partition, or failing that, the minimum possible number of partitions. However, as shown in the examples, it is quite possible to design wide partition-style tables that approach Cassandra’s built-in limits. Performing sizing analysis on tables may reveal partitions that are potentially too large, either in number of values, size on disk, or both.

The technique for splitting a large partition is straightforward: add an additional column to the partition key. In most cases, moving one of the existing columns into the partition key will be sufficient. Another option is to introduce an additional column to the table to act as a sharding key, but this requires additional application logic.

Continuing to examine the available rooms example, if you add the date column to the partition key for the available\_rooms\_by\_hotel\_date table, each partition would then represent the availability of rooms at a specific hotel on a specific date. This will certainly yield partitions that are significantly smaller, perhaps too small, as the data for consecutive days will likely be on separate nodes.

Another technique known as **bucketing** is often used to break the data into moderate-size partitions. For example, you could bucketize the available\_rooms\_by\_hotel\_date table by adding a month column to the partition key, perhaps represented as an integer. The comparision with the original design is shown in the figure below. While the month column is partially duplicative of the date, it provides a nice way of grouping related data in a partition that will not get too large.



**Cassandra Data Modeling Tools**

There are several tools available to help you design and manage your Cassandra schema and build queries.

* **Hackolade** is a data modeling tool that supports schema design for Cassandra and many other NoSQL databases. Hackolade supports the unique concepts of CQL such as partition keys and clustering columns, as well as data types including collections and UDTs. It also provides the ability to create Chebotko diagrams.
* **Kashlev Data Modeler** is a Cassandra data modeling tool that automates the data modeling methodology described in this documentation, including identifying access patterns, conceptual, logical, and physical data modeling, and schema generation. It also includes model patterns that you can optionally leverage as a starting point for your designs.
* DataStax DevCenter is a tool for managing schema, executing queries and viewing results. While the tool is no longer actively supported. DevCenter features syntax highlighting for CQL commands, types, and name literals. DevCenter provides command completion as you type out CQL commands and interprets the commands you type, highlighting any errors you make. The tool provides panes for managing multiple CQL scripts and connections to multiple clusters. The connections are used to run CQL commands against live clusters and view the results. The tool also has a query trace feature that is useful for gaining insight into the performance of your queries.
* IDE Plugins - There are CQL plugins available for several Integrated Development Environments (IDEs), such as IntelliJ IDEA and Apache NetBeans. These plugins typically provide features such as schema management and query execution.

Some IDEs and tools that claim to support Cassandra do not actually support CQL natively, but instead access Cassandra using a JDBC/ODBC driver and interact with Cassandra as if it were a relational database with SQL support. When selecting tools for working with Cassandra make sure they support CQL and reinforce Cassandra best practices for data modeling.

**CASSANDRA EXAMPLES**

Refer the tutorial Apache Cassandra

[**https://cassandra.apache.org/doc/latest/cassandra/data\_modeling/intro.html**](https://cassandra.apache.org/doc/latest/cassandra/data_modeling/intro.html)

**CASSANDRA CLIENTS**

[**https://cassandra.apache.org/doc/latest/cassandra/getting\_started/drivers.html**](https://cassandra.apache.org/doc/latest/cassandra/getting_started/drivers.html)