SEPTEMBER 16, 2019

Comparing Database Types: How Database Types Evolved to Meet Different Needs

Justin Ellingwood
@jmellingwood

Join the discussion 10

Many types of databases exist, each with their own benefits. In this guide, we'll compare the relational, document, key-value, graph, and wide-column databases and talk about what each of them offer.

Database types, sometimes referred to as database models or database families, are the patterns and structures used to organize data within a database management system. Many different database types have been developed over the years. Some are mainly historic predecessors to current databases, while others have stood the test of time. In the last few decades, new types have been developed to address changing requirements and different use patterns.

Your choice of database type can have a profound impact on what kind of operations your application can easily perform, how you conceptualize your data, and the features that your database management system offers you during development and runtime. In this guide, we'll take a look at how database types have evolved over time and what advantages and trade-offs are present in each design.

Legacy databases: paving the way for modern systems

Legacy database types represent milestones on the path to modern databases. These may still find a foothold in certain specialized environments, but have mostly been replaced by more robust alternatives for production environments.

This section is dedicated to historic database types that aren't used much in modern development. You can skip ahead to the section on relational databases if you aren't interested in that background.

Flat-file databases: simple data structures for organizing small amounts of local data

The simplest way to manage data on a computer outside of an application is to store it in a basic file format. The first solutions for data management used this approach and it is still a popular option for storing small amounts of information without heavy requirements.

The first flat file databases represented information in regular, machine parse-able structures within files. Data is stored in plain text, which limits the type of content that can be represented within the database itself. A special character or other indicator is chosen to use as a *delimiter*, or marker for when one field ends and the next begins. For example, a comma is used in CSV (comma-separated values) files, while colons or white-space are used in many data files in Unix-like systems.

`/etc/passwd` on *nix systems:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
```

```
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
bob:x:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

The `/etc/passwd` file defines users, one per line. Each user has attributes like name, user and group IDs, home directory and default shell, each separated by a colon.

While flat file databases are simple, they are very limited in the level of complexity they can handle. The system that reads or manipulates the data cannot make easy connections between the data represented. File-based systems usually don't have any type of user or data concurrency features either. Flat file databases are usually only practical for systems with small read or write requirements. For example, many operating systems use flat-files to store configuration data.

In spite of these limitations, flat-file databases are still widely used for scenarios where local processes need to store and organized small amounts of data. A good example of this is for configuration data for many applications on Linux and other Unix-like systems. In these cases, the flat-file format serves as an interface that both humans and applications can easily read and manage. Some advantages of this format are that it has robust, flexible tooling, is easily managed without specialized software, and is easy to understand and work with.

Examples:

- `/etc/passwd` and `/etc/fstab` on Linux and Unix-like systems
- CSV files

Hierarchical databases: using parent-child relationships to map data into trees

Initial introduction: 1960s

Hierarchical databases were the next evolution in database management development. They encode a relationship between items where every record has a single parent. This builds a tree-like structure that can be used to categorize records according to their

parent record.

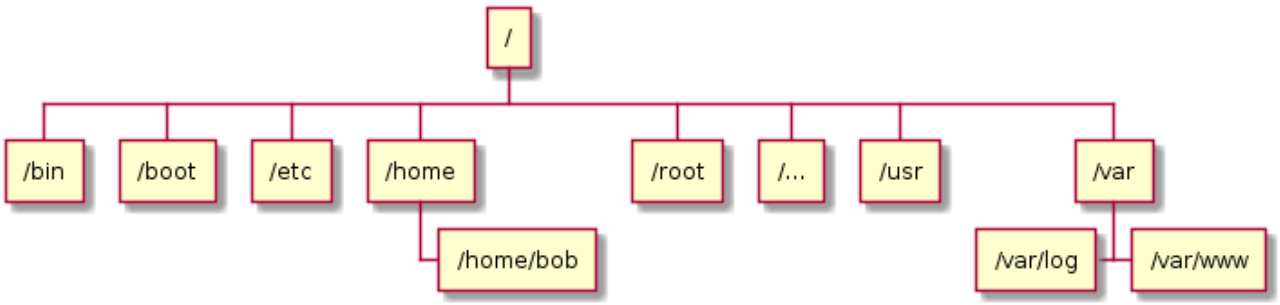

Diagram of a hierarchical database

This simple relationship mapping provides users with the ability to establish relationships between items in a tree structure. This is very useful for certain types of data, but does not allow for complex relationship management. Furthermore, the meaning of the parent-child relationship is implicit. One parent-child connection could be between a customer and their orders, while another might represent an employee and the equipment they have been allocated. The structure of the data itself does not distinguish between these relationships.

Hierarchical databases are the beginning of a movement towards thinking about data management in more complex terms. The trajectory of database management systems that were developed afterwards continues this trend.

Hierarchical databases are not used much today due to their limited ability to organize most data and because of the overhead of accessing data by traversing the hierarchy. However, a few incredibly important systems could be considered hierarchical databases. A filesystem, for instance, can be thought of as a specialized hierarchical database, as the system of files and directories fit neatly into the single-parent / multiple-child paradigm. Likewise, DNS and LDAP systems both act as databases for hierarchical datasets.

Examples:

- Filesystems
- DNS

- LDAP directories

Network databases: mapping more flexible connections with non-hierarchical links

Initial introduction: late 1960s

Network databases built on the foundation provided by hierarchical databases by adding additional flexibility. Instead of always having a single parent, as in hierarchical databases, network database entries can have more than one parent, which effectively allows them to model more complex relationships. When talking about network databases, it is important to realize that *network* is being used to refer to connections between different data entries, not connections between different computers or software.

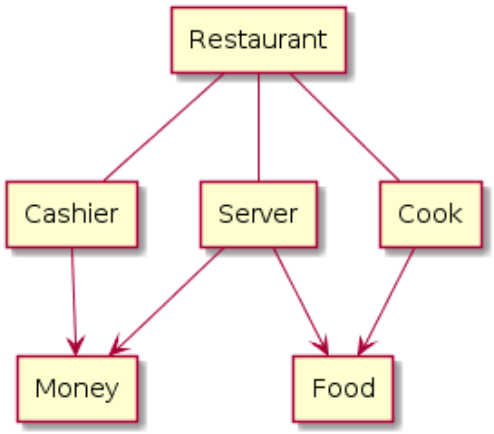

Diagram of a network database

Network databases can be represented by a generic *graph* instead of a *tree*. The meaning of the graph was defined by a **schema**, which lays out what each data node and each relationship represents. This gave structure to the data in a way that could previously only be reached through inference.

> ***Definition: Schema***
>
> *A database schema is a description of the logical structure of a database or the elements it contains. Schemas often include declarations for the structure of individual entries, groups of entries, and the individual attributes that database entries are comprised of.*

> *These may also define data types and additional constraints to control the type of data that may be added to the structure.*

Network databases were a huge leap forward in terms of flexibility and the ability to map connections between information. However, they were still limited by the same access patterns and design mindset of hierarchical databases. For instance, to access data, you still needed to follow the network paths to the record in question. The parent-child relationship carried over from hierarchical databases also affected the way that items could connect to one another.

It is difficult to find modern examples of network database systems. Setting up and working with network databases required a good deal of skill and specialized domain knowledge. Most systems that could be approximated using network databases found a better fit once relational databases appeared.

Examples:

- IDMS

Relational databases: working with tables as a standard solution to organize well-structured data

Initial introduction: 1969

Relational databases are the oldest general purpose database type still widely used today. In fact, relational databases comprise the majority of databases currently used in production.

Relational databases organize data using *tables*. Tables are structures that impose a schema on the records that they hold. Each column within a table has a *name* and a *data type*. Each row represents an individual record or data item within the table, which contains values for each of the columns. Relational databases get their name from the fact that relationships can be defined between tables.

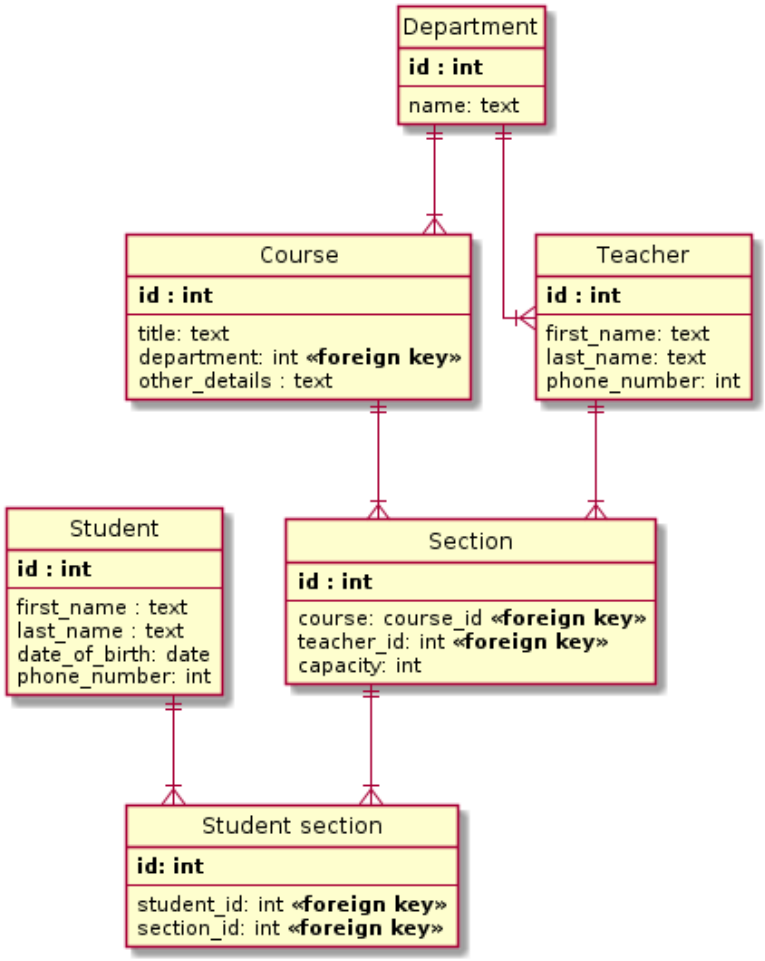

Diagram of relational schema used to map entities for a school

Special fields in tables, called *foreign keys*, can contain references to columns in other tables. This allows the database to bridge the two tables on demand to bring different types of data together.

The highly organized structure imparted by the rigid table structure, combined with the flexibility offered by the relations between tables makes relational databases very powerful and adaptable to many types of data. Conformity can be enforced at the table level, but database operations can combine and manipulate that data in novel ways.

While not inherent to the design of relational databases, a querying language called SQL, or structured query language, was created to access and manipulate data stored with that format. It can query and join data from multiple tables within a single statement. SQL can also filter, aggregate, summarize, and limit the data that it returns. So while SQL is not a part of the relational system, it is often a fundamental part of working with these databases.

> ***Definition: SQL***
>
> *SQL, or structured querying language, is a language family used to query and manipulate data within relational databases. It excels at combining data from multiple tables and filtering based on constraints which allow it to be used to express complex queries. Variants of the language has been adopted by almost all relational databases due to its flexibility, power, and ubiquity.*

In general, relational databases are often a good fit for any data that is regular, predictable, and benefits from the ability to flexibly compose information in various formats. Because relational databases work off of a schema, it can be more challenging to alter the structure of data after it is in the system. However, the schema also helps enforce the integrity of the data, making sure values match the expected formats, and that required information is included. Overall, relational databases are a solid choice for many applications because applications often generate well-ordered, structured data.

Examples:

- MySQL
- MariaDB
- PostgreSQL
- SQLite

NoSQL databases: modern alternatives for data that doesn't fit the relational paradigm

NoSQL is a term for a varied collection of modern database types that offer approaches that differ from the standard relational pattern. The term NoSQL is somewhat of a misnomer since the databases within this category are more of a reaction against the relational archetype rather than the SQL querying language.

Key-value databases: simple, dictionary-style lookups for basic storage and retrieval

Initial introduction: 1970s | Rise in popularity: 2000-2010

Key-value databases, or key-value stores, are one of the simplest database types. Key-value stores work by storing arbitrary data accessible through a specific *key*. To store data, you provide a key and the blob of data you wish to save, for example a JSON object, an image, or plain text. To retrieve data, you provide the key and will then be given the blob of data back. The database does not evaluate the data it is storing and allows limited ways of interacting with it.

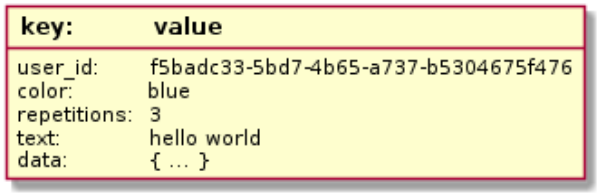

Diagram of key-value data store

If key-value stores appear simple, it's because they are. But that simplicity is often an asset in the kinds of scenarios where they are most often deployed. Key-value stores are often used to store configuration data, state information, and any data that might be represented by a *dictionary* or *hash* in a programming language. Key-value stores provide fast, low-complexity access to this type of data.

Key-value databases don't prescribe any schema for the data they store, and as such, are often used to store many different types of data at the same time. The user is responsible for defining any naming scheme for the keys that will help identify the values and are responsible for ensuring the value is of the appropriate type and format. Key-value storage is most useful as a lightweight solution for storing simple values that can be operated on externally after retrieval.

One of the most popular uses for key-value databases are to store configuration values and application variables and flags for websites and web applications. Programs can check the key-value store, which is usually very fast, for their configuration when they start. This allows you to alter the runtime behavior of your services by changing the data in the key-value store. Applications can also be configured to recheck periodically or to restart when they see changes. These configuration stores are often persisted to disk periodically to prevent loss of data in the event of a system crash.

Examples:

- Redis
- memcached
- etcd

Document databases: Storing all of an item's data in flexible, self-describing structures

Rise in popularity: 2009

Document databases, also known as document-oriented databases or document stores, share the basic access and retrieval semantics of key-value stores. Document databases also use a key to uniquely identify data within the database. In fact, the line between advanced key-value stores and document databases can be fairly unclear. However, instead of storing arbitrary blobs of data, document databases store data in structured formats called documents, often using formats like JSON, BSON, or XML.

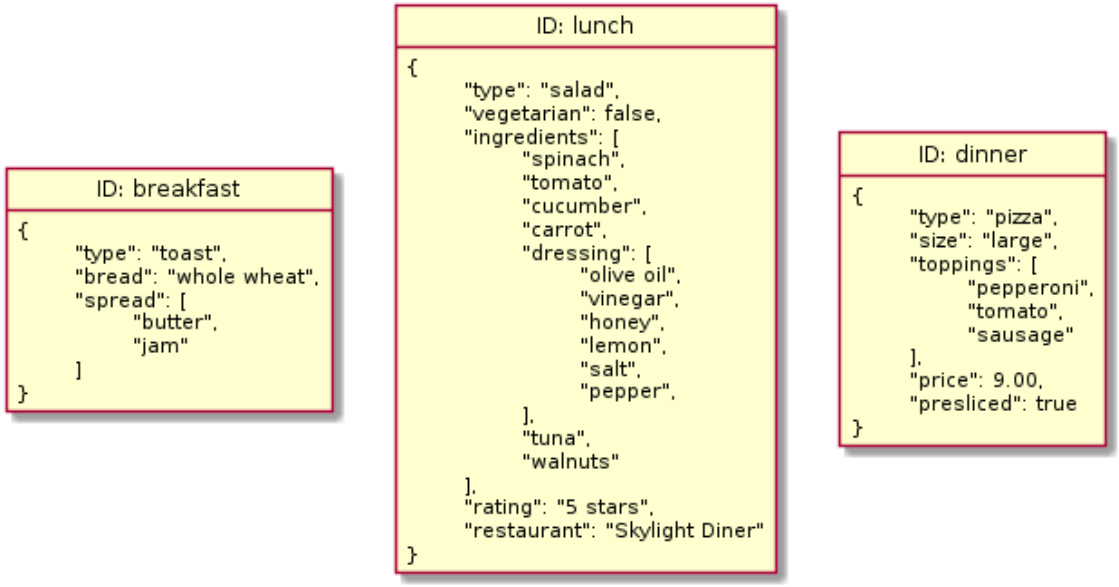

Diagram of document database

Though the data within documents is organized within a structure, document databases do not prescribe any specific format or schema. Each document can have a different internal structure that the database interprets. So, unlike with key-value stores, the

content stored in document databases can be queried and analyzed.

In some ways, document databases sit in between relational databases and key-value stores. They use the simple key-value semantics and loose requirements on data that key-value stores are known for, but they also provide the ability to impose a structure that you can use to query and operate on the data in the future.

The comparison with relational databases shouldn't be overstated, however. While document databases provide methods of structuring data within documents and operating on datasets based on those structures, the guarantees, relationships, and operations available are very different from relational databases.

Document databases are a good choice for rapid development because you can change the properties of the data you want to save at any point without altering existing structures or data. You only need to backfill records if you want to. Each document within the database stands on its own with its own system of organization. If you're still figuring out your data structure and your data is mainly composed discrete entries that don't include a lot of cross references, a document database might be a good place to start. Be careful, however, as the extra flexibility means that you are responsible for maintaining the consistency and structure of your data, which can be extremely challenging.

Examples:

- MongoDB
- RethinkDB
- Couchbase

Graph databases: mapping relationships by focusing on how connections between data are meaningful

Rise in popularity: 2000s

Graph databases are a type of NoSQL database that takes a different approach to establishing relationships between data. Rather than mapping relationships with tables

and foreign keys, graph databases establish connections using the concepts of *nodes*, *edges*, and *properties*.

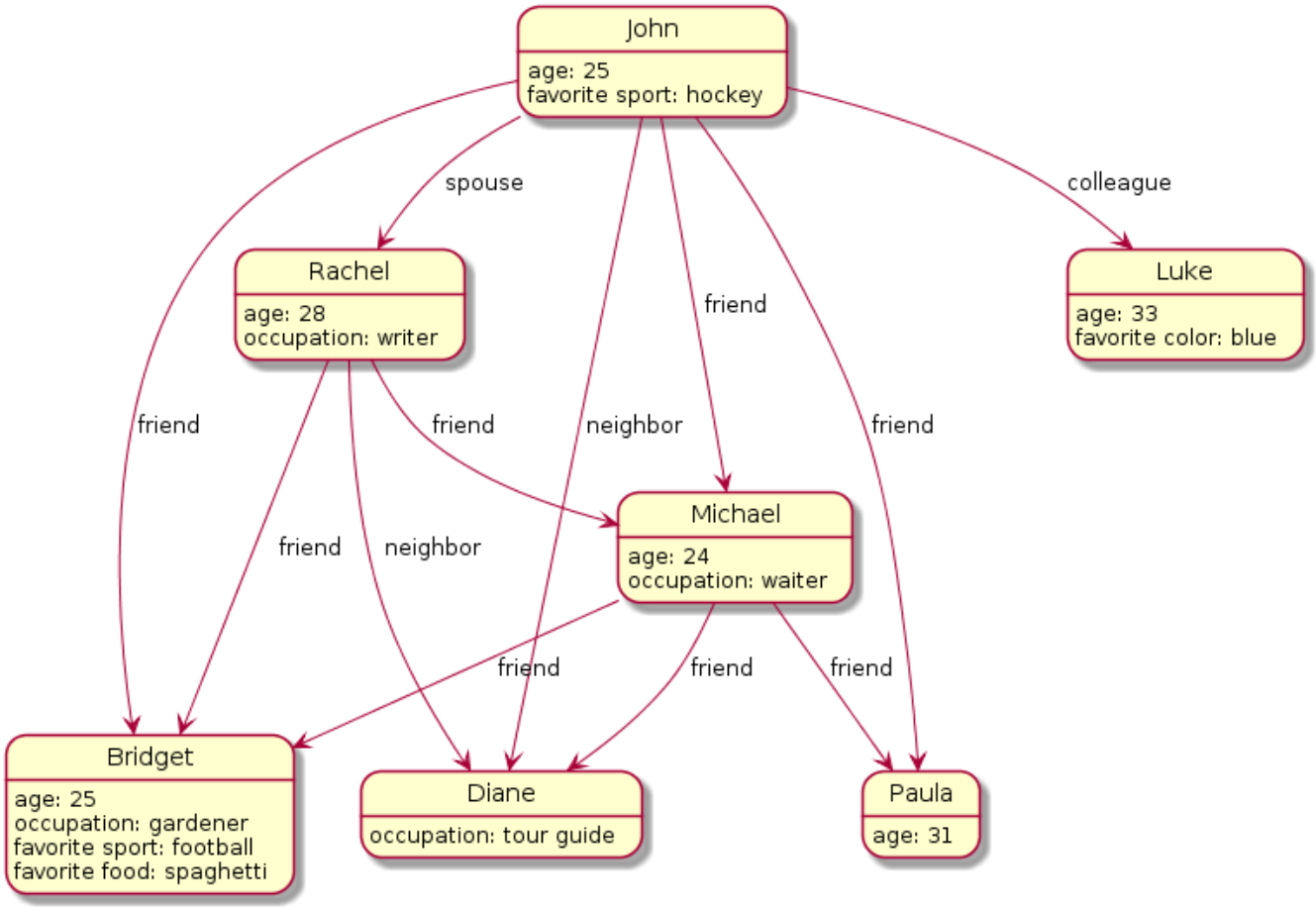

Diagram of a graph database structure

Graph databases represents data as individual nodes which can have any number of properties associated with them. Between these nodes, edges (also called relationships) are established to represent different types of connections. In this way, the database encodes information about the data items within the nodes and information about their relationship in the edges that connect the nodes.

At a glance, graph databases appear similar to earlier network databases. Both types focus on the connections between items and allow for explicit mapping of relationships between different types of data. However, network databases require step-by-step traversal to travel between items and are limited in the types of relationships they can represent.

Graph databases are most useful when working with data where the relationships or

connections are highly important. It is essential to understand that when talking about relational databases, the word "relational" refers to the ability to tie information in different tables together. On the other hand, with graph databases, the primary purpose is defining and managing relationships themselves.

For example, querying for the connection between two users of a social media site in a relational database is likely to require multiple table joins and therefore be rather resource intensive. This same query would be straightforward in a graph database that directly maps connections. The focus of graph databases is to make working this type of data intuitive and powerful.

- Neo4j
- Titan

Column-family databases: databases with flexible columns to bridge the gap between relational and document databases

Rise in popularity: 2000s

Column-family databases, also called non-relational column stores, wide-column databases, or simply column databases, are perhaps the NoSQL type that, on the surface, looks most similar to relational databases. Like relational databases, wide-column databases store data using concepts like rows and columns. However, in wide-column databases, the association between these elements is very different from how relational databases use them.

In relational databases, a schema defines the column layout in a table by specifying what columns the table will have, their respective data types, and other criteria. All of the rows in a table must conform to this fixed schema.

Instead of tables, column-family databases have structures called *column families*. Column families contain rows of data, each of which define their own format. A row is composed of a unique row identifier — used to locate the row — followed by sets of column names and values.

With this design, each row in a column family defines its own schema. That schema can be easily modified because it only affects that single row of data. Each row can have different numbers of columns with different types of data. Sometimes it helps to think of column family databases as key-value databases where each key (row identifier) returns a dictionary of arbitrary attributes and their values (the column names and their values).

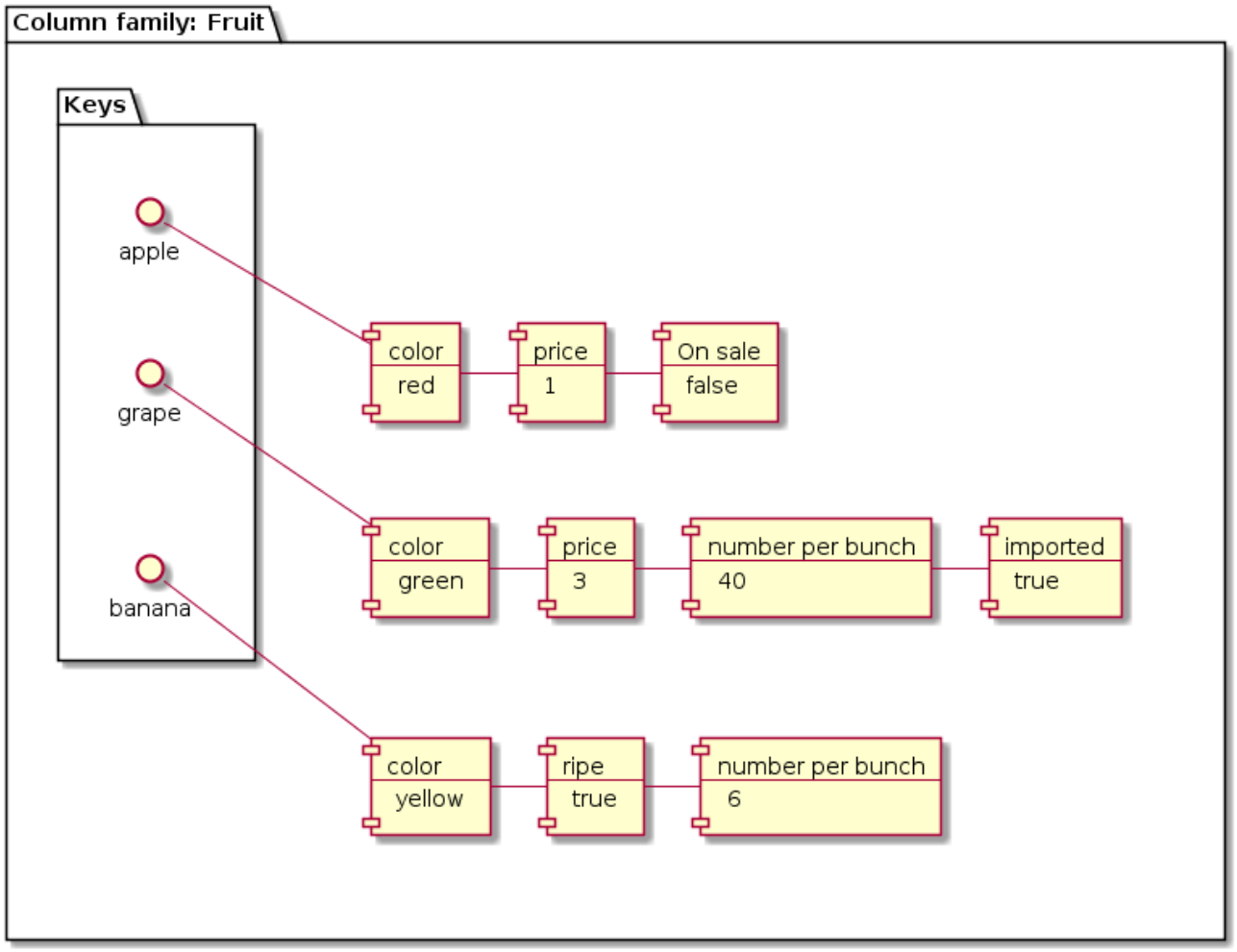

Diagram of column-family database structure

Column-family databases are good when working with applications that requires great performance for row-based operations and highly scalability. Since all of the data and metadata for an entry is accessible with a single row identifier, no computationally expensive joins are required to find and pull the information. The database system also typically makes sure all of the data in a row is collocated on the same machine in a cluster, simplifying data sharding and scaling.

However, column-family databases do not work well in all scenarios. If you have highly relational data that requires joins, this is not the right type of database for your application. Column-family databases are firmly oriented around row-based operations. This means that aggregate queries like summing, averaging, and other analytics-oriented processes can be difficult or impossible. This can have a great impact on how you design your applications and what types of usage patterns you can use.

Examples:

- Cassandra
- HBase

NewSQL databases: bringing modern scalability and performance to the traditional relational pattern

Rise in popularity: 2010s

NoSQL databases are great options for situations where your data does not fit neatly into the relational pattern. Since they were developed more recently, NoSQL systems tend to be designed with scalability and modern performance requirements in mind.

However, until recently, no solution existed to easily *scale* relational data. To address this need, a new type of relational databases called NewSQL databases were developed.

NewSQL databases follow the relational structure and semantics, but are built using more modern, scalable designs. The goal is to offer greater scalability than relational databases and greater *consistency guarantees* than NoSQL alternatives. They achieve this by sacrificing certain amounts of availability in the event of a networking partition. The trade offs between consistency and availability is a fundamental problem of distributed databases described by the *CAP theorem*.

> ***Definition: CAP Theorem***
>
> *The CAP theorem is a statement about the trade offs that distributed databases must*

> *make between availability and consistency. It asserts that in the event of a network partition, a distributed database can choose either to remain available or remain consistent, but it cannot do both. Cluster members in a partitioned network can continue operating, leading to at least temporary inconsistency. Alternatively, at least some of the disconnected members must refuse to alter their data during the partition to ensure data consistency.*

To address the availability concern, new architectures were developed to minimize the impact of partitions. For instance, splitting data sets into smaller ranges called *shards* can minimize the amount of data that is unavailable during partitions. Furthermore, mechanisms to automatically alter the roles of various cluster members based on network conditions allow them to regain availability quickly.

Because of these qualities, NewSQL databases are best suited for use cases with high volumes of relational data in distributed, cloud-like environments.

While NewSQL databases offer most of the familiar features of conventional relational databases, there are some important differences that prevent it from being a one-to-one replacement. NewSQL systems are typically less flexible and generalized than their more conventional relational counterparts. They also usually only offer a subset of full SQL and relational features, which means that they might not be able to handle certain kinds of usage. Many NewSQL implementations also store a large part of or their entire dataset in the computer's main memory. This improves performance at the cost of greater risk to unpersisted changes.

NewSQL databases are a good fit for relational datasets that require scaling beyond what conventional relational databases can offer. Because they implement the relational abstraction and provide SQL interfaces, transitioning to a NewSQL database is often more straightforward than moving to a NoSQL alternative. However, it's important to keep in mind that although they mostly seek to replicate the conventional relational environments, there are differences that may affect your deployments. Be sure to research these differences and identify situations where the resemblance breaks down.

Examples:

- MemSQL
- VoltDB
- Spanner
- Calvin
- CockroachDB
- FaunaDB
- yugabyteDB

Conclusion

Database types have changed a lot since their initial introduction and new database ideas are actively being developed today. Each of the types used in modern systems have distinct advantages that are worth exploring given the right access patterns, data properties, and requirements. One of the first and most important decisions when starting a new project is evaluating your needs and finding the type that matches your project's demands.

Many times, using a mixture of different database types is the best approach for handling the data of your projects. Your applications and services will influence the type of data being generated as well as the features and access patterns you require. For example, user information for your system might fit best in a relational database, while the configuration values for your services might benefit from an in-memory key-value store. Learning what each type of database offers can help you recognize which systems are best for all of your different types of data.