How to Choose the Right Database System: RDBMS vs. NoSql vs. NewSQL

ALEX CASTROUNIS

Introduction

Choosing the right database system is one of, if not the most important decisions to make when designing a new data-driven software solution, and as we will see, is based on many factors and tradeoffs.

Proper database technology selection is required for data-centric solutions such as SaaS platforms, big data pipelines, analytics platforms, machine learning/artificial intelligence platforms, and so on.

As we'll soon discuss, choosing a database system involves expertise and careful consideration of a vast amount of highly technical details and factors, any of which can warrant an article of its own. Given that, this article is not meant to be a complete or rigorous treatment of the subject, but rather an overview to introduce the reader to key concepts and guide further research.

It's also worth mentioning that many of the following statements about the capabilities and functionality of different database system types (e.g., RDBMS, NoSQL, NewSQL) are generalized. In reality, much of this is highly vendor-specific and can therefore vary significantly.

Database Considerations and Requirements

The first and most obvious purpose of a database is to store, update, and access data. All database systems allow these operations in one form or another.

Other functional and nonfunctional system considerations and requirements include:

- Consistency, availability, and partition tolerance (CAP)
- Robustness and reliability
- Scalability
    - Performant and highly available functioning regardless of concurrent demands on the system
- Performance and speed
- Partitioning ability
    - Distributed data partitions of a complete database across multiple separate nodes in order to spread load and increase performance
    - Horizontal (sharding) and/or vertical partitioning
- Distributability
- In-database analytics and monitoring

- Operational and querying capabilities
- Storage management
- Talent pool and availability of relevant skills
- Database integrity and constraints
- Data model flexibility
- Database security
- Database vendor/system funding, stability, community, and level of establishment
    - Note that many commercial relational database systems (RDBMS) have been developed and widely used at scale for a very long time, and thus offer a significant level of robustness, functionality, reliability, and so on.
- And many more

One must also carefully consider the type and structure of data being stored, and the ideal method of modeling the data.

Data Structure, Models, and Schemas

Data is usually referred to as being structured, semi-structured, or unstructured. These distinctions are important because they're directly related to the type of

database technologies and storage needed, the software and methods by which the data is queried and processed, and the complexity of dealing with the data.

Structured data refers to data that is usually represented by a logical data model that is defined by a schema in a database. Often it's easily queryable using SQL (structured query language) since the structure of the data is known. A sales order record is a good example, with order number and order date being a few of the potential data fields.

Unstructured data, roughly speaking, is data that's not defined by any schema (schemaless) or model, and is not organized in a specific way. In reality, unstructured data can sometimes have minimal structure, but the structured data is non-relational. A good example of unstructured data is text that is found in artifacts such as files, documents, and the body of an email.

It follows naturally that semi-structured data is a combination of the two. It's basically unstructured data that also has structured data (metadata) appended to it. Every time you use your smartphone to take a picture, the shutter captures light

Every time you use your smartphone to take a picture, the shutter captures light reflection information as a bunch of binary data (unstructured), but the camera also appends additional data (the structured part) that includes the date and time the photo was taken, last time it was modified, image size, etc.

Data, often called domain or business data in the structured case, can be logically modeled in many different ways, and is usually represented as objects in object-oriented software applications that create, read, update, and delete (CRUD) the data.

Data models differ from logical data schemas in that schemas represent the way that structured data is represented and stored in a database. This includes the data itself, along with any relations, data types and formats, indexes, and constraints such as uniqueness, keys, and data integrity.

Data schemas basically enforce the structure and integrity of the data in the system, whereas data models are logical, abstract representations of data. A structured data model object in application code, for example, may be logically organized and constrained in a database by a schema that includes multiple tables that may require joins in order to retrieve a single data model object or record.

Distributed Systems

Modern software and data applications often require significant scalability to handle high request volumes and data throughput. In addition, globalization and performance expectations have led to the global distribution of data and other network-based resources for quicker access depending on location.

In order to meet these scalability, global availability, and speed requirements, many software and data systems are now distributed across multiple computers in the same or different locations.

These database systems are known as a distributed database system (DDBS) or distributed database management system (DDBMS). Popular techniques such as replication and duplication are used in order to keep data consistent and current across all distributed locations.

Replication describes the situation where any changes made to one location in the distributed database are identified and subsequently made to all locations in the database. This process is managed by the system and can be very time and computing resource intensive. Duplication on the other hand describes the case where there is a master database, which is then copied to other locations so that there are multiple copies of the database.

Prior to the development and widespread usage of NoSQL systems and/or robust distribution functionality, the primary way to scale a database system was to vertically scale, or scale up to a much more expensive and powerful single server.

Determining scalability requirements is often handled by devops folks and/or site reliability engineers (SRE), a discussion of which is out of scope here.

Database Transactions, ACID, and BASE

A transaction describes a sequence of operations performed by a transaction-processing system (e.g., database) as a single logical unit of work. Normally these systems employ features such as rollbacks, rollforwards, deadlocks, and others, although these will not be described here.

ACID is an acronym to describe database transaction properties, and includes atomicity, consistency, isolation, and durability.

Atomicity indicates that a transaction is all or nothing, which means that any failure in a transaction causes the entire transaction to fail and therefore leaves the database unchanged.

Consistency ensures that all transactions result in a valid state of the database, and that all validation rules and constraints are met, and required actions are carried out.

Isolation means that the database is able to perform concurrent transactions that result in the same database state as if the transactions had occurred one after the other. This is a form of concurrency control, which can be implemented in different ways, with serializability referring to the highest level of isolation.

Durability ensures that a committed transaction persists, or is permanently stored under any and all conditions.

The ACID model for transactions strongly favors consistency and is not without criticism, which has led to an alternate and more partition tolerant, availability-focused, and highly scalable transactional model known as BASE.

BASE is an acronym used to describe eventually consistent services, and stands for basically available, soft state, and eventual consistency.
Basically available means that any data request should receive a response, but that response may indicate a failure or changing state as opposed to the requested data.

Soft state indicates that given eventual consistency, the system may be in a changing state until consistency is reached.

Eventual consistency describes the situation where a DDBS achieves high availability while loosely guaranteeing that data, in the absence of updates, will eventually reflect the last updated value across the system, and therefore the data may vary in value until the system reaches a consistent state. This combined with the fact that the system will continue to serve requests without checking the consistency of previous transactions.

Eventual consistency is considered a optimistic replication model, as opposed to ACID, which is considered a pessimistic replication model.

These concepts and potential tradeoffs are important to consider when selecting database technologies, as each address and prioritize requirements in different ways.

CAP and PACELC

The CAP theorem (or Brewer's theorem) says that it's not possible for a distributed computer system to simultaneously provide consistency, availability, and partition tolerance guarantees. A great reference, and basis for this discussion on CAP and a newer formulation that we'll discuss is Consistency Tradeoffs in Modern Distributed Database System Design by Daniel J. Abadi of Yale University.

Consistency indicates that all successful database operations or transactions result in a valid state, and that any data requested is up to date or an error is returned. The concept can be described by many different types of consistency models that are out of scope here.

Availability says that every request receives a response, without guarantee that it contains the most recent version of the information. It is implied that availability here refers to database nodes that are in a non-failing state, and that return requested information as opposed to an error.
Partition tolerance ensures that the system continues to operate despite arbitrary partitioning due to network failures

A partition in this case refers to a network partition in which one component or subsystem of a network is no longer able to communicate with another. In the case of a DDBS, one or more distributed nodes can no longer communicate to others in the presence of a network partition.

Importantly, the CAP theorem is only relevant in the existence of a network partition and certain failures, but not during normal DDBS operation.

Given DDBS systems where the CAP theorem is applicable, one can list three possible combinations of guarantees that a DDBS can offer:

- AP: Highly available and partition tolerant, but not consistent
- CP: Consistent and partition tolerant, but not highly available
- CA: Highly available and consistent, but not partition tolerant

To illustrate the tradeoffs involved, imagine a DDBS, and a network partition occurs leaving one node unable to communicate with the others. In this case, it has two options.

Either stop serving requests which ensures consistency, but violates availability, or continue to respond to requests with potentially stale data and/or result in write conflicts since the same data may have been updated differently via multiple nodes. This latter case violates consistency, but preserves availability.

Given this, and since partition tolerance is a more or less mandatory requirement for distributed systems, typically one must prioritize between consistency and availability, and thus sacrifice one for the other. This is known as the availability/consistency tradeoff.

However, Abadi points out that network partitions are relatively rare and less frequent than other DDBS failure types, and that another very important tradeoff must be considered. It is the one between consistency and latency, and has resulted in what’s known as the PACELC formulation.

The consistency/latency tradeoff is relevant even in the absence of a network partition, and is particularly important with web-based software applications. Many studies have shown that relatively minor increases in web page latency when serving to a browser or during application usage can result in significant loss of users and potential revenue.

Since latency and availability are somewhat related, the tradeoff is more focused on latency and consistency. Latency and availability are related since very high latency is essentially the same as being unavailable, while web-based latency within normal acceptable limits is essentially available.

The tradeoff is the result of high availability requirements that cause the need for replicated data. Just the possibility of a node in the system failing requires some degree of data replication, with the highest possible level of availability coming from data replication over a wide area network (WAN) to protect against localized disasters for example.

The consistency/latency tradeoff arises from this data replication requirement, and

The consistency/latency tradeoff arises from this data replication requirement, and the fact that there are three possible methods of replicating the data. The three possibilities are:

- Data is sent to all replicas simultaneously
- Data is sent to a preconfigured master node first
- Data is sent to a single arbitrary node first

Regardless of the replication method employed, there will be a tradeoff between consistency and latency as a result. Please refer to Abadi's article for a detailed discussion of this tradeoff that results from systems that replicate data. In the absence of data replication, node failures or overload causes availability issues that can be considered extreme latency.

It's important to note that the PACELC formulation described the two most important trade offs related to distributed systems. The availability/consistency trade off in the presence of a network partition, and the consistency/latency tradeoff under normal operating conditions of replicating systems.

These potential consistency/latency tradeoffs can be used to choose whether or not to replicate data.

Relational Database Management Systems (RDBMS)

Relational database management systems (RDBMS) are very well suited for storing and querying structured relational data, although some support storing unstructured data and multiple storage types as well. They are typically characterized as CA systems that therefore sacrifice partition tolerance, and are often implemented as a single server, which requires very expensive and reliable computer hardware to scale.

Relational data means that data stored in different parts (i.e., tables) of the database are often related to one another, and usually in a one to many, or many to many relationship. Each table (or relation) consists of rows (records) and columns (fields or attributes), with a unique identifier (key) per row.

As mentioned earlier, data is usually represented in software applications as domain objects. These data objects (or entities) aren’t represented or shaped in the same way in software as they are in a relational database, although a common pattern is to represent a single domain object (or entity) as a row entry in a single entity-specific table (e.g., active record).

Data often needs to be stored and accessed across multiple tables due its relational nature, and this creates a mismatch between applications and database data representations that is usually referred to as the object-relational impedance mismatch.

This mismatch occurs more specifically due to the need to map software objects to database tables created from relational schemas. In order to address this mismatch, various architectural patterns and software applications have been created to map software objects to database tables and vice versa. This is known as object-relational mapping (ORM).

RDBMS are generally very focused on providing strong consistency and ACID transactions (although not always), tremendous functionality, stability, and strong reliability and robustness guarantees. They are also very well-suiting for complex querying and non-real time analytics applications.

Potential downsides include lack of data modeling flexibility, scalability, availability, throughput, performance, and schema rigidity.

Schema rigidity refers to the difficulty of changing both database-level schemas and data model (e.g., table) schemas after being used in production. These schemas are rarely correctly and completely determined up front, and often require changes over time. These changes can lead to very time consuming and complex data migrations, along with large potential for errors, and therefore increased QA and testing requirements.

Lastly, RDBMS systems can be fairly complex to install, manage, take full advantage of (e.g., complex queries, stored procedures, triggers, …), and optimize. As a result, these systems have traditionally been worked on by highly specialized people with roles such as Database Administrator (DBA), and less so by software engineers.

NoSQL and NewSQL Databases

NoSQL database systems were created for, and have gained widespread popularity primarily due to benefits relating to scalability and high availability. They favor the eventual consistency model, and these systems typically model and store data in ways other than the traditional tabular relations of relational databases.

These systems are also characterized as being modern web-scale databases that are typically schema-free (dynamic schemas, and do not suffer from schema rigidity), provide easy replication, have simple APIs, and are eventually consistent (BASE). They are best suited for unstructured data and applications involving huge volumes of data, e.g., big data.

Data is being generated and consumed in larger volumes, varieties, and velocities (three V's) than ever before. NoSQL databases are in many ways much better suited for this, and in big data solutions in general relative to their RDBMS counterpart.

There are multiple types of NoSQL databases, with document, key-value, graph, and wide-column being the most prevalent. The different types refer mainly to how the data is stored and the characteristics of the database system itself.

Each type involves tradeoffs between preferred data model, simplicity, querying and operational capabilities, partitionability, consistency, availability, performance, tolerance for object impedance mismatch, and so on.

Benefits of NoSQL databases are simplicity, scalability, flexibility, speed, low latency, high availability, high throughput, and partition tolerance. In fact, many of these systems are designed for extraordinary request and data volumes that can leverage massive horizontal scaling (e.g., thousands of servers) in order to meet demand.

Unlike RDBMS systems, many NoSQL databases can be easily installed, managed, and almost fully utilized by software engineers without requiring a DBA background. This is important since they do not require multiple highly specialized roles, and since having the same people write software and database queries/operations is highly efficient, more agile, faster, and less error-prone.

The simplified database API and querying language used by many of these systems are also very software engineer-friendly and familiar, with JSON as a primary example. This allows for common and developer-friendly language usage between database and application coding.

Potential downsides are the consistency compromise as per the CAP theorem (although many offer consistency at the record or document level), general lack of ACID transactions (although not always) and potential data loss, querying capability, stability, and the bespoke querying languages that these systems employ, although some support common SQL querying as well.

It’s worth noting another type of database system that’s been getting some attention in recent years. NewSQL database systems are relational database systems that combine strong consistency and transactional ACID guarantees with NoSQL-like scalability and performance.

Further discussion of NewSQL is out of scope for this article, particularly since these database systems are relatively new and not yet as established as relational and NoSQL systems. You are certainly encouraged to research this option further.

Putting It All Together

All database systems are not created equal and have certain advantages, disadvantages, strengths, and weaknesses.

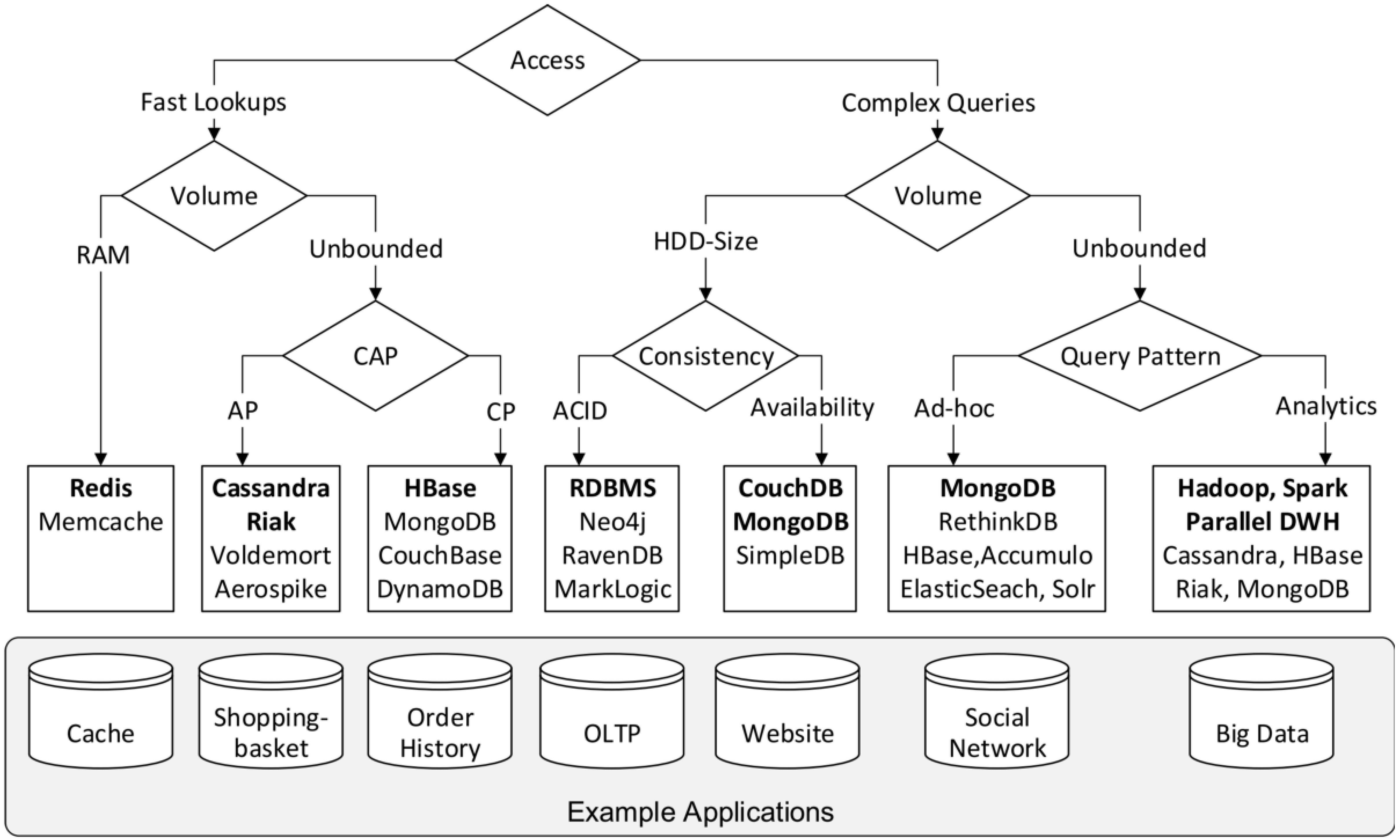
At a high-level, the following should be carefully considered and determined in an order something like the following:

- Identify the type (or combination) of data to be stored, queried, and updated (structured, unstructured, or semi-structured), and the preferred way to model the data if applicable (e.g., schema vs. schemaless, relational, document, key-value, etc.)
  - Note that there are definite advantages and disadvantages to enforcing schemas that are worth looking into
- Decide whether transactional and ACID guarantees are a mandatory requirement
- Determine if horizontal scalability (multiple distributed nodes) is a priority, and if so, prioritize between the tradeoffs outlined by CAP and PACELC
- Ascertain if replication and/or horizontal partitioning (or sharding) is a priority
- Decide any other functional and nonfunctional database requirements and priorities such as those described earlier in this article
  - Particularly items such as security, scalability, performance, vendor establishment and stability, and availability of related skills.
- Determine if an open source database is an option, or if a vendor-backed solution with an SLA is required
- Identify any cost constraints and/or budgets

Once these have been considered, the next step is to research actual database systems and vendors (e.g., relational, NoSQL, NewSQL) to establish which are best suited to your needs.

There are new technologies emerging constantly, so making specific recommendations is not the goal of this article. The right solution will be highly dependent on everything described here, and what the best available options are at the time.

That said, here is a interesting decision tree for choosing NoSQL database systems from an article by Felix Gessert.



Felix Gessert: https://medium.baqend.com/nosql-databases-a-survey-and-decision-guidance-ea7823a822d

Also, software and data solutions can, and often do employ more than one type of data storage system, so definitely consider whether or not that applies to your particular application. If yes, determine the appropriate database system as described above for each.

I hope this article has been helpful, and good luck choosing your next database system!

FACEBOOK 0 TWITTER LINKEDIN 0 0 LIKES

Previous

Data Science, Machine Learning, Artificial Intelligence, Big Data, and IoT Resources

AI, BIG DATA, DATA SCIENCE, MACHINE LEARNING

Next

Artificial Intelligence, Deep Learning, and Neural Networks Explained

AI, MACHINE LEARNING

InnoArchiTech LLC

27 N Wacker Dr, Suite 124

Chicago, IL 60606

hello@innoarchitech.com

Subscribe to our mailing list

email address

Subscribe