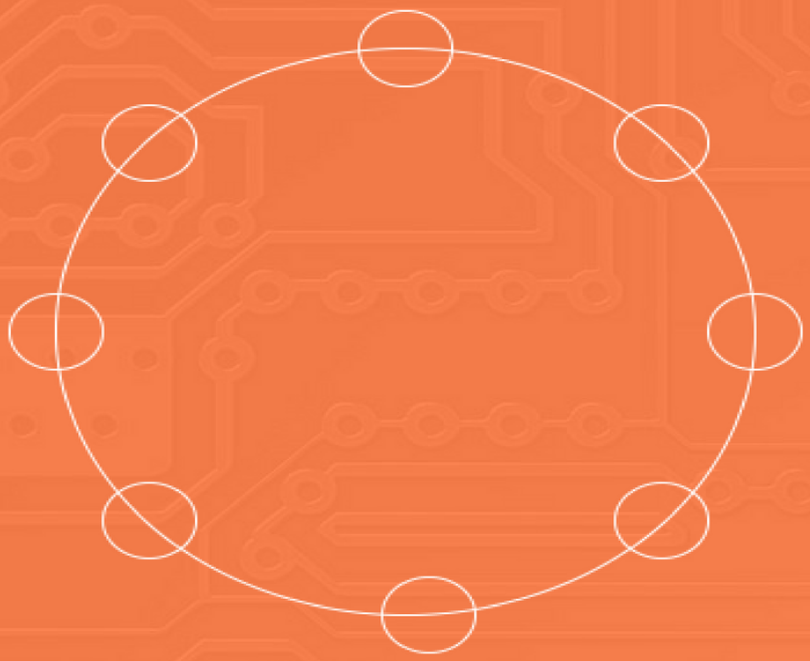


APACHE CASSANDRA FROM THE GROUND UP

By Akhil Mehra

Apache Cassandra From The Ground Up

Akhil Mehra

This book is for sale at http://leanpub.com/apachecassandrafromthegroundup

This version was published on 2023-06-20

Contents

An Introduction To NoSQL & Apache Cassandra **1**
Database Evolution 1
Scaling 3
NoSQL Database 9
Key Foundational Concepts 10
Apache Cassandra 19

An Introduction To NoSQL & Apache Cassandra

Welcome to Apache Cassandra from The Group Up. The primary goal of this book to help developers and database administrators understand Apache Cassandra. We start off this chapter exploring database history. An overview of database history lays the foundation to understand databases. This historical context enables a good understanding of the NoSQL ecosystem and Apache Cassandra's place in this ecosystem. The chapter concludes by introducing Apache Cassandra's its key features and applicable use cases. This context is invaluable to evaluate and get to grips with Apache Cassandra.

Database Evolution

> Those who are unaware of history are destined to repeat it

Let's start with the basics. What is a database? According to Wikipedia, a database is an organized collection of data. Purely mathematical calculations were the primary use of early digital computers. Using computers for mathematical calculations was short lived. Applications grew in complexity and needed to read, write and manipulate data. To cope with the growing complexity companies wrote individual software applications that would enable users to read, write and manipulate data. Early databases stored data sequentially on media such as paper and magnetic tapes. Sequential access made fast retrieval of individual records impossible. The advent of magnetic spinning disk allowed random access to individual records. Advancement in file management led to further random access improvements. The invention of file management systems such as Index Sequential Access Method (ISAM) enabled sequential and random access to files. Improved random access led to the birth of Online Transaction Processing systems (OLTP). Initially, every application wrote its custom code for storing and retrieving data. Everyone writing custom code for data manipulation was an unproductive approach. Database Management Systems (DBMS) were created to address this need. DBMS is a software application/component responsible for storing, manipulating and retrieving data.[1]

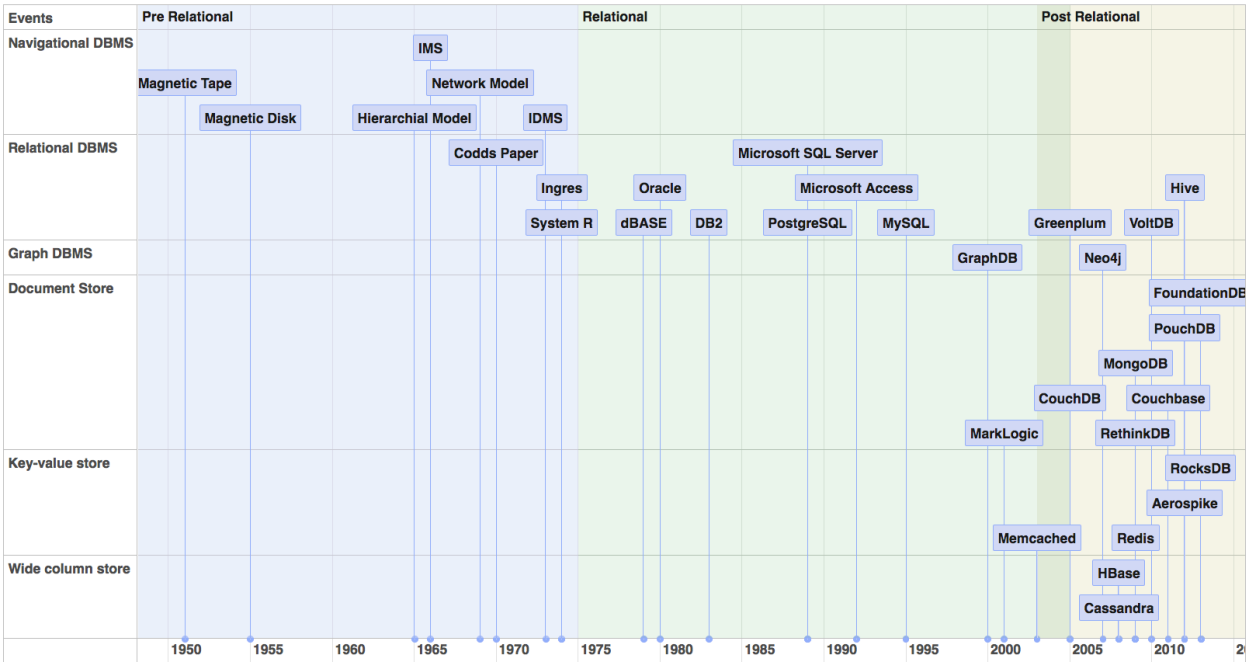
Just like any technology databases have evolved over the past three decades. Database evolution, based on data models, can be broken up into three major eras, i.e., Navigational, SQL/Relational, and Post Relational.[2]

[1] Next Generation Databases: NoSQL, NewSQL, and Big Data
[2] Next Generation Databases: NoSQL, NewSQL, and Big Data

- **Navigational Databases Era** - Navigational database were popular in the 1960's and early 1970's. The primary goal of early DBMS was to provide concurrent data manipulation while maintaining the integrity of the database. It also optimized data retrieval via caching and sophisticated algorithms. Early DBMS ran exclusively on mainframe computer systems. These DBMS's were called Navigational Databases because they made heavy use of pointers and links. Finding data involved traversing these pointers and links. Two main types of navigational data models were the hierarchical model and the navigational model.[3]
- **SQL/Relational Era** - The seminal paper "A Relational Model of Data for Large Shared Data Banks" written by E. F. Codd in 1970 sparked the second database revolution [4]. Codd believed that existing database (Navigational DB's) were too hard to use and lacked theoretical foundation. Codd advocated searching for data by its content instead of following links. His paper laid down the core ideas for the relational data model. The relational model focussed on data presented to users instead of focusing on how data layout on disk. Although Codd's paper provided the foundation for the relational model, it did not define ways of handling concurrent data modification and access. In late 1970's Jim Gray established the most widely accepted transaction model in his paper "The Transaction Concept: Virtues and Limitations"[5]. A few years later Andreas Reuter and Theo Härder coined the term ACID[6] (Atomic, Consistent, Independent, and Durable) that described Jim Gray's set of properties. IBM built the first relational database System R in 1974. IBM's San Jose Research Laboratory developed System R as part of a research project. Initially, researches theorized that a database would struggle to provide both transaction processing and performance. System R was a seminal project which busted this myth. System R also provided the first implementation of Structured Query Language (SQL). The success of System R resulted in the development of many new RDBMS in the succeeding decade. These include Sybase, Microsoft SQL Server, Informix, MySQL, and DB2. These databases relied on three fundamental principles, i.e., the relational model, SQL language, and the ACID transaction model. Relational databases were the de facto choice for application storage needs till the late 2000's[7].
- **Post Relational Era** - The massive explosion in data, i.e., Big Data drove the post relational database revolution. Big data is a broad term for large data sets. These data sets are often complicated and unprocessable by traditional data processing applications. In 2012 Gartner defined Big data as "high volume, high velocity, and/or high variety information assets that need new forms of processing to enable enhanced decision making, insight discovery and process optimization"[8]. Significant challenges around big data include capture, curation, storage, analysis, querying and visualization of these information assets. For over thirty years Relations Database Management Systems (RDBMS) has been the de facto choice for applications data storage needs. The Big Data revolution changed this. It challenged the RDBMS's domination over the storage space. Databases were now required to store massive amounts of structured, semi-structured and unstructured data. The explosion of data, both

[3] Next Generation Databases: NoSQL, NewSQL, and Big Data
[4] A Relational Model of Data for Large Shared Data Banks
[5] The Transaction Concept:Virtues and Limitations
[6] Principles of transaction-oriented database recovery
[7] Next Generation Databases: NoSQL, NewSQL, and Big Data
[8] Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data

structured and unstructured, has made the need to scale and handle non-relational data imperative. International Data Corporation (IDC) estimates that the world's digital information is doubling every two years[9], a large part of which is semi structured or unstructured data. The explosion in big data led to the emergence of a vast number of open source and commercial RDBMS alternatives. These new breeds of databases were called NoSQL database. More on NoSQL database later in this chapter.



Database Era

Scaling

As established in the previous section the post relational era was driven by the need to scale database.

So what is scalability? Scalability is the ability to handle a growing workload in an efficient and cost effective manner.

Vertical vs. Horizontal Scaling

There are essentially two ways to scale:

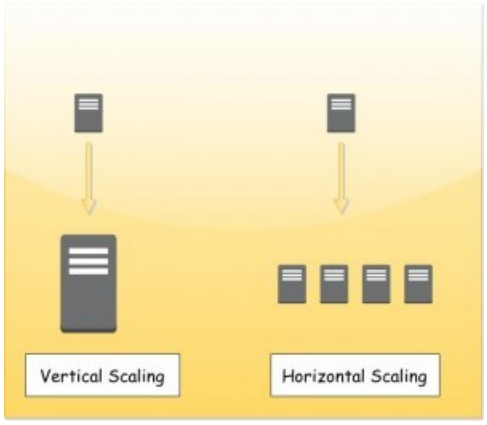
- **Vertical Scaling** - Vertical scaling is also known as scaling up. Vertical scaling refers to adding more resource to a single node, i.e., adding in additional CPU, RAM and Disk to enable a single node to handle a growing workload. Vertical scaling has many limitations the most obvious

[9] Extracting Value from Chaos

one being outgrowing the largest available system. Vertical scaling is also more expensive as your grow. Cost wise scaling vertically is not linear.

- **Horizontal Scaling** - Horizontal scaling is also called scaling out. Horizontal scaling is adding capacity by increasing the number of machines/nodes to a system so that each node can share the processing. Horizontal scaling is a cheaper and more flexible option. This flexibility does come at a cost. Sharing processing and storage amongst an army of nodes is complex. Horizontal scaling makes use of distributed computing to achieve scalability. Andrew S. Tanenbaum defined distributed system as "A collection of independent computers that appears to its users as a single coherent system.". There are three key aspects to a distributed system: These are:
  - Nodes/computers operate concurrently.
  - Nodes/computers fail independently.
  - Computers do not share a global clock.

  Building and maintaining distributed systems is hard. Only use distributed systems when necessary.



Horizontal vs Vertical Scaling

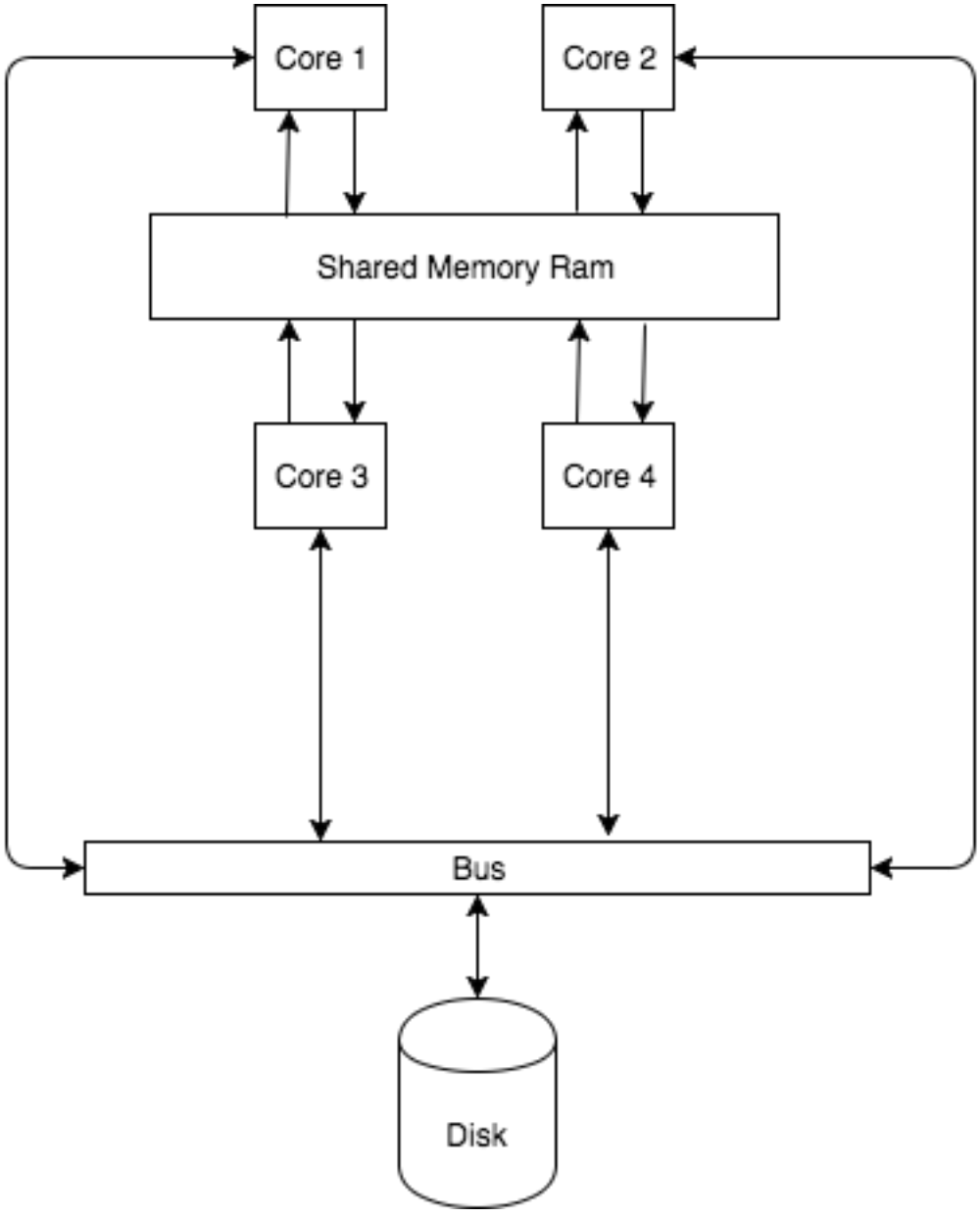
New and emerging technologies prefer to scale horizontally because:

- Increase capacity on the fly.
- Cost effective in comparison to vertical scaling.
- Moreover, in theory, it is infinitely scalable since adding nodes increases capacity proportionally.

Scaling Hardware

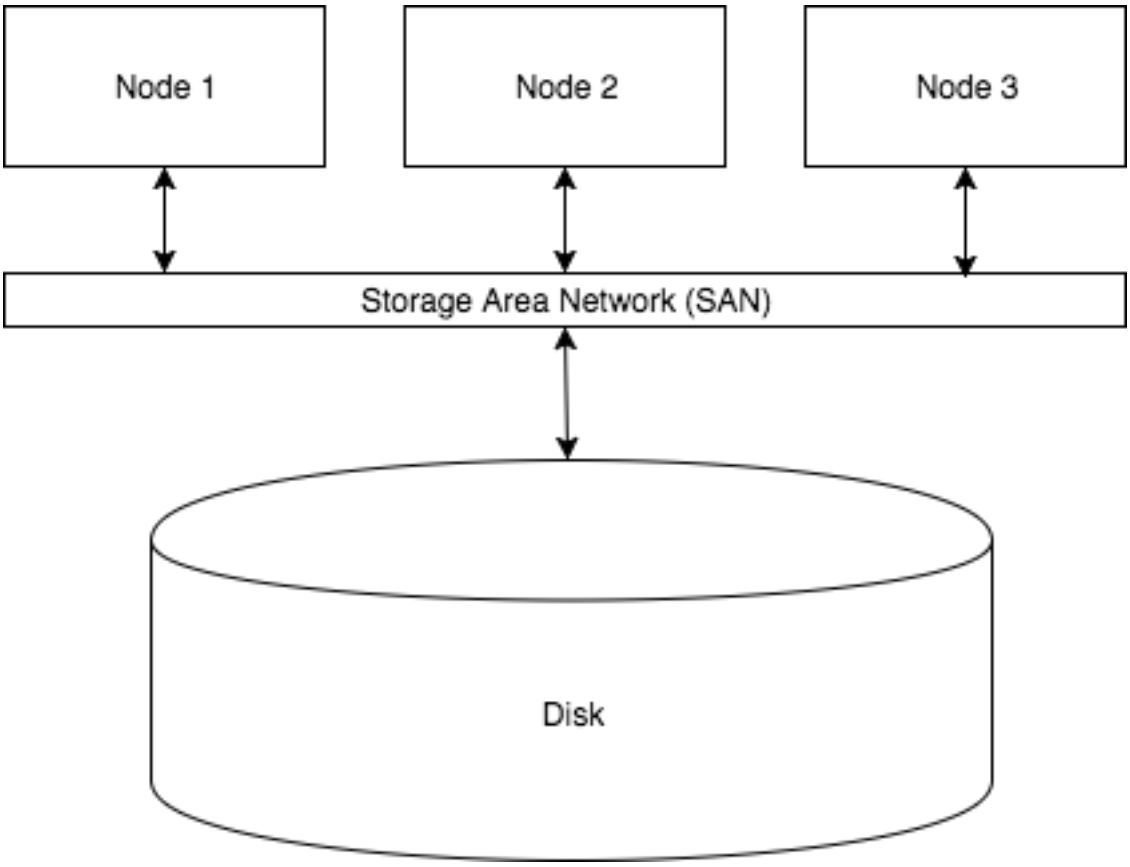
To understand scaling it is important to grasp possible approaches to scaling hardware, i.e., possible hardware deployment architectures. The hardware deployment architecture chosen by a database dictates how it can scale. At a high level, there are three different hardware deployment architectures. These are:

- **Shared Memory, i.e., Traditional Deployment Architecture** - Shared memory is the standard traditional hardware architecture used by database systems. This architecture is characterized by having many cores sharing a block of RAM via a common cache and memory bus. In other words, it is a single machine with many cores accessing a shared memory and single disk. Scaling using this approach (vertical scaling) is buying bigger and better hardware, i.e., you scale by adding more CPU, RAM to your existing machine. Highly parallel shared memory machines are one of the last cash cows of the hardware industry. Traditionally RDBMS database has worked well on shared memory architecture.



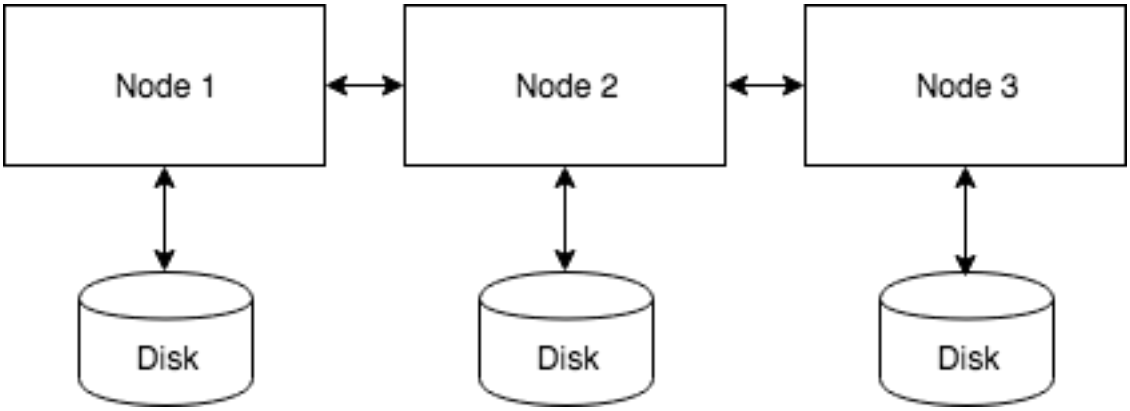
Shared Memory Architecture

- **Shared Disk** - A shared disk approach is characterized by independent nodes which have their own RAM and CPU but share a common disk. Shared disk file systems use Storage Area Network (SAN) to provide direct disk access from multiple computers at a block level. The Shared Disk architecture has gained traction with the rise in popularity of a Storage Area Networks (SAN). Popular RDBMS such as Oracle and MS SQL use a shared disk architecture to scale horizontally.



Shared Disk Architecture

- **Shared Nothing** - A shared nothing architecture is characterized by having a cluster of independent machines communicating over a high-speed network. There is no way of accessing memory or disk of another system. It is up to the database implementor to coordinate efficiently among various machines. Data storage is spread across the cluster as each part of a cluster stores a portion of the data. The main advantage of this approach is the ability to scale. Shared nothing architectures are scalable linearly because there is no single bottleneck in the architecture and have been proven to scale linearly.



Shared Nothing Architecture

Databases designed to scale vertically use a shared memory hardware deployment model. Many modern RDBMS's are deployed into a shared disk architecture to scale. Shared disk based systems have their limits. They are difficult to scale beyond a point as they eventually bottle neck on the centralized disk. Databases designed to scale horizontally use a shared nothing hardware deployment model [10].

Scaling The Web

The emergence of Search, Cloud, Mobile, and Social computing led to a massive explosion in data. Google was one of the first companies to face a big data challenges. Google choose to scale horizontally using three fundamental principles:

- Google met their processing and storage needs by using parallel and distributing processing across a cluster of commodity servers. Google mainly choose to scale horizontally using a shared nothing hardware deployment architecture.
- Using commercial-software was discouraged. Google was committed to building or using open source software for all its computing needs.
- Data Centers were built using commodity hardware primarily commodity-class x86 server computers running customized versions of Linux.

Google's approach to handling big data resulted in the need for distributed storage and distributed processing of huge data sets. This led to the creation of three key pieces of infrastructure.

- **Google File System** (GFS) - A proprietary distributed file system. Its main goal was to provide an efficient, reliable access to data using large clusters of commodity hardware.
- **MapReduce** - A distributed processing model for parallelizing processing of large datasets on a cluster of commodity hardware.
- **Bigtable** - A distributed, non-relational database system that could store massive amounts of data. It used the Google File System for storage.

[10] Architecture of a Database System

Doug Cutting and Mike Cafarella at Yahoo developed Hadoop. Key elements of the Google stack inspired the creation of Hadoop. The base Apache Hadoop framework is composed of the following modules:

- **Hadoop Common** – Contains libraries and utilities needed by other Hadoop modules;
- **Hadoop Distributed File System (HDFS)** – A distributed file-system inspired by GFS that stores data on commodity machines.
- **Hadoop YARN (Yet Another Resource Negotiator)** – A resource-management platform responsible for managing computing resources in clusters and using them for scheduling of user's applications.
- **Hadoop MapReduce** – An implementation of the MapReduce programming model for large-scale data processing.

The Hadoop ecosystem provided an open source implementation of many of the technologies pioneered by Google. Hadoop thus provided an economical way for storing and processing vast volumes of data. Hadoop was primarily designed to handle batch based analytic workloads. The Search, Cloud, Mobile, and Social computing era also needed an on line transaction processing (OLTP) solution for storing big data. This led to the creation of various NoSQL Database.[11]

Scaling RDBMS

The explosion in E-commerce and Web 2.0 revolution led to scalability issues. It was relatively easy to scale the web layer (essentially by adding additional web servers), but the database layer became a bottleneck. Initially, companies choose to scale/up vertically by buying bigger and better hardware. Companies abandoned the scale up solution for three main reasons:

- The scale up solution made the database a single point of failure
- The scale up solution was costly
- Some large companies such as Amazon found that the biggest centralized solutions did not meet their needs.

During the early 2000's MySQL gained in popularity. One of the most popular databases at that time was MySQL. Although far less capable than its commercial counterparts, MySQL was extremely popular especially when building E-Commerce and Web 2.0 websites. MySQL was popular because it was open source and thus free. When hitting single server limits with MySQL engineers looked to scale a DBMS horizontally. Engineers observed that in a typical application reads significantly out numbered writes. To alleviate read pressure from a MySQL instance engineers made use of MySQL's read replication and external caching. Read requests could get directed to other servers by replicating data. Memcache, a distributed object cache, was also extensively used to alleviate read pressure. The use of Memcache with MySQL's read replication lead to a significant boost in read performance.

[11] Next Generation Databases: NoSQL, NewSQL, and Big Data

It is trickier to scale writes. **Sharding** is a popular technique to scale writes. "A database shard is a horizontal partition of data in a database." Each partition is called a shard or database shard. Each shard is held on a separate database server instance, to spread the load.". Sharding although simple in concept was complex in practice.

Memcache, read replication and sharding together provided a shared nothing approach to scaling MySQL. Although straight forward in theory this approach leads to practical issues chiefly boiling down to immense complexity. This complexity led to the birth of NoSQL databases.[12]

NoSQL Database

NoSQL database refers to a group of databases that do not follow the traditional relational data model. Google and Amazon were one of the first companies required to store large amounts of data. They essentially found that storing data in a relational database did not allow them to store vast amounts of data in a cost effective manner. They successfully pursued alternative approaches and published their findings in seminal papers **Bigtable: A Distributed Storage System for Structured Data** and **Dynamo: Amazon's Highly Available Key-value Store** respectively. Rumour has it that Jeff Bezos was livid with the publication of Amazon's paper as he believed that it gave away too much of Amazon's secret sauce. Although there were some NoSQL database before the publication of these papers the NoSQL (Not Only SQL) movement gained popularity, and a number of new open source and commercial NoSQL were inspired by these papers. NoSQL databases have grown in popularity due to their ability to scale horizontally and handle unstructured and semi structured data efficiently.

Key Features of a NoSQL Database

The main characteristics of a NoSQL database are:

- **Based on distributed computing** - Unlike traditional RDBMS, NoSQL databases have been designed to favor distributed computing and a shared nothing architecture. This is because scaling horizontally is believed to be the only cost effective way of handling large volumes of data. Additionally horizontally scaling databases is a simpler way to handle large workloads.
- **Commodity Hardware** - Most NoSQL databases have been designed to run on cheap commodity hardware (in reality high-end commodity hardware) instead to high-end servers. Commodity hardware enables scaling in a cost effective manner.
- **Provide a flexible schema** - To store the large growing amount of semi structured and unstructured data developers need a flexible solution that easily accommodates different types of data. Additionally, due to the constant change in requirements, a schema which is easily evolvable is also desirable. Thus most new NoSQL databases provide a flexible schema which can be evolved as opposed to the rigid schemas required by RDBMS. Schema flexibility has made working with semi structured and unstructured data a lot easier.

[12] Next Generation Databases: NoSQL, NewSQL, and Big Data

NoSQL Database Categories

NoSQL databases are categorized according to their data model. Broadly there are four categories:

- **Key-Value databases** - Key value stores provide a simple form of storage that can only store pairs of keys and values. Values stored are essentially blobs and are retrieved based on keys provided. Some key values stores persist data to disk while others such as Memcached keeps data in memory only. Riak, Redis, Amazon Dynamo DB, FoundationDB, MemcacheDB, and Aerospike are examples of popular key value stores.
- **Document Databases** - Document Stores are an advanced form of a key value store where the value part store is not a blob but a well-known document format. XML, JSON, BSON are popular document formats. A specified document format enables the database to examine the document. It also enables the database to do operations on the document. Popular Document stores include RavenDB, Apache CouchDB, Couchbase, MarkLogic and MongoDB
- **Column Family Databases** - Column family based (not to be confused with column oriented) database are again an evolution of the key value store where the value part contains a collection of columns. Each row in a column family has a key and associates arbitrary number columns with it. Column Family are useful for accessing related data together. Popular column family based databases include Apache Cassandra, HBASE, and Hypertable.
- **Graph Databases** - A graph database is one which uses a graph structure to store data. Graph databases enable you to store entities and establish relationships between these entities.

NoSQL Design Decisions

One of the key design decisions made by NoSQL database was to trade one or more of the ACID (atomicity, consistency, isolation, and durability) properties for BASE properties (Basic Availability, Soft-state, Eventual consistency). NoSQL databases use distributed computing thus have chosen BASE (Basic Availability, Soft-state, Eventual consistency) over ACID. While ACID is a pessimistic approach and forces consistency at the end of each transaction, BASE is an optimistic approach where by it accepts that data is in a state of flux but will eventually sort itself out. Choosing BASE over ACID enables systems to scale horizontally thus being able to capture and query big data.

Key Foundational Concepts

Even thought it may feel theoretical there are some key concepts that get bandied around all the time. When coming for a non distributed world these concepts may feel alien. Even worse these concepts are often misunderstood which adds to the confusion. It is critical to have a good understanding of these concepts to make informed choices about using or designing a distributed data system.

Computer science researches love mnemonic. They provide as a means to understand difficult concepts. Although often imprecise these mnemonic help direct the conversation and research in the right direction. ACID, BASE and the CAP and PACELC are important mnemonic that we must

understand. Good understating of these concepts will help us understand some the the NoSQL design choices.

ACID

Jim Gray in 1970 defined a set of properties desirable all storage engines. The properties looked to ensure reliable processing of database transactions (single logical operation). A few years later Andreas Reuter and Theo Härder officially coined the term ACID that described Jim Gray's set of properties. ACID compliance became a de facto standard for all databases. ACID acronym stands for:

- **Atomicity** - Atomicity is not about concurrency. Atomicity refers to all or nothing state changes as a result of a transaction. A transaction is a logical unit of work or a group of related changes. A database client defines the scope of a transaction. Either the entire logical unit of related changes take effect, or none of them take effect. Atomicity ensures incomplete state changes are not visible to the client. Atomicity enables databases to rollback a group of changes. As suggested by Martin Kleppmann abort is a better word to describe atomicity. [13]
- **Consistency** - A database is in a consistent state if it conforms to all rules defined in the database. The consistency property in ACID ensures that all data written to a database is always valid according to all rules defined in the database. Rules can include database constraints, cascades, triggers and any combination of the aforementioned rules. Consistency ensures that no database rules are ever violated as a result of transaction execution. Consistency in ACID guarantees data integrity with regards to database rules.
- **Isolation** - Must make sure that no transaction has access to data as a result of an unfinished or currently processed transaction. Each transaction is independent and is not affected by other transactions. Of the four ACID properties, the isolation property is the most configurable and often the most relaxed. The ANSI/ISO SQL standard defines a number isolation levels that are implemented by most DBMS. These include serializable, repeatable reads, read committed, read uncommitted. Serializable is the only isolation level that ensures that no transactions have access to data as a result of an unfinished transaction. A database with serializable isolation level is just too slow and impractical for most applications. In fact, most database I have worked with do not ship with serializable isolation as their default isolation level. Databases have introduced a variety of isolation levels so that applications can use an appropriate isolation level for their particular use case. Each of the mentioned isolation levels has its subtle nuances which are important to understand. Martin Kleppmann talk Transactions: myths, surprises and opportunities[14] is an awesome talks that explains ACID very well.
- **Durability** - Ensures that the results of completed transactions permanently stored.

Data Replication

Distributed systems replicate data, i.e., make copies of data and store these copies on different nodes. Data replication:

[13] Transactions: myths, surprises and opportunities
[14] https://martin.kleppmann.com/2015/09/26/transactions-at-strange-loop.html

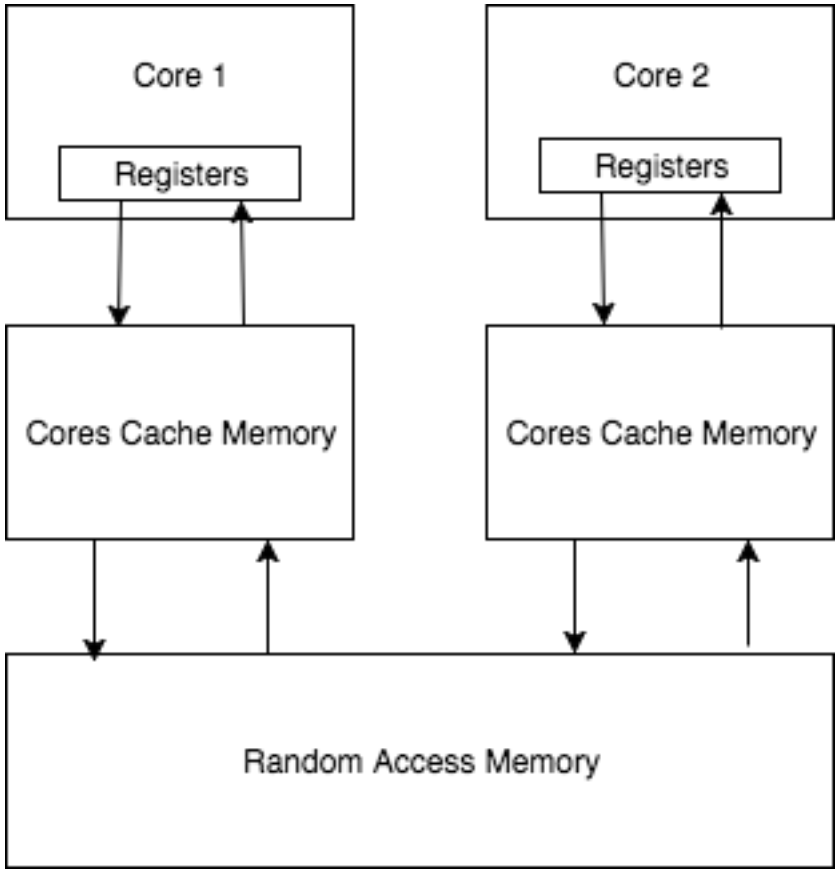
- Ensure continued access to the data in the event of node failure.
- Can locate data geographically close to client
- Enables scaling by increasing the number of nodes that can handle client queries.

Consistency

Consistency is a highly overloaded term that causes a great deal of confusion. Consistency means different things in the RDBMS and distributed computing communities. NoSQL brought these worlds together and thus the confusing overloaded terminology. Understanding the various usage of consistency is important. Understanding the different interpretations of consistency enables readers to understand read and write operations on both RDBMS and NoSQL data stores.

Consistency refers to different things depending on the context it is used in:

- Consistency can refer to the C in the famous ACID mnemonic. As mentioned above this is simply adherence to DB rules when executing transactions.



Consistency in Shared Memory Architectures

- Consistency can refer to **consistency models**, i.e., guarantees provided while reading and writing data in a distributed systems or shared memory system. The challenges faced when sharing memory among a number of cores is similar to problems faced by distributed system. Cores do not read and write data directly to memory. They read and write to a CPU via a cache. The main challenge is a shared memory system is maintaining consistency between CPU caches that read and write data to a shared memory. In distributed systems the challenge lies in ensuring consistency between data replicated across nodes. Thus the concurrency models are applicable to both shared memory architectures and distributed systems. Concurrent writes to a replicate store can lead to inconsistent data. This can make data stores very confusing for programmers. Consistency models specify a contract providing a programmer predictability. Literature on consistency models often refer to reading and writing to registers as they were initially used Think of this as a reading or writing a single value. The single value could be a single value in a key value store or a single row in a database. There are many types of consistency models. Following are some the of important consistency models that one should be aware of:
  - **Linearizability aka Strong Consistency** - Linearizability main goal is to provide a single global state for a distributed system. Linearizability guarantees that values read are the most recent up to date copy of the data. Linearizability also guarantees cluster wide order. All operations are atomic and every client has the same view of the data. Every node/process agrees on the order of the operation. Thus a client can connect to any node in a distributed systems and be assured that he/she is reading the most up to date copy of the data.
  - **Sequential Consistency** - Is a weaker form of consistency. It guarantees that writes to a register appear in the same order to all nodes. However, it does not guarantee that all writes are seen instantaneously by all nodes. It guarantees order but does not guarantee recency of data. Two clients reading from two different nodes are not guaranteed to read the same value. However they will see updates in the same order.
  - **Causal Consistency** - Causal consistency is a weaker form of sequential consistency. The order is not guaranteed over all operations but only over causally related operations.
  - **Pipelined Random-Access Memory (PRAM) Consistency** - PRAM Consistency is a weaker consistency model than causal consistency. It ensures write operations performed by a single process are seen by all other processes in the order in which they were performed on that processor. Write operations on each processor is performed as if all write were written into a pipeline for each processor.
  - **Eventual Consistency** - Out of all the above consistency levels, eventual consistency is the weakest form of consistency. Eventual consistency makes no guarantees for returning the most recent write. It just guarantees that provided there are no new writes systems will eventually become consistent over time. There are no guarantees as to how much time this might take. A better word for eventual consistency is convergence as replicated data will eventually converge to the same value.
- Database isolation levels i.e the I in ACID are often gets confused with consistency models. The first thing to note is that database isolation levels apply to transactions (a group of

operation) while consistency models described above are only applicable to single values, i.e., a value written and read from a register. Isolation level that defines the degree to which one transaction must be isolated from data modifications made by other transactions. As opposed to consistency it has nothing to do with the order of read and write operations.
  - **Serializability vs. Linearizability** - The serializable isolation level specifies that all transactions in a system appear as if they have occurred in a completely isolated fashion. All transactions appear as if they have occurred serially one after another. Transactions can execute at the same time only if they can maintain the illusion of serial execution. The main goal of the serializable isolation level is to preserve correctness. Unlike linearizability, serializability does not impose deterministic order. Let's say we have two transactions T1 and T2. T1 started before T2. Serializability does not impose any order. Thus transactions T2 can finish before or after T1 and still be in line with serializability guarantees. Serializability only requires that database provides the illusion of being serially executed.
- Strict Serializability refers to combining serializability and linearizability, thus guarantying both isolation and order.

BASE

Eventual consistency (do not confuse with consistency in ACID) is a consistency model that guarantees all changes are eventually replicated across the replica set. Thus eventually all replicas return the same last updated value. Eventually consistent databases are often said to provide BASE (**B**asically **A**vailable, **S**oft state, **E**ventually consistency) semantics. BASE refers to:

- **Basically available** indicates that the system does guarantee availability, in terms of the CAP theorem, i.e., every nonfailing node returns a response in a reasonable amount of time.
- **Soft state** indicates that the state of the system may change over time, even without user input. A database is in a soft state because of it uses an eventually consistency model.
- **Eventual consistency** indicates that the system becomes consistent over time, given that the system does not receive input during that time.

ACID and BASE represent two opposite ends of the consistency-availability design spectrum. The ACID properties focus on ensuring consistency in a system while BASE properties focus on making systems available.

CAP Theorem

The CAP theorem is a tool used to makes system designers aware of trade-offs while designing networked shared-data systems. CAP has influenced the design of many distributed data systems. It made designers aware of a wide range of tradeoff to consider while designing distributed data systems. Over the year the CAP theorem has been **widely misunderstood tool** used to categorize databases. There is much misinformation floating around CAP. Most blog posts around CAP are historical and possibly incorrect.

> It is important to understand CAP so that you can identify a lot of the misinformation around it.

The CAP theorem applies to distributed systems that stores state. Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC) conjectured that in any networked shared-data system there is a fundamental trade-off between consistency, availability, and partition tolerance. In 2002 Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer's conjecture[15].

The theorem states that **networked shared-data systems** can only guarantee/strongly support two of the following three properties:

- **Consistency** - A guarantee that every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data. There are various type of consistency models. Consistency in CAP (used to prove the theorem) refers to linearizability or sequential consistency a very strong form of consistency.
- **Availability** - **Every** non-failing node returns a response for all read and write requests in a reasonable amount of time. The key word here is every. To be available every node on (either side of a network partition) must be able to respond in a reasonable amount of time.
- **Partition Tolerant** - The system continues to function and uphold its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

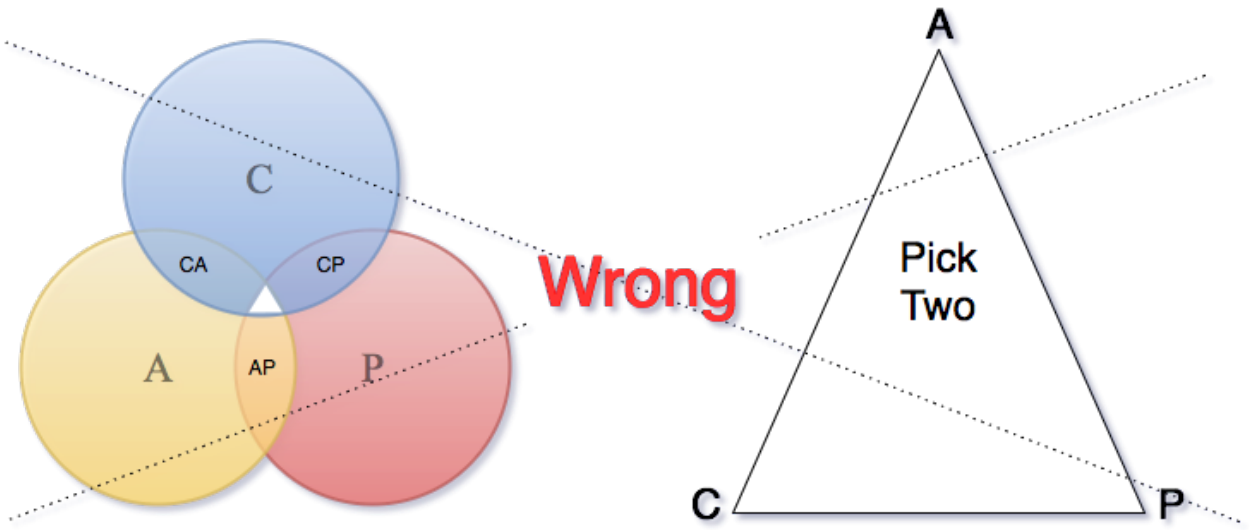
> The C and A in ACID represent different concepts than C and in A in the CAP theorem.

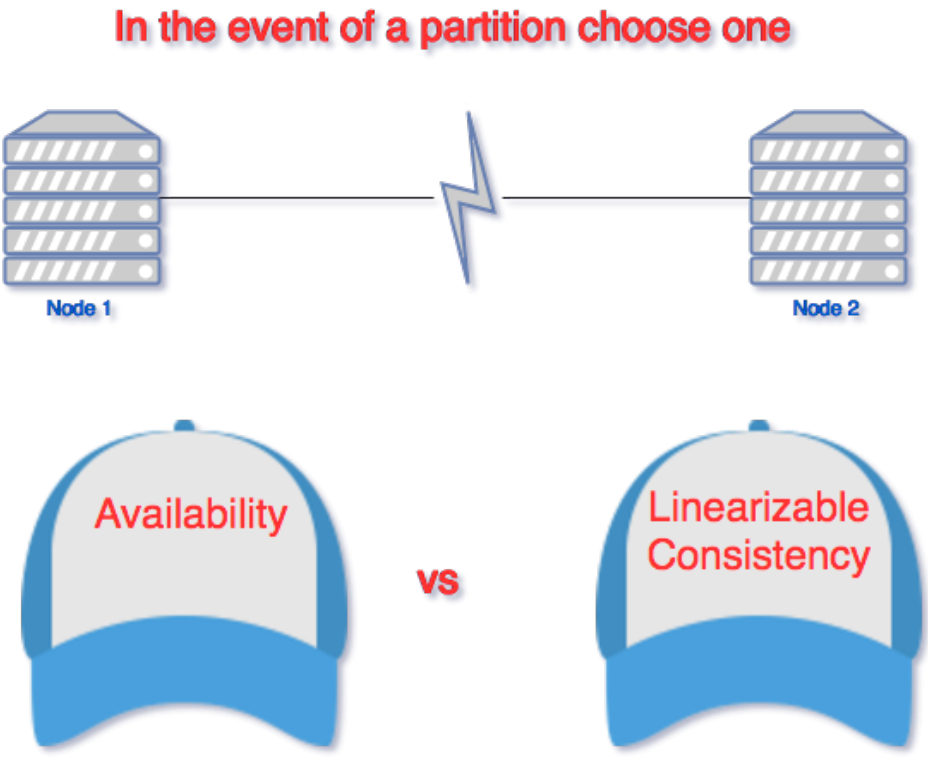
The CAP theorem categories systems into three categories:

- CP (Consistent and Partition Tolerant) - At first glance, the CP category is confusing, i.e., a system that is consistent and partition tolerant but never available. CP is referring to a category of systems where availability is sacrificed only in the case of a network partition. [16]
- CA (Consistent and Available) - CA systems are consistent and available systems in the absence of any network partition. Often a single node DB servers are categorized as CA systems. Single node DB servers do not need to deal with partition tolerance and are thus considered CA systems. The only hole in this theory is that single node DB systems are not a network shared data system and thus do not fall under the preview of CAP. [17]
- AP (Available and Partition Tolerant) - These are systems that are available and partition tolerant but cannot guarantee consistency.

[15] Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services
[16] Problems with CAP, and Yahoo's little known NoSQL system
[17] Problems with CAP, and Yahoo's little known NoSQL system

A Venn diagram or a triangle is frequently used to visualize the CAP theorem. Systems fall into the three categories that depicted using the intersecting circles.





CAP Theorem

The part where all three sections intersect is white because it is impossible to have all three properties in networked shared-data systems.

A Venn diagram or a triangle is a **incorrect visualization** of the CAP. Any CAP theorem visualization such as a **triangle or a Venn diagram** is a misleading.

The correct way to think about CAP is that in case of a network partition (a rare occurrence) one needs to choose between availability and partition tolerance. Instead of choose two is more like choose one.

In any networked shared-data systems partition tolerance is a must. Network partitions, dropped messages are a fact of life and must be handled appropriately. Consequently, system designers must choose between consistency and availability. Simplistically speaking a network partition forces designers to either choose perfect consistency or perfect availability. Picking consistency means not being able to answer a clients query as the system cannot guarantee to return the most recent write. This sacrifices availability. Network partition force nonfailing node to reject clients request as these nodes cannot guarantee consistent data. At the opposite end of the spectrum being available means being able to respond to a clients request but the system cannot guarantee consistency, i.e., the most recent value written. Available systems provide the best possible answer under the given circumstance.

> During normal operation (lack on network partition) the CAP theorem does not impose constraints on availability or consistency.

The CAP theorem is criticized for being too simplistic and often misleading [18] [19]. A decade after the release of the CAP theorem Brewer acknowledge that the CAP theorem oversimplified the choices available in the event of a network partition. According to Brewer, the CAP theorem prohibits only a "tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare" [20]. System designers have a broad range of options for dealing and recovering from network partitions. The goal of every system must be to "maximize combinations of consistency and availability that make sense for the specific application" [21].

The CAP theorem is a simple straw man to make system designers aware of trade-offs while designing networked shared-data systems. It is a simple starting point and has been widely used to design and discuss tradeoff in NoSQL database.

PACELC

Daniel Abadi suggested PACELC as an extension for CAP. The central thesis of PACELC is that ignoring the consistency/latency tradeoff of replicated systems is a major oversight in CAP. Abadi

[18] Please stop calling databases CP or AP
[19] Problems with CAP, and Yahoo's little known NoSQL system
[20] Please stop calling databases CP or AP
[21] Please stop calling databases CP or AP

argued that the consistency latency tradeoff is far more significant as it presents it self during normal system operation [22]. PACELC suggest that in case of a network partition (P) one has to tradeoff between availability and consistency (A and C); else (E) normally running systems (absence of partitions) need to trade-off between latency (L) and consistency (C) [23]. Abadi observed that systems that give up consistency in the event of a partition also give up consistency in favor of lower latencies.

- **PA/EL** - In the event of a partition the system favors availability. Under normal operation, latency is favored over consistency.
- **PC/EC** – In the event of a partition the system favors consistency. Under normal operation the system favors consistency.
- **PA/EC** – In the event of a partition the system favors availability. Under normal operation the system favors consistency.
- **PC/EL** – In the event of a partition the system favors consistency. Under normal operation the system latency. PC/EL is an unusual combination as very few systems would prefer consistency under a partition scenario while choosing latency over consistency under normal operation.

PACELC like all other mnemonic is not perfect. Many systems cannot be assigned to a single PACELC as they leave this tradeoff to clients on a per query basis. Additionally PACELC like PC/EL is unusual and there are hardly any systems that use this classification.

Delay-Sensitivity Framework (DSF) an Alternative to CAP

Martin Kleppmann proposes an alternative to the CAP theorem. Kleppmann acknowledges the usefulness of PACELC and builds on it.

Kleppmann proposes the delay sensitivity framework as an alternative to CAP for reasoning about tradeoffs between consistency guarantees and network faults tolerance in a replicated database[24]. Operations are categorized by their latency or non-latency sensitivity. The table is an example of DSF for read and write operations applied to three consistency levels.

Consistency Level	Write Operation	Read Operation
Linearizable consistency	Function of network delay O(d)	Function of network delay O(d)
Sequential consistency	Function of network delay O(d)	Constant time O(1)
Causal consistency	Constant time O(1)	Constant time O(1)

Latency sensitivity is compared with service level agreement (SLA) to determine if a system is acceptable under a network fault occurs.

Kleppmann major grouses with CAP is it vagueness regarding defining consistency and availability. Kleppmann proposes five parameters that can be used to reason and compare networked share data

[22] Consistency Tradeoffs in Modern Distributed Database System Design
[23] Problems with CAP, and Yahoo's little known NoSQL system
[24] A Critique of the CAP Theorem

systems. The suggested parameters include:

- **Availability** - Availability is a metric that can be measured. In the delay-sensitivity framework, availability refers to the percentage of successful requests over some period of system operation.
- **Delay-sensitive** - Delay-sensitive operations refers to tasks whose latency is determined by network throughput. At the other end, delay-independent operations are in no way affected by network throughput. It is suggested the one identifies delay sensitive and delay nonsensitive operations
- **Network faults** - Systems must plan and account for all kinds of network faults. Partitions are just form or network faults.
- **Fault tolerance** - Refers to defining the systems fault tolerance, i.e., what exactly can be tolerated and what cannot. This is in favor of using imprecise words such as high availability. A good example is specifying the exact number of nodes that need to be up for the system to function. Kleppmann encourages users to stay away from vague terms such as high availability.
- **Consistency** - Refers to assigning a specific consistency model to the system. There are numerous consistency models often vague and imprecise. Kleppmann encourages assigning only well defined and exact consistency models.

> CAP, PACELC, DSF can all get overwhelming and confusing. My simple take them are: - **CAP** - High level and an imprecise choice between consistency and availability in the event of a network partition. - **PACELC** - High level consistency vs latency tradeoff during normal systems normal operation. - **Delay Sensitivity Framework** - A detailed approach to thinking about consistency vs latency tradeoff.

Apache Cassandra

Apache Cassandra is an open source distributed storage system which was initially developed at Facebook and in 2009 open sourced to the Apache Foundation. Cassandra was conceptualized in one of Facebook's hackathons to solve their storage needs for Facebook's Inbox search problem. The original note by Avinash Lakshman[25], the person credited with creating Cassandra, outlines his vision for Cassandra. Over the past eight years, Cassandra has come a long way. DataStax has a very interesting write up comparing[26] the original Cassandra paper with Apache Cassandra latest 2.0 release highlighting how Apache Cassandra has evolved over the years. Today Cassandra is used by many companies including NetFlix, Instagram, eBay, Twitter, Reddit, and Apple. Recently it was revealed that Apple has one of the largest production Cassandra deployment, a mind blowing 75,000 nodes storing over 10 PB of data.

Key features provided by Apache Cassandra are:

[25] https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919
[26] http://www.datastax.com/documentation/articles/cassandra/cassandrathenandnow.html

- **Distributed Storage System** - Databases, in a nutshell, have two main tasks storage and computation. Distributed systems accomplish the two central tasks by using an army of nodes. Cassandra runs on a cluster of nodes using a shared nothing architecture as it is designed to store huge amounts of data.
- **Runs on Commodity Hardware** - Like most distributed systems Apache Cassandra has been designed to run on commodity hardware. Commodity Hardware enables Cassandra to scale in a cost effective manner.
- **Fault Tolerant** - When running an application on a cluster of nodes network, hardware and system failures are a reality. Cassandra has been designed from the ground up to work acceptably in spite of faults in the system. Cassandra enables applications to tradeoff consistency and availability at a granular level.
- **Linearly Scalable** - Cassandra is linearly scalable, i.e., doubling the number of nodes in your cluster doubles the performance. This linear scalability has enabled Cassandra to handle terabytes of data and thousands of concurrent operations per second. Netflix has done a benchmark[27] where it showns that Cassandra scales linearly and can perform one million writes per second.
- **Atomicity, Isolation and Durability (AID) Support** - Cassandra is touted as having AID support. Understanding AID support can get confusing especially if you are comparing AID to ACID in RDBMS. Cassandra does not support transactions in the RDBMS sense. This makes atomicity and isolation in an RDBMS incomparable to atomicity and isolation in Cassandra. Since Cassandra does not have transaction, a group of related changes cannot be aborted in Cassandra. All Cassandra guarantees are atomic writes for a single write operation on each node at the CQL row level. Atomicity in Cassandra ensures that writes are only visible when successful. In the same vein isolation is also not comparable. Due to the lack of transactions, Cassandra cannot isolate a group of operations from one another. Durability is the only part of the mnemonic that is loosely comparable. Both Cassandra and RDBMS ensure that data once written is permanent. An RDBMS and Cassandra both use a commit log to achieve durability.
- **Elastically Scalable** - Apache Cassandra can elastically scale, i.e., it can cope with growing/shrinking loads dynamically. It can expand and shrink resources according to the workload. Elastically scalability is of particular importance when using cloud resources as cloud resources follow a pay-per-use model.
- **Multi Data Center** - Cassandra is architected so that it can be easily deployed across multiple data centers. Clusters can be configured to geographically distribute data to cater for redundancy, fail over and disaster recovery.
- **Open Source** - Apache Cassandra is open source software distributed under the Apache 2.0 license. The Apache 2.0 license is a permissive license that allows the user of the software freedom to use the software for any purpose.

Apache Cassandra Key Benefits

There are four key benefits to using Apache Cassandra. These are availability, scale, performance, and cost.

[27] http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html

- **High Availability** - Apache Cassandra is a highly available fault tolerant database. The precise definition of high availability is application dependent. Cassandra is designed using a peer-to-peer architecture. A Cassandra cluster does not have any special nodes. A peer-to-peer architecture is inherently available and scalable. One of the key tenets build into Apache Cassandra design is operational simplicity. Various operational tasks can be carried out without any downtime. Dealing with hardware failures, increasing capacity, and other maintenance tasks can be done without any downtime. Hardware failures are a fact of life. Apache Cassandra has been designed to deal with these gracefully. Node failures need not have an impact on running databases. Nodes are replaceable in a running production database. Similarly adding capacity to a production database can be done on the fly without any downtime. One can add and remove a node from the cluster when the database is in operation. Apache Cassandra supports distributing data across multiple-data centers. Multiple data center support come out of the box and is very easy to configure. Apart from obvious performance benefits, multiple data center support provides an additional layer of fault tolerance and disaster recovery.
- **Scale** - Cassandra has been designed from the ground up for massive scale. There are numerous Cassandra deployments spanning hundreds/thousands of nodes holding multi-terabyte/petabytes of data. As mentioned in the previous section Cassandra has been proven to be linearly scalable. Cassandra is also elastically scalable, and thus capacity can be increased and decreased on the fly. Cassandra also provides multiple-data center support which can be used for optimal geographic distribution of data.
- **Performance** - Apache Cassandra is known for it high write throughput due. Netflix benchmarks have shown Cassandra to sustain 1 million writes a second. Although Cassandra is known for its write speed, it is equally efficient at reading data. Cassandra works well for applications that have a high currency requirement. One of the main performance bottlenecks in traditional databases has been locking. There is no locking in Cassandra which leads to an enormous performance boost. The lack of locking results in efficient handling of concurrent requests.
- **Cost** - Apache Cassandra is an open source database and thus has no licensing cost. Anyone can download and deploy Cassandra for free. Cassandra has been designed to run on commodity hardware thus saving hardware costs. Its elastic scalability helps optimize the use of cloud resources. Apache Cassandra has a flexible data model which helps minimize costs associated with evolving an application.

Apache Cassandra Uses Cases

Apache Cassandra is a good fit for storing a large amount of data in a cost effective manner. Cassandra shines when one needs a distributed, failure tolerant database. Cassandra is an excellent candidate for high-throughput systems. Large deployments with the need to geographically distribute data should seriously consider Apache Cassandra. The sweet spot for a typical Cassandra use case is time series data. Examples include event stores, financial transactions, and sensor data. A variety of industries uses Cassandra. Banks and other financial institutions have large quantities of time services data and are using Cassandra to meet their storage needs. Similarly, many IoT based companies are using Cassandra to store various forms of sensor data.

Do not use Cassandra :

- If your application requires strong consistency
- If you have complicated queries that keep changing. Cassandra is not suitable of ad-hoc querying. You need to know your queries ahead of time. You data model closely linked to your queries.
- Small data loads. Do not use Cassandra at small data loads. If your application can scale using an RDBMS then please stick to an RDBMS.

In the next chapter, we will install and play with Apache Cassandra's.