Lecture #23: Introduction to Distributed Databases

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Distributed DBMSs

A distributed DBMS divides a single logical database across multiple physical resources. The application is (usually) unaware that data is split across separated hardware. The system relies on the techniques and algorithms from single-node DBMSs to support transaction processing and query execution in a distributed environment. Important goals of using a distributed DBMS are fault tolerance (avoiding a single node failure taking down the entire system) and scalability (storing data that does not fit in a single node).

The differences between **parallel** and **distributed** DBMSs are:

Parallel Database:

- Nodes are physically close to each other.
- Nodes are connected via high-speed LAN (fast, reliable communication fabric).
- The communication cost between nodes is assumed to be small. As such, one does not need to worry about nodes crashing or packets getting dropped when designing internal protocols.

Distributed Database:

- Nodes can be far from each other.
- Nodes are potentially connected via a public network, which can be slow and unreliable.
- The communication cost and connection problems cannot be ignored. Nodes can crash and packets can get dropped.

2 System Architectures

A DBMS's system architecture specifies what shared resources are directly accessible to CPUs. It affects how CPUs coordinate with each other and where they retrieve and store objects in the database.

A single-node DBMS uses what is called a *shared everything* architecture. This single node executes workers on a local CPU(s) with its own local memory and a local disk.

Shared Nothing

In a *shared nothing* environment, each node has its own CPU, memory, and disk. Nodes only communicate with each other via network. Before the rise of cloud storage platforms, the shared nothing architecture was the common way to build distributed DBMSs since it can be built using the off-the-shelf servers.

It is more difficult to increase capacity in this architecture because the DBMS has to physically move data to new nodes. It is also difficult to ensure consistency across all nodes in the DBMS, since the nodes must coordinate with each other on the state of transactions. The advantage, however, is that shared nothing DBMSs can potentially achieve better performance and are more efficient then other types of distributed DBMS architectures.

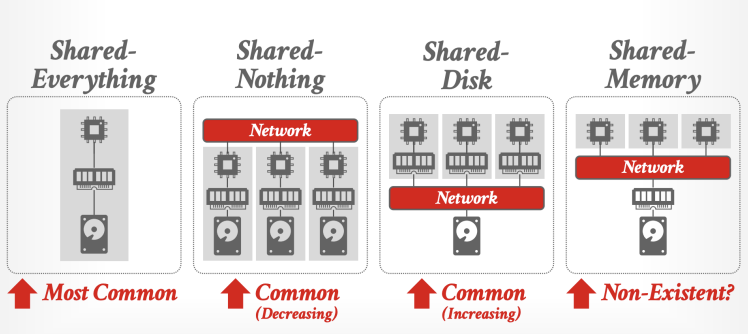

Figure 1: Database System Architectures – Four system architecture approaches ranging from sharing everything (used by non distributed systems) to sharing memory, disk, or nothing.

Shared Disk

In a *shared disk* architecture, all CPUs can read and write to a single logical disk through a network, but each have its own private memory. Each compute node usually has a small local disk to cache the data from the shared disk. This approach is more common in cloud-based DBMSs, facilitating data lakes and serverless systems.

Amazon S3. GCS

By decoupling DBMS's storage layer from its execution layer, we can scale them independently. Adding new storage nodes or execution nodes does not affect the layout or data location in the other layer.

Nodes must send messages between them to learn about other node's current state. That is, since memory is local, if data is modified, changes must be communicated to other CPUs in the case that piece of data is in main memory for the other CPUs.

! propagation

Shared Memory

In a *shared memory* architecture, CPUs have access to common memory address space via a fast interconnect. They also share the same disk. Each processor has a global view of all the in-memory data structures.

In practice, most DBMSs do not use this architecture because the network as fast as CPU and memory is expensive.

3 Design Issues

Distributed DBMSs aim to maintain *data transparency*, meaning that users should not be required to know where data is physically located, or how tables are partitioned or replicated. The details of how data is being stored is hidden from the application. In other words, a SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

The key design questions that distributed database systems must address are the following:

- How does the application find data?
- Where does the application send queries?
- How should queries be executed on a distributed data? Should the query be pushed to where the data is located? Or should the data be pooled into a common location to execute the query?
- How should the database be divided across resources?
- How does the DBMS ensure correctness?

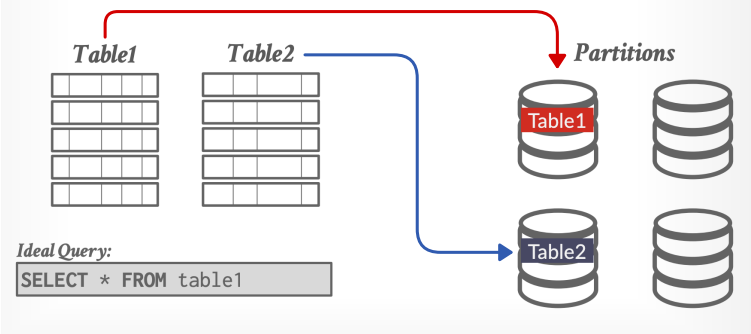

Figure 2: Naive Table Partitioning – Given two tables, place all the tuples in table one into one partition and the tuples in table two into the other.

4 Partitioning Schemes

Distributed system must partition the database across multiple resources, including disks, nodes, processors. This process is sometimes called *sharding* in NoSQL systems. When the DBMS receives a query, it first analyzes the data that the query plan needs to access. The DBMS may potentially send fragments of the query plan to different nodes, then combines the results to produce a single answer.

The goal of a partitioning scheme is to maximize single-node transactions, or transactions that only access data contained on one partition. This allows the DBMS to not need to coordinate the behavior of concurrent transactions running on other nodes. On the other hand, a distributed transaction accesses data at one or more partitions. This requires expensive, difficult coordination, discussed in the below section.

Implementation

The simplest way to partition tables is *naive data partitioning*. Each node stores one table, assuming enough storage space for a given node. This is easy to implement because a query is just routed to a specific partitioning. However, this does not scale and is suboptimal when queries join data across tables or when there is non-uniform access patterns (some nodes are more utilized than others). See Figure 2 for an example.

Another way of partitioning is *vertical partitioning*, which splits a table's attributes into separate partitions. Each partition must also store tuple information for reconstructing the original record.

A more common approach is *horizontal partitioning* that splits a table's tuples into disjoint subsets. Choose column(s) that divides the database equally in terms of size, load or usage. These keys are called the *partitioning key(s)*. The DBMS can partition a database physically (shared nothing) or logically (shared disk) based on hashing, data ranges or predicates. See Figure 3 for an example.

- **Logical Partitioning:** A node is responsible for a set of keys, but it doesn't actually store those keys. This is commonly used in a shared disk architecture.
- **Physical Partitioning:** A node is responsible for a set of keys, and it physically stores those keys. This is commonly used in a shared nothing architecture.

Handling Cluster Size Changes

The problem of hash partitioning is that when a node is added or removed, a lot of data has to be shuffled around. The solution is to use a hashing scheme that can add/remove paritions without having to reorganize the entire database.

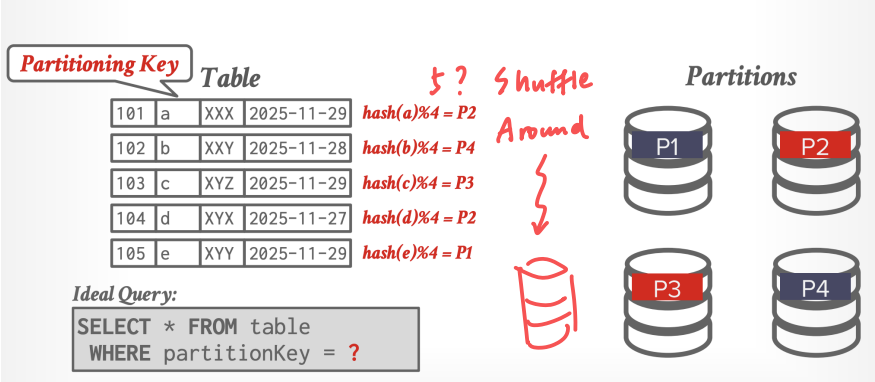

Figure 3: Horizontal Table Partitioning – Use hash partitioning to decide where to send the data. When the DBMS receives a query, it will use the table's partitioning key(s) to find out where the data is.

Consistent Hashing: This approach assigns every node to a location on some logical ring. Then the hash of every partition key maps to a location on the ring. The node that is closest to the key in the clockwise direction is responsible for that key. See Figure 4 for an example. When a node is added or removed, keys are only moved between nodes adjacent to the new/removed node and so only $1/n$ fraction of the keys are moved. A replication factor of k means that each key is replicated at the k closest nodes in the clockwise direction.

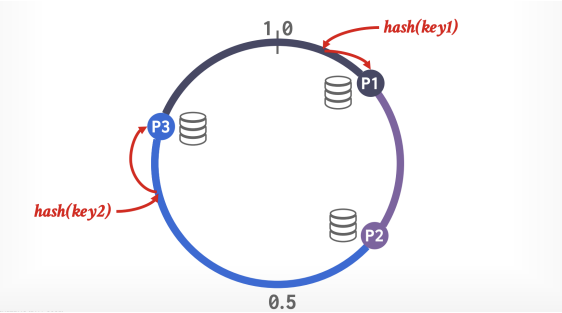

Figure 4: Consistent Hashing – All nodes are responsible for some portion of hash ring. Here, node $P1$ is responsible for storing $key1$ and node $P3$ is responsible for storing $key2$.

Rendezvous Hashing: Rendezvous Hashing (also called Highest-Random Weight hashing) assigns each key to a partition by computing a hash score for every partition and choosing the one with the highest score. For a key k and partition identifier p, the DBMS computes a hash value over the concatenation $h(k\|p)$; the partition whose score ranks highest *wins* the key. This avoids the ring structure required by Consistent Hashing and provides stable mappings: when nodes join or leave, only keys whose top-ranked partition changed must be moved.

(Consistent Hashing can be viewed as a specialized optimization of Rendezvous Hashing.)

5 Distributed Concurrency Control

A *single-node transaction* accesses data that is contained in one partition and does not require inter-node coordination. In contrast, a *distributed transaction* accesses data at one or more partitions, which requires

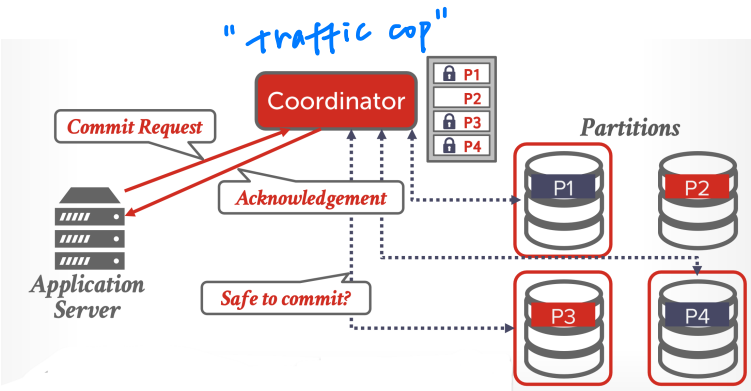

Figure 5: Centralized Coordinator – The client communicates with the coordinator to acquire locks on the partitions that the client wants to access. Once it receives an acknowledgement from the coordinator, the client sends its queries to those partitions./Once all queries for a given transaction are done, the client sends a commit request to the coordinator. The coordinator then communicates with the partitions involved in the transaction to determine whether the transaction is allowed to commit.

expensive coordination. There are two approaches for the coordination: *centralized* and *decentralized*.

Centralized coordinator

The centralized coordinator acts as a global "traffic cop" that coordinates all the behavior. See Figure 5 for a diagram.

Centralized coordinators can be implemented as a *middleware*, which accepts query requests and routes queries to correct partitions.

Decentralized coordinator

In a decentralized approach, nodes organize themselves. The client directly sends queries to one of the nodes. This *leader node* will send results back to the client. The leader node is in charge of communicating with other partitions in other nodes and ensuring correct commit behavior.

6 Federated Databases

These are distributed architectures that connect together multiple DBMSs into a single logical system. This is more popular in bigger companies. A query can access data at any location. This is hard due to different data models, query languages, and limitations of each individual DBMS. Additionally, there is no easy way to optimize queries. Lastly, there is a lot of data copying involved.

For example, say there is an application server which makes some queries. These queries then go through a middleware layer (which will convert the query into a readable format for a given DBMS used in the bigger system) that via *connectors*, will go through the multiple back-end DBMSs that are deployed in the system. The middleware will then handle the results returned from the DBMSs.

Lecture #24: Distributed Database Systems II

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 OLTP vs. OLAP

On-line Transaction Processing (OLTP)

- Short-lived read/write transactions.
- Small footprint.
- Repetitive operations.

simple query & write heavy

On-line Analytical Processing (OLAP)

- Long-running, read-only queries.
- Complex joins.
- Exploratory queries.

complex query & read heavy

2 Atomic Commit Protocols

When a multi-node transaction finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit. Depending on the protocol, a majority of the nodes or all of the nodes may be needed to commit. Examples include:

- Two-Phase Commit (1970s)
- Three-Phase Commit (1983)
- Viewstamped Replication (1988)
- Paxos (1989)
- ZAB (2008? Zookeeper Atomic Broadcast protocol, **Apache Zookeeper**)
- Raft (2013)

All of these atomic commit protocols have a common structure. They usually have a notion of **Resource Managers (RMs)** that manage resources or databases (or part of a database) on different nodes. The RMs need to coordinate together to decide the fate of each transaction: *Commit* or *Abort*. Using the RMs, an atomic commit protocol needs to guarantee the following properties:

- **Stability:** Once the fate of a transaction is decided, it cannot be changed.
- **Consistency:** All the RMs end up in the same state, even after failures.
- **Liveness:** The protocol must always have some way of progressing forward (e.g., enough nodes are alive and connected for the duration of the protocol).

In the following sections, we will focus on Two-Phase Commit and Paxos. If the coordinator fails after the prepare message is sent, Two-Phase Commit (2PC) blocks until the coordinator recovers. On the other hand, Paxos is non-blocking if a majority of participants are alive, provided that there is a sufficiently long period without further failures. If the nodes are in the same data center, do not fail often, and are not malicious, then 2PC is often preferred over Paxos as 2PC usually results in fewer round trips.

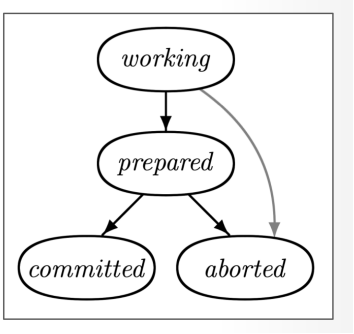

Figure 1: Atomic commit protocols can usually be modeled as state machines.

Two-Phase Commit

The client sends a *Commit Request* to the coordinator. In the first phase of this protocol, the coordinator sends a *Prepare* message, essentially asking the participant nodes if the current transaction is allowed to commit. If a given participant verifies that the given transaction is valid, they send an *OK* to the coordinator. If the coordinator receives an *OK* from all the participants, the system can now go into the second phase in the protocol. If anyone sends an *Abort* to the coordinator, the coordinator sends an *Abort* to the client.

The coordinator sends a *Commit* to all the participants, telling those nodes to commit the transaction if all the participants have sent an *OK*. Once the participants respond with an *OK*, the coordinator can tell the client that the transaction is committed. If the transaction was aborted in the first phase, the participants receive an *Abort* from the coordinator, to which they should respond with an *OK*. Either everyone commits or no one does. The coordinator can also be a participant in the system.

Additionally, in the case of a crash, all nodes keep track of a non-volatile log of the outcomes of each phase. Two-Phase Commit is a **blocking protocol**, which means nodes block until they can figure out the next course of action. If the coordinator crashes, the participants must decide what to do. A safe option is just to abort. Alternatively, the nodes can communicate with each other to see if they can commit without the explicit permission of the coordinator. If a participant crashes, the coordinator assumes that it responded with an abort if it has not sent an acknowledgement yet. timeout

Optimizations:

- *Early Prepare Voting* – If the DBMS sends a query to a remote node that it knows will be the last one executed there, then that node will also return their vote for the prepare phase with the query result.
- *Early Acknowledgement after Prepare* – If all nodes vote to commit a transaction, the coordinator can send the client an acknowledgement that their transaction was successful before the commit phase finishes.

Paxos

Paxos (along with Raft) is more prevalent in modern systems than 2PC. 2PC is a degenerate case of Paxos; Paxos uses $2F+1$ coordinators and makes progress as long as at least $F+1$ of them are working properly, 2PC sets $F = 0$.

Paxos is a consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed. This protocol does not block if a majority of participants are available and has provably minimal message delays in the best case. For Paxos, the coordinator is called the **proposer** and participants are called **acceptors**.

The client will send a *Commit Request* to the proposer. The proposer will send a *Propose* to the other nodes in the system, or the acceptors. A given acceptor will send an *Agree* if they have not already sent an *Agree* on a higher logical timestamp. Otherwise, they send a *Reject*.

Once the majority of the acceptors have sent an *Agree*, the proposer will send a *Commit*. The proposer must wait to receive an *Accept* from a majority of acceptors before sending the final message to the client saying that the transaction is committed, unlike 2PC.

Use exponential backoff times when trying to propose again after a failed proposal, to avoid dueling proposers. 1 2 4 8

Multi-Paxos: If the system elects a single leader that oversees proposing changes for some period, then it can skip the propose phase. The system periodically renews who the leader is using another Paxos round. When there is a failure, the DBMS can fall back to full Paxos.

3 CAP Theorem

The *CAP Theorem*, proposed by Eric Brewer and later proved in 2002 at MIT, explained that it is impossible for a distributed system to always be Consistent, Available, and Partition Tolerant. Only two of these three properties can be chosen.

Consistency is synonymous with linearizability for operations on all nodes. Once a write completes, all future reads should return the value of that write applied or a later write applied. Additionally, once a read has been returned, future reads should return that value or the value of a later applied write. NoSQL systems compromise this property in favor of the latter two. Other systems will favor this property and one of the latter two.

AP CP

Availability is the concept that all nodes that are up can satisfy all requests.

Partition tolerance means that the system can still operate correctly despite some message loss between nodes that are trying to reach consensus on values. If consistency and partition tolerance are chosen for a system, updates will not be allowed until a majority of nodes are reconnected, typically done in traditional or NewSQL DBMSs.

There is a modern version that considers consistency vs. latency trade-offs: *PACELC Theorem*. In case of network partitioning (P) in a distributed system, one has to choose between availability (A) and consistency (C), else (E), even when the system runs normally without network partitions, one has to choose between latency (L) and consistency (C).

4 Distributed Join Algorithms

For analytical workloads, the majority of time is spent computing joins and reading from disk, so optimizing joins is important. The efficiency of a distributed join depends on the target tables' partitioning schemes.

OLAP

One approach is to put entire tables on a single node and then perform the join there. However, the DBMS loses the parallelism of a distributed DBMS, defeating the purpose of making it distributed. This option also entails costly data transfer over the network.

To join tables *R* and *S*, the DBMS needs to get the proper tuples on the same node. Once there, it executes the same join algorithms discussed earlier in the semester. One should always send the minimal amount needed to compute the join.

There are four scenarios for distributed join algorithms.

Scenario 1

One of the tables is replicated at every node and the other table is partitioned across nodes. Each node joins its local data in parallel and then sends its results to a coordinating node.

Scenario 2

Both tables are partitioned on the join attribute, with IDs matching on each node. Each node performs the join on local data and then sends them to a node for coalescing.

Scenario 3

Both tables are partitioned on different keys. If one of the tables is small, then the DBMS broadcasts that table to all nodes. This takes us back to Scenario 1. Local joins are computed and then those joins are sent to a common node to perform the final join. This is known as a *broadcast join*.

Scenario 4

Worst !

This is the worst-case scenario. Both tables are not partitioned on the join key. The DBMS partitions the tables across nodes via the shuffle operator such that nodes have data matching on the join ID. This recovers Scenario 2. Local joins are computed and then the results are sent to a common node for the final join. If there isn't enough disk space, a failure is unavoidable. This is called a *shuffle join*.

Semi-Join Optimization

Before pulling data from another node, a *semi-join filter* can be used to reduce data movement. This filter blocks rows from the data that will be discarded in the result anyway due to a predicate. An approximate filter like a Bloom filter can be used here.

WHERE ···

Shuffle Operation

In addition to being useful in joins, a shuffle operation is generally useful. It can be used to rebalance data based on observed characteristics, and adjust the capacity allocated to a subcomputation. The shuffle operator is basically the repartition type of exchange operator discussed previously in lecture.

Exchange Type #3 – Repartition

→ Shuffle multiple input streams across multiple output streams.

→ Some DBMSs always perform this step after every pipeline (e.g., Google BigQuery).