Lecture #22: Introduction to Distributed Databases

15-445/645 Database Systems (Spring 2025)
https://15445.courses.cs.cmu.edu/spring2025/
Carnegie Mellon University
Jignesh Patel

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.

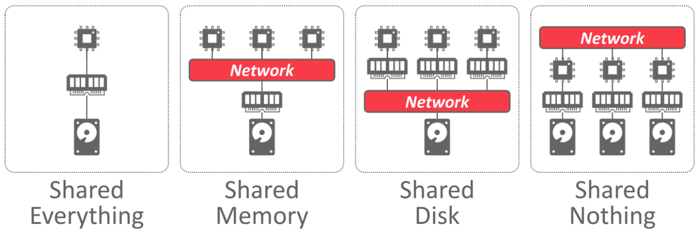


Figure 1: Database System Architectures – Four system architecture approaches ranging from sharing everything (used by non distributed systems) to sharing memory, disk, or nothing.

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.

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.

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?

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.

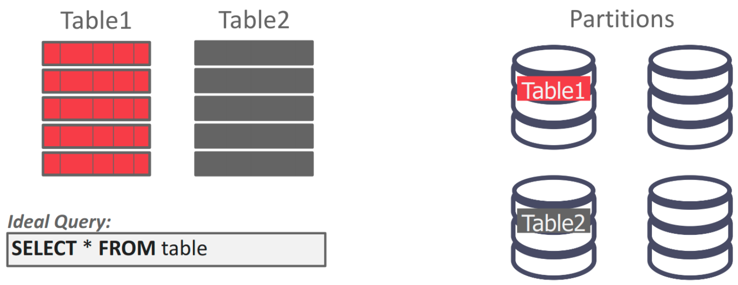


Figure 2: Naive Table Partitioning – Given two tables, place all the tuples in table one into one partition and the tuples in table two into the other.

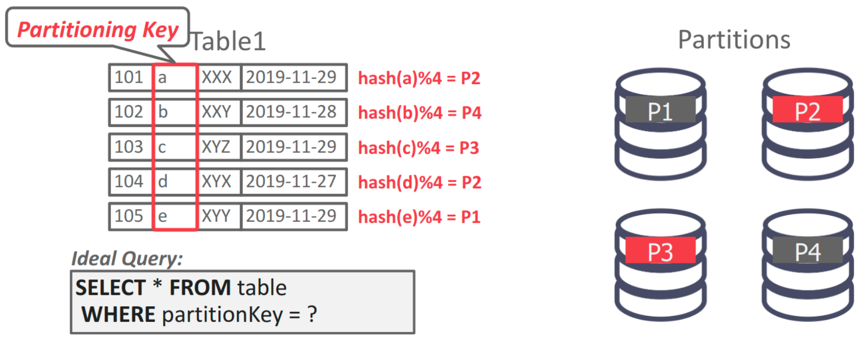
Another way of partitioning is *vertical partitioning*, which splits a table's attributes into separate partitions. Each partition must also store tuple information for reconstructing the original record.

More commonly, *horizontal partitioning* 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.

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 for this is *Consistent Hashing*.



101	a	XXX	2019-11-29	hash(a)%4 = P2
102	b	XXY	2019-11-28	hash(b)%4 = P4
103	c	XYZ	2019-11-29	hash(c)%4 = P3
104	d	XYX	2019-11-27	hash(d)%4 = P2
105	e	XYY	2019-11-29	hash(e)%4 = P1

```
SELECT * FROM table
 WHERE partitionKey = ?
```

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 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.

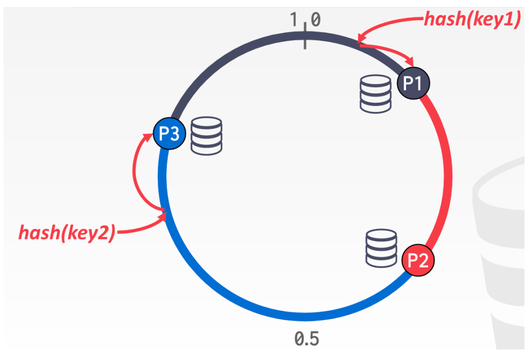


Figure 4: Consistent Hashing – All nodes are responsible for some portion of hash ring. Here, node $P1$ is responsible for storing $key1$ and node $P3$ is responsible for storing $key2$.

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 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.

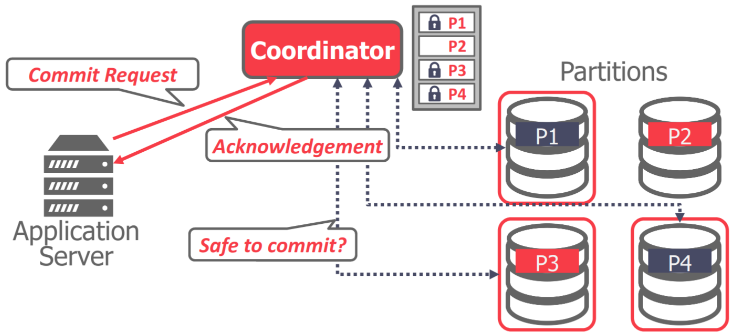


Figure 5: Centralized Coordinator – The client communicates with the coordinator to acquire locks on the partitions that the client wants to access. Once it receives an acknowledgement from the coordinator, the client sends its queries to those partitions. Once all queries for a given transaction are 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.

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.