Lecture #22: Introduction to Distributed Databases

15-445/645 Database Systems (Fall 2022)
https://15445.courses.cs.cmu.edu/fall2022/
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. An important goal in designing a distributed DBMS is fault tolerance (i.e., avoiding a single one node failure taking down the entire system).

Differences between **parallel** and **distributed** DBMSs:

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 (i.e., 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 address space and disk.

Shared Memory

An alternative to shared everything architecture in distributed systems is *shared memory*. CPUs have access to common memory address space via a fast interconnect. CPUs also share the same disk.

In practice, most DBMSs do not use this architecture, as it is provided at the OS / kernel level. It also causes problems, since each process's scope of memory is the same memory address space, which can be modified by multiple processes.

Each processor has a global view of all the in-memory data structures. Each DBMS instance on a processor has to "know" about the other instances.

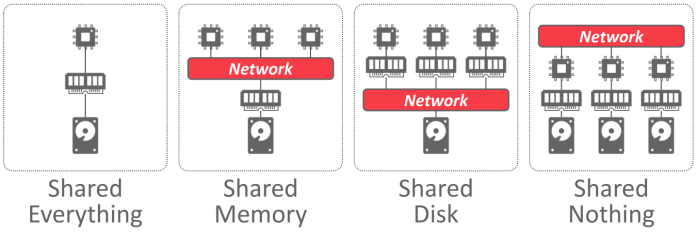

Figure 1: Database System Architectures – Four system architecture approaches ranging from sharing everything (used by non distributed systems) to sharing memory, disk, or nothing.

Shared Disk

In a *shared disk* architecture, all CPUs can read and write to a single logical disk directly via an interconnect, but each have their own private memories. The local storage on each compute node can act as caches. This approach is more common in cloud-based DBMSs.

The DBMS's execution layer can scale independently from the storage layer. Adding new storage nodes or execution nodes does not affect the layout or location of data 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.

Nodes have their own buffer pool and are considered stateless. A node crash does not affect the state of the database since that is stored separately on the shared disk. The storage layer persists the state in the case of crashes.

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 used to be considered the correct way to build distributed DBMSs.

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.

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?
- 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 does the DBMS ensure correctness?

Another design decision to make involves deciding how the nodes will interact in their clusters. Two options are *homogeneous* and *heterogeneous* nodes, which are both used in modern-day systems.

Homogeneous Nodes: Every node in the cluster can perform the same set of tasks (albeit on potentially different partitions of data), lending itself well to a shared nothing architecture. This makes provisioning and failover "easier". Failed tasks are assigned to available nodes.

Heterogeneous Nodes: Nodes are assigned specific tasks, so communication must happen between nodes to carry out a given task. Can allow a single physical node to host multiple "virtual" node types for dedicated tasks. Can independently scale from one node to other. An example is MongoDB, which has router nodes routing queries to shards and config server nodes storing the mapping from keys to shards.

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.

For *logically partitioned nodes*, particular nodes are in charge of accessing specific tuples from a shared disk. For *physically partitioned nodes*, each shared nothing node reads and updates tuples it contains on its own local disk.

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. This can be bad, since it is not scalable. One partition's resources can be exhausted if that one table is queried on often, not using all nodes available. 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.

More commonly used is *horizontal partitioning*, which splits a table's tuples into disjoint subsets. Choose column(s) that divides the database equally in terms of size, load, or usage, called the *partitioning key(s)*.

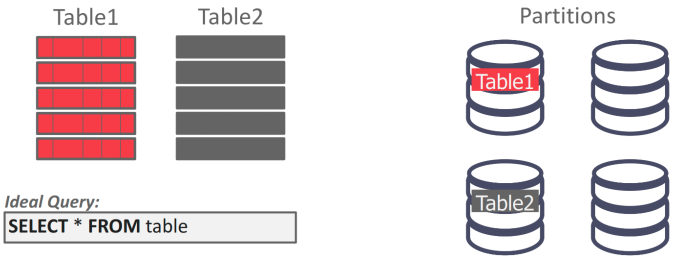

Figure 2: Naive Table Partitioning – Given two tables, place all the tuples in table one into one partition and the tuples in table two into the other.

The DBMS can partition a database physically (shared nothing) or logically (shared disk) via hash partitioning or range partitioning. See Figure 3 for an example. The problem of hash partitioning is that when a new node is added or removed, a lot of data needs to be shuffled around. The solution for this is *Consistent Hashing*.

Consistent Hashing assigns every node to a location on some logical ring. Then the hash of every partition key maps to some 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. A replication factor of n means that each key is replicated at the n closest nodes in the clockwise direction.

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.

5 Distributed Concurrency Control

A distributed transaction accesses data at one or more partitions, which requires expensive coordination.

Centralized coordinator

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

Middleware

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

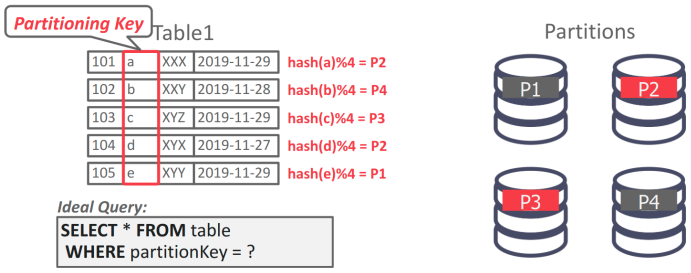

Figure 3: Horizontal Table Partitioning – Use hash partitioning to decide where to send the data. When the DBMS receives a query, it will use the table's partitioning key(s) to find out where the data is.

Decentralized coordinator

In a decentralized approach, nodes organize themselves. The client directly sends queries to one of the partitions. This *home partition* will send results back to the client. The home partition is in charge of communicating with other partitions and committing accordingly.

Centralized approaches give way to a bottleneck in the case that multiple clients are trying to acquire locks on the same partitions. It can be better for distributed 2PL as it has a central view of the locks and can handle deadlocks more quickly. This is non-trivial with decentralized approaches.

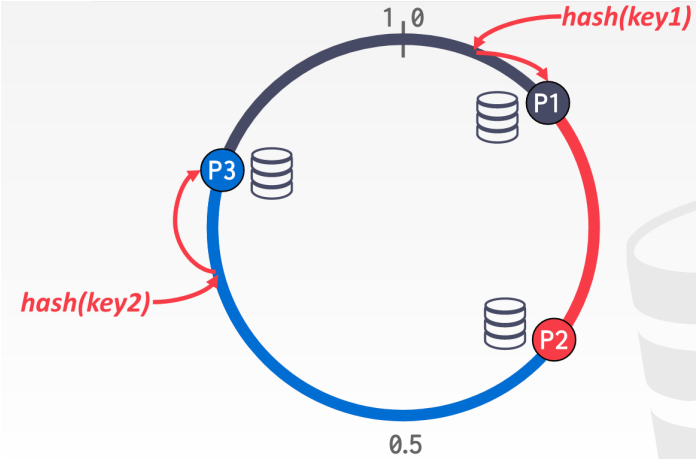

Figure 4: Consistent Hashing – All nodes are responsible for some portion of hash ring. Here node P1 is responsible for storing key1 and node P3 is responsible for storing key2.

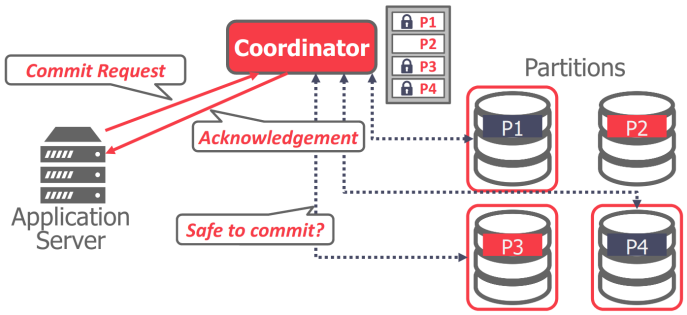

Figure 5: Centralized Coordinator – The client communicates with the coordinator to acquire locks on the partitions that the client wants to access. Once it receives an acknowledgement from the coordinator, the client sends its queries to those partitions. Once all queries for a given transaction are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