Lecture #22: Introduction to Distributed Databases

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Distributed DBMSs

One can use the building blocks from single-node DBMSs to support transaction processing and query execution in distributed environments. An important goal in designing a distributed DBMS is to make it fault tolerant (i.e., to avoid 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

CPUs have access to common memory address space via a fast interconnect. CPUs also share the same disk. In practice, no one does this, as this is provided at the OS / kernel level. It causes problems, since each process's scope of memory is the same memory address space, which could 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.

Shared Disk

All CPUs can read and write to a single logical disk directly via an interconnect, but each have their own private memories. 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

Each node has its own CPU, memory, and disk. Nodes only communicate with each other via network.

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

Some design questions to consider that will be covered more in-depth in future lectures:

- 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 between **homogeneous** and **heterogeneous** nodes, both used in modern-day systems:

Homogeneous: 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 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.

4 Partitioning Schemes

Split database across multiple resources, including disks, nodes, processors. This process is sometimes called *sharding* in NoSQL systems. The DBMS executes query fragments on each partition and then combines the results to produce a single answer. Users should not be required to know where data is physically located and how tables are partitioned or replicated. A SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

We want to pick a partitioning scheme that maximizes 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

Naive Table Partitioning: Each node stores one table, assuming enough storage space for a given node. This is each to implement as 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.

Horizontal Partitioning: Split 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)*. The DBMS can partition a database physical (shared nothing) or logically (shared disk) via hash partitioning or range partitioning.

5 Distributed Concurrency Control

If the DBMS supports multi-operation and distributed transactions, we need a way to coordinate their execution in the system.

Coordinator

This is a centralized approach with a global "traffic cop" that coordinates all the behavior. 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.

Middleware

This is the same as the centralized coordinator except that all queries are sent to a middleware directly layer.

Decentralized

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.