Today’s Class

• High-level overview of distributed DBMSs.
• Not meant to be a detailed examination of all aspects of these systems.

Why Do We Need Parallel/Distributed DBMSs?

• PayPal in 2008…
• Single, monolithic Oracle installation.
• Had to manually move data every xmas.
• Legal restrictions.
Why Do We Need Parallel/Distributed DBMSs?

- Increased Performance.
- Increased Availability.
- Potentially Lower TCO.

Parallel/Distributed DBMS

- Database is spread out across multiple resources to improve parallelism.
- Appears as a single database instance to the application.
  - SQL query for a single-node DBMS should generate same result on a parallel or distributed DBMS.

Parallel vs. Distributed

- **Parallel DBMSs:**
  - Nodes are physically close to each other.
  - Nodes connected with high-speed LAN.
  - Communication cost is assumed to be small.
- **Distributed DBMSs:**
  - Nodes can be far from each other.
  - Nodes connected using public network.
  - Communication cost and problems cannot be ignored.

Database Architectures

- The goal is parallelize operations across multiple resources.
  - CPU
  - Memory
  - Network
  - Disk
Database Architectures

Shared Memory

- CPUs and disks have access to common memory via a fast interconnect.
  - Very efficient to send messages between processors.
  - Sometimes called “shared everything”
- Examples: All single-node DBMSs.

Shared Disk

- All CPUs can access all disks directly via an interconnect but each have their own private memories.
  - Easy fault tolerance.
  - Easy consistency since there is a single copy of DB.
- Examples: Oracle Exadata, ScaleDB.

Shared Nothing

- Each DBMS instance has its own CPU, memory, and disk.
- Nodes only communicate with each other via network.
  - Easy to increase capacity.
  - Hard to ensure consistency.
- Examples: Vertica, Parallel DB2, MongoDB.
Early Systems

- **MUFFIN** – UC Berkeley (1979)
- **SDD-1** – CCA (1980)
- **System R*** – IBM Research (1984)
- **Gamma** – Univ. of Wisconsin (1986)
- **NonStop SQL** – Tandem (1987)

Inter- vs. Intra-query Parallelism

- **Inter-Query**: Different queries or txns are executed concurrently.
  - Increases throughput & reduces latency.
  - Already discussed for shared-memory DBMSs.
- **Intra-Query**: Execute the operations of a single query in parallel.
  - Decreases latency for long-running queries.

Parallel/Distributed DBMSs

- **Advantages**:
  - Data sharing.
  - Reliability and availability.
  - Speed up of query processing.
- **Disadvantages**:
  - May increase processing overhead.
  - Harder to ensure ACID guarantees.
  - More database design issues.

Today’s Class

- Overview & Background
- Design Issues
- Distributed OLTP
- Distributed OLAP
Design Issues

• How do we store data across nodes?
• How does the application find data?
• How to execute queries on distributed data?
  – Push query to data.
  – Pull data to query.
• How does the DBMS ensure correctness?

Database Partitioning

• Split database across multiple resources:
  – Disks, nodes, processors.
  – Sometimes called “sharding”
• The DBMS executes query fragments on each partition and then combines the results to produce a single answer.

Naïve Table Partitioning

• Each node stores one and only table.
• Assumes that each node has enough storage space for a table.

Ideal Query:

```
SELECT * FROM table
```
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.
  - Each tuple contains all of its columns.
- Three main approaches:
  - Round-robin Partitioning.
  - Hash Partitioning.
  - Range Partitioning.

Vertical Partitioning

- Split the columns of tuples into fragments:
  - Each fragment contains all of the tuples’ values for column(s).
- Need to include primary key or unique record id with each partition to ensure that the original tuple can be reconstructed.
Replication

- **Partition Replication**: Store a copy of an entire partition in multiple locations.
  - Master – Slave Replication
- **Table Replication**: Store an entire copy of a table in each partition.
  - Usually small, read-only tables.
- The DBMS ensures that updates are propagated to all replicas in either case.

Data Transparency

- Users should not be required to know where data is physically located, how tables are partitioned or replicated.
- A SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

OLTP vs. OLAP

- **On-line Transaction Processing**: 
  - Short-lived txns.
  - Small footprint.
  - Repetitive operations.
- **On-line Analytical Processing**: 
  - Long running queries.
  - Complex joins.
  - Exploratory queries.
Workload Characterization

Complex

OLAP

Social Networks

Simple

OLTP

Writes

Reads

Operation Complexity

Workload Focus

Michael Stonebraker – “Ten Rules For Scalable Performance In Simple Operation’ Datastores”
http://cacm.acm.org/magazines/2011/6/108651

Today’s Class

• Overview & Background
• Design Issues
• Distributed OLTP
• Distributed OLAP

Distributed OLTP

• Execute txns on a distributed DBMS.
• Used for user-facing applications:
  – Example: Credit card processing.
• Key Challenges:
  – Consistency
  – Availability

Single-Node vs. Distributed Transactions

• Single-node txns do not require the DBMS to coordinate behavior between nodes.
• Distributed txns are any txn that involves more than one node.
  – Requires expensive coordination.
Transaction Coordination

• Assuming that our DBMS supports multi-operation txns, we need some way to coordinate their execution in the system.
• Two different approaches:
  – **Centralized**: Global “traffic cop”.
  – **Decentralized**: Nodes organize themselves.

TP Monitors

• Example of a centralized coordinator.
• Originally developed in the 1970-80s to provide txns between terminals + mainframe databases.
  – Examples: ATMs, Airline Reservations.
• Many DBMSs now support the same functionality internally.
Centralized Coordinator

Application Server

Middleware

Partitions

P1
P2
P3
P4
P5

Decentralized Coordinator

Application Server

Commit Request

Partitions

Safe to commit?

P1
P2
P3
P4
P5

Observation

• **Q:** How do we ensure that all nodes agree to commit a txn?
  – What happens if a node fails?
  – What happens if our messages show up late?

CAP Theorem

• Proposed by Eric Brewer that it is impossible for a distributed system to always be:
  – Consistent
  – Always Available
  – Network Partition Tolerant
• Proved in 2002.
CAP Theorem

- Consistency
- Availability
- Partition Tolerant

No Man’s Land

Linearizability

All up nodes can satisfy all requests.

Still operate correctly despite message loss.

CAP – Consistency

Application Server

Set $A=2, B=9$

A=2 B=9

Application Server

Node 1

Node 2

CAP – Availability

Application Server

Read $A$

A=1 B=8

A=3 B=6

A=2 B=9

Application Server

Node 1

Node 2

CAP – Partition Tolerance

Application Server

Set $A=2, B=9$

A=2 B=9

Application Server

Master

Set $A=3, B=6$

A=3 B=6

Application Server

Master
**CAP Theorem**

- **Relational DBMSs**: CA/CP
  - Examples: IBM DB2, MySQL Cluster, VoltDB
- **NoSQL DBMSs**: AP
  - Examples: Cassandra, Riak, DynamoDB

These are essentially the same!

**Atomic Commit Protocol**

- When a multi-node txn finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit.
  - All nodes must agree on the outcome
- **Examples**:
  - Two-Phase Commit
  - Three-Phase Commit
  - Paxos

---

**Two-Phase Commit**

- **Phase1: Prepare**
- **Phase2: Commit**

**Node 1**
- Coordinator
- **Commit Request**
- **Phase1: Prepare**
- **Node 2**
- **OK**
- **Node 3**
- **OK**
- **Application Server**
- **Commit Request**

**Node 2**
- Participant
- **OK**

**Node 3**
- Participant
- **OK**

**Node 1**
- Coordinator
- **OK**

**Node 2**
- Participant
- **OK**

**Node 3**
- Participant
- **OK**

**Phase2: Commit**

- **ABORT**

---

**Two-Phase Commit**

- **Phase1: Prepare**
- **Phase2: Abort**

**Node 1**
- Coordinator
- **Commit Request**
- **Phase1: Prepare**
- **Node 2**
- **OK**
- **Node 3**
- **OK**

**Node 2**
- Participant
- **OK**

**Node 3**
- Participant
- **OK**

**Node 1**
- Coordinator
- **OK**

**Node 2**
- Participant
- **ABORT**

**Node 3**
- Participant
- **OK**
Two-Phase Commit

- Each node has to record the outcome of each phase in a stable storage log.
- **Q**: What happens if coordinator crashes?
  - Participants have to decide what to do.
- **Q**: What happens if participant crashes?
  - Coordinator assumes that it responded with an abort if it hasn’t sent an acknowledgement yet.
- The nodes have to block until they can figure out the correct action to take.

Three-Phase Commit

- The coordinator first tells other nodes that it intends to commit.
  - If the coordinator fails, then the participants elect a new coordinator and finish commit.
- Nodes do not have to block if there are no network partitions.

Paxos

- Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.
- Does not block if a majority of participants are available and has provably minimal message delays in the best case.

2PC vs. Paxos

- **Two-Phase Commit**: blocks if coordinator fails after the prepare message is sent, until coordinator recovers.
- **Paxos**: non-blocking as long as a majority participants are alive, provided there is a sufficiently long period without further failures.
Distributed Concurrency Control

- Need to allow multiple txns to execute simultaneously across multiple nodes.
  - Many of the same protocols from single-node DBMSs can be adapted.
- This is harder because of:
  - Replication.
  - Network Communication Overhead.
  - Node Failures.

Recovery

- **Q:** What do we do if a node crashes in CA/CP DBMS?
- If node is replicated, use Paxos to elect a new primary.
  - If node is last replica, halt the DBMS.
- Node can recover from checkpoints + logs and then catch up with primary.

Today’s Class

- Overview & Background
- Design Issues
- Distributed OLTP
- Distributed OLAP
Distributed OLAP

- Execute analytical queries that examine large portions of the database.
- Used for back-end data warehouses:
  - Example: Data mining
- Key Challenges:
  - Data movement.
  - Query planning.

Distributed Joins Are Hard

- Assume tables are horizontally partitioned:
  - Table1 Partition Key → table1.key
  - Table2 Partition Key → table2.key
- **Q:** How to execute?
- Naïve solution is to send all partitions to a single node and compute join.

Semi-Joins

- Main Idea: First distribute the join attributes between nodes and then recreate the full tuples in the final output.
  - Send just enough data from each table to compute which rows to include in output.
- Lots of choices make this problem hard:
  - What to materialize?
  - Which table to send?