Distributed OLTP Databases (Part I)
ADMINISTRIVIA

Project #3: TODAY @ 11:59am

Homework #5: Monday Dec 3rd @ 11:59pm

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 12th @11:59pm

Final Exam: Sunday Dec 16th @ 8:30am
Monday Dec 3\textsuperscript{rd} – VoltDB Lecture
→ Dr. Ethan Zhang (Lead Engineer)

Wednesday Dec 5\textsuperscript{th} – Potpourri + Review
→ Vote for what system you want me to talk about.
→ https://cmudb.io/f18-systems

Wednesday Dec 5\textsuperscript{th} – Extra Credit Check
→ Submit your extra credit assignment early to get feedback from me.
UPCOMING DATABASE EVENTS

**Swarm64 Tech Talk**
→ Thursday November 29th @ 12pm
→ GHC 8102 ← Different Location!

**VoltDB Research Talk**
→ Monday December 3rd @ 4:30pm
→ GHC 8102
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.
DISTRIBUTED DBMSs

Use the building blocks that we covered in single-node DBMSs to now support transaction processing and query execution in distributed environments.

→ Optimization & Planning
→ Concurrency Control
→ Logging & Recovery
**OLTP VS. OLAP**

**On-line Transaction Processing (OLTP):**
- Short-lived read/write txns.
- Small footprint.
- Repetitive operations.

**On-line Analytical Processing (OLAP):**
- Long-running, read-only queries.
- Complex joins.
- Exploratory queries.
TODAY'S AGENDA

System Architectures
Design Issues
Partitioning Schemes
Distributed Concurrency Control
A DBMS's system architecture specifies what shared resources are directly accessible to CPUs.

This affects how CPUs coordinate with each other and where they retrieve/store objects in the database.
SYSTEM ARCHITECTURE

- Shared Everything
- Shared Memory
- Shared Disk
- Shared Nothing
CPUs have access to common memory address space via a fast interconnect.

→ 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.
All CPUs can access a single logical disk directly via an interconnect but each have their own private memories.

→ Can scale execution layer independently from the storage layer.
→ Have to send messages between CPUs to learn about their current state.
SHARED DISK EXAMPLE

Application Server

Node

Get Id=101

Page ABC

Storage

Node

Page ABC

Node

Page ABC
SHARED DISK EXAMPLE

Application Server

Get Id=200

Page XYZ

Node

Node

Storage
SHARED DISK EXAMPLE

Application Server

Node

Get Id = 101

Node

Page ABC

Node

Storage

Page ABC
SHARED DISK EXAMPLE
SHARED DISK EXAMPLE

Application Server

Node

Update 101

Node

Page ABC

Node

Storage

Page ABC

Node

Application Server

Storage
SHARED DISK EXAMPLE

Application Server

Update 101

Node

Page ABC

Node

Node

Storage

Page ABC

Node

Node

Update 101
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.
SHARED NOTHING EXAMPLE

Get Id=200

P1→ID: 1–150

P2→ID: 151–300
Get Id=10
Get Id=200

Get Id=200

P1→ID: 1-150

P2→ID: 151-300
SHARED NOTHING EXAMPLE

Node

P1→ ID: 1–150

Node

P2→ ID: 151–300

Node

Application Server
SHARED NOTHING EXAMPLE

Node → P1 → ID: 1-100

Node → P2 → ID: 201-300

Node → P3 → ID: 101-200

Application Server
EARLY DISTRIBUTED DATABASE SYSTEMS

MUFFIN – UC Berkeley (1979)
SDD-1 – CCA (1979)
Gamma – Univ. of Wisconsin (1986)
NonStop SQL – Tandem (1987)
DESIGN ISSUES

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?
HOMOGENOUS VS. HETEROGENOUS

Approach #1: Homogenous Nodes
→ Every node in the cluster can perform the same set of tasks (albeit on potentially different partitions of data).
→ Makes provisioning and failover "easier".

Approach #2: Heterogenous Nodes
→ Nodes are assigned specific tasks.
→ Can allow a single physical node to host multiple "virtual" node types for dedicated tasks.
MONGODB CLUSTER ARCHITECTURE

- Config Server (mongod)
- Router (mongos)
- Shards (mongod)

Application Server

Get Id=101

P1

P2

P3

P4
**MongoDB Cluster Architecture**

- **Router (mongos)**
  - P1→ID: 1-100
  - P2→ID: 101-200
  - P3→ID: 201-300
  - P4→ID: 301-400

- **Config Server (mongod)**

- **Shards (mongod)**
  - P1
  - P2
  - P3
  - P4

- **Application Server**

- **Get Id=101**
MONGODB CLUSTER ARCHITECTURE

Router (mongos)

Shards (mongod)

P1→ ID:1-100
P2→ ID:101-200
P3→ ID:201-300
P4→ ID:301-400

Config Server (mongod)

Application Server

Get Id=101
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.
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.
NAÏVE TABLE PARTITIONING

Table 1

Table 2

Partitions

Ideal Query:

```
SELECT * FROM table
```
Naïve Table Partitioning

Table1  

Table2  

Partitions

Ideal Query:

SELECT * FROM table
NAÏVE TABLE PARTITIONING

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.
→ Hash Partitioning, Range Partitioning

The DBMS can partition a database physically (shared nothing) or logically (shared disk).
HORIZONTAL PARTITIONING

**Ideal Query:**

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

**Partitioning Key**

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>101 a XXX 2017-11-29</td>
<td>hash(a)%4 = P2</td>
</tr>
<tr>
<td>102 b XXX 2017-11-28</td>
<td>hash(b)%4 = P4</td>
</tr>
<tr>
<td>103 c XYZ 2017-11-29</td>
<td>hash(c)%4 = P3</td>
</tr>
<tr>
<td>104 d XYX 2017-11-27</td>
<td>hash(d)%4 = P2</td>
</tr>
<tr>
<td>105 e XYY 2017-11-29</td>
<td>hash(e)%4 = P1</td>
</tr>
</tbody>
</table>

**Ideal Query:**

```sql
SELECT * FROM table
WHERE partitionKey = ?
```
## Horizontal Partitioning

### Ideal Query:

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

### Table 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Key</th>
<th>Value</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
<td>XXX</td>
<td>2017-11-29</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2017-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2017-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2017-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2017-11-29</td>
</tr>
</tbody>
</table>

### Partitions

- P1
- P2
- P3
- P4

**Partitioning Key**

- hash(a)%4 = P2
- hash(b)%4 = P4
- hash(c)%4 = P3
- hash(d)%4 = P2
- hash(e)%4 = P1
LOGICAL PARTITIONING

Application Server

Node

Get Id=3

Storage

Node

Id=1
Id=2

Id=1
Id=2
Id=3
Id=4

Id=3
Id=4
PHYSICAL PARTITIONING

Application Server

Node

Node
PHYSICAL PARTITIONING

Node

Application Server

Id=1
Id=2

Id=3
Id=4
PHYSICAL PARTITIONING

Application Server

Get Id=1

Node

Id=1
Id=2

Node

Id=3
Id=4
PHYSICAL PARTITIONING

Application Server

Get Id=3

Node

Id=1
Id=2

Node

Id=3
Id=4
SINGLE-NODE VS. DISTRIBUTED

A **single-node** txn only accesses data that is contained on one partition.
→ The DBMS does not need coordinate the behavior concurrent txns running on other nodes.

A **distributed** txn accesses data at one or more partitions.
→ Requires expensive coordination.
If our DBMS supports multi-operation and distributed txns, we need a 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 and mainframe databases.
→ Examples: ATMs, Airline Reservations.

Many DBMSs now support the same functionality internally.
CENTRALIZED COORDINATOR
CENTRALIZED COORDINATOR

Application Server

Coordinator

Lock Request

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Coordinator

Lock Request

Acknowledgement

Application Server

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Application Server

Coordinator

Commit Request

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Commit Request

Coordinator

Safe to commit?

Application Server

Partitions

P1
P2
P3
P4
CENTRALIZED COORDINATOR

Coordinator

Commit Request

Acknowledgement

Safe to commit?

Application Server

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Application Server

Query Requests

Middleware

Partitions

P1

P2

P3

P4

P1→ID: 1-100
P2→ID: 101-200
P3→ID: 201-300
P4→ID: 301-400
CENTRALIZED COORDINATOR

Application Server

Query Requests

Middleware

Partitions

P1: ID: 1-100
P2: ID: 101-200
P3: ID: 201-300
P4: ID: 301-400
CENTRALIZED COORDINATOR

Middleware

Commit Request

Safe to commit?

Partitions

Application Server

P1→ID: 1-100
P2→ID: 101-200
P3→ID: 201-300
P4→ID: 301-400
DECENTRALIZED COORDINATOR

Application Server

Begin Request

Partitions

P1

P2

P3

P4

CMU 15-445/645 (Fall 2018)
DECENTRALIZED COORDINATOR

Application Server

Query Request

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Partitions

Commit Request

Safe to commit?
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.
→ Clock Skew.
DISTRIBUTED 2PL

Node 1

Set A=2

A=1

Node 2

Set B=7

B=8

Application Server

Application Server

NETWORK
DISTRIBUTED 2PL

Node 1

Set A=2

Node 2

Set B=7

Application Server

A=2

B=7

NETWORK

Application Server

Set A=2

Set B=7
DISTRIBUTED 2PL

Application Server

Set A=2
Set B=9

Node 1

A=2

Network

Node 2

B=7

Set B=7
Set A=0

Application Server
DISTRIBUTED 2PL

Waits-For Graph

Node 1

Set A = 2

Application Server

Set A = 0

Waits

Application Server

Set B = 7

Graph

T₁

T₂

= 7

= 0

Node 2

A = 2

B = 7

NETWORK
OBSERVATION

We have not discussed how to ensure that all nodes agree to commit a txn and then to make sure it does commit if we decide that it should.

→ What happens if a node fails?
→ What happens if our messages show up late?
ATOMIC COMMIT PROTOCOL

When a multi-nodetxn 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 (not used)
→ Paxos
→ Raft
→ ZAB (Apache Zookeeper)
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Application Server

Coordinator

Node 1

Node 2

Node 3
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Node 2

Node 3
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

OK

OK

Application Server

Coordinator

Node 1

Node 2

Node 3
**TWO-PHASE COMMIT (SUCCESS)**

1. **Phase 1: Prepare**
   - Application Server sends a **Commit Request** to the Coordinator.
   - Coordinator broadcasts the request to all participants.

2. **Phase 2: Commit**
   - Each participant, including Node 1, Node 2, and Node 3, responds with an **OK**.
   - Coordinator collects the responses and initiates the commit process.
TWO-PHASE COMMIT (SUCCESS)

Application Server

Commit Request

Phase1: Prepare

Coordinator

Phase2: Commit

Node 1

OK

OK

Node 2

OK

OK

Node 3

Participant

Participant
TWO-PHASE COMMIT (SUCCESS)
TWO-PHASE COMMIT (ABORT)

Commit Request

Application Server

Coordinator

Node 1

Participant

Node 2

 Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase 1: Prepare

Coordinator

Node 1

Node 2

Node 3

Application Server

Participant

Participant

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant

Aborted

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Node 1

Coordinator

Phase 2: Abort

Node 2

Participant

Aborted

Node 3

Participant

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Aborted

Phase2: Abort

Node 1

OK

Node 2

ABORT!

OK

Node 3

Participant

Participant

Coordinator
**2PC OPTIMIZATIONS**

**Early Prepare Voting**
→ If you send a query to a remote node that you know will be the last one you execute 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 txn, the coordinator can send the client an acknowledgement that their txn was successful before the commit phase finishes.
EARLY ACKNOWLEDGEMENT

- Application Server
  - Coordinator
    - Node 1
  - Participant
    - Node 2
    - Node 3

Commit Request
EARLY ACKNOWLEDGEMENT

Phase1: Prepare

Commit Request
EARLY ACKNOWLEDGEMENT

Phase1: Prepare

Commit Request

Node 1

Coordinator

Node 2

Node 3

Participant

Participant
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Phase 1: Prepare

Success!

OK

OK

Node 2

Node 3

Participant

Participant
EARLY ACKNOWLEDGEMENT

**Phase 1: Prepare**

**Phase 2: Commit**

Success!

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant
EARLY ACKNOWLEDGEMENT

Success!

Phase 1: Prepare

Phase 2: Commit

Coordinator

Node 1

OK

OK

OK

Node 2

OK

OK

OK

Node 3

Application Server

Participant

Participant
TWO-PHASE COMMIT

Each node has to record the outcome of each phase in a stable storage log.

**What happens if coordinator crashes?**
→ Participants have to decide what to do.

**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.
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.

→ First correct protocol that was provably resilient in the face asynchronous networks
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.
CONCLUSION

I have barely scratched the surface on distributed txn processing...

It is **really** hard to get right.

More info (and humiliation):
→ Kyle Kingsbury's Jepsen Project
NEX T CL ASS

Replication
CAP Theorem
Real-World Examples