ADMINISTRIVIA

Homework #5: Sunday Dec 6th @ 11:59pm

Project #4: Sunday Dec 13th @ 11:59pm

Potpourri + Review: Wednesday Dec 9th
→ Vote for what system you want me to talk about.
   https://cmudb.io/f20-systems

Final Exam:
→ Session #1: Thursday Dec 17th @ 8:30am
→ Session #2: Thursday Dec 17th @ 1:00pm
UPCOMING DATABASE TALKS

Confluent ksqlDB (Kafka) → Monday Nov 23\textsuperscript{rd} @ 5pm ET

Microsoft SQL Server Optimizer → Monday Nov 30\textsuperscript{th} @ 5pm ET

Snowflake Lecture → Monday Dec 7\textsuperscript{th} @ 3:20pm ET
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
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

Network
CPU's 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.
SHARED DISK

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.
→ Must send messages between CPUs to learn about their current state.
SHARED DISK EXAMPLE

Application Server

Get Id=101

Page ABC

Storage
SHARED DISK EXAMPLE

Application Server

Node

Storage

Get Id=200

Page XYZ
SHARED DISK EXAMPLE

Application Server

Node

Get Id=101

Page ABC

Storage
SHARED DISK EXAMPLE

Application Server

Node

Node

Node

Storage

Update 101

Page ABC
**SHARED DISK EXAMPLE**

Application Server

Node

Node

Node

Storage

Update 101

Page ABC
Each DBMS instance has its own CPU, memory, and disk.
Nodes only communicate with each other via network.
→ Harder to scale capacity.
→ Harder to ensure consistency.
→ Better performance & efficiency.
SHARED NOTHING EXAMPLE

Node

P1\rightarrow ID: 1-150

Node

P2\rightarrow ID: 151-300

Application Server

Get Id=200
**SHARED NOTHING EXAMPLE**

**Application Server**

Node 1
- ID: 1-150
- Get Id=10
- Get Id=200

Node 2
- ID: 151-300
- Get Id=200

Annotations:
- P1 → ID: 1-150
- P2 → ID: 151-300
SHARED NOTHING EXAMPLE

Node

P1→ID: 1-100

Node

P3→ID: 101-200

Node

P2→ID: 201-300
EARLY DISTRIBUTED DATABASE SYSTEMS

MUFFIN – UC Berkeley (1979)
SDD-1 – CCA (1979)
Gamma – Univ. of Wisconsin (1986)
NonStop SQL – Tandem (1987)
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.
MongodDB Heterogeneous 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 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.
→ Often called "sharding" in NoSQL systems.

The DBMS executes query fragments on each partition and then combines the results to produce a single answer.
NAÏVE TABLE PARTITIONING

Assign an entire table to a single node.
Assumes that each node has enough storage space for an entire table.

Ideal if queries never join data across tables stored on different nodes and access patterns are uniform.
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

Table1

Table2

Partitions

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.
→ Hash Partitioning, Range Partitioning

The DBMS can partition a database **physical** (shared nothing) or **logically** (shared disk).
### Ideal Query:

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

**Partitioning Key**

Table1

<table>
<thead>
<tr>
<th>101</th>
<th>a</th>
<th>XXX</th>
<th>2019-11-29</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2019-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2019-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2019-11-29</td>
</tr>
</tbody>
</table>

**Partitions**

- P1
- P2
- P3
- P4

**Ideal Query:**

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

**Partitioning Key**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Value</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
<td>XXX</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2019-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2019-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2019-11-29</td>
</tr>
</tbody>
</table>

Ideal Query:

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

**Partitioning Key**

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
<th>Partition</th>
<th>Dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
<td>XXX</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2019-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2019-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2019-11-29</td>
</tr>
</tbody>
</table>

**Ideal Query:**

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

Partitioning Key

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
<td>XXX</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2019-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2019-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2019-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2019-11-29</td>
</tr>
</tbody>
</table>

Ideal Query:

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

hash(key1)

P1

P2

P3
CONSISTENT HASHING

\[
\text{hash(key1)}
\]

\[
\text{hash(key2)}
\]
CONSISTENT HASHING

hash(key1)
CONSISTENT HASHING

If hash(key) = P4

P1
P3
P4
P2
CONSISTENT HASHING
CONSISTENT HASHING
CONSISTENT HASHING

Replication Factor = 3
CONSISTENT HASHING

Replication Factor = 3

hash(key1)

P1

P2

P3

P4

P5

P6

0.5
CONSISTENT HASHING

hash(key1)

Replication Factor = 3

P1 → P2 → P3 → P4 → P5 → P6

hash(key1) = 0.5

P1 → P2

P1 → P6

P1 → P5

P1 → P4

P1 → P3

P1 → P2 → P3

P1 → P2 → P4

P1 → P2 → P5

P1 → P2 → P6

P1 → P2 → P1

P1 → P3 → P1

P1 → P4 → P1

P1 → P5 → P1

P1 → P6 → P1

P1 → P1 → P2

P1 → P1 → P3

P1 → P1 → P4

P1 → P1 → P5

P1 → P1 → P6

P1 → P1 → P1
LOGICAL PARTITIONING

Application Server

Node

Node

Storage

Id=1
Id=2
Id=3
Id=4
LOGICAL PARTITIONING

Application Server

Node

Get Id=1

Id=1
Id=2

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

Node

Storage

Id=1
Id=2
Id=3
Id=4
LOGICAL PARTITIONING

Application Server

Node

Id=1
Id=2

Get Id=3

Node

Id=3
Id=4

Storage

Id=1
Id=2
Id=3
Id=4
PHYSICAL PARTITIONING

Application Server

Get Id=1

Node

Id=1
Id=2

Id=3
Id=4

Node
PHYSICAL PARTITIONING

Application Server

Get Id=3
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.
A **TP Monitor** is an example of a centralized coordinator for distributed DBMSs. 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

Coordinator

Lock Request

Application Server

Partitions

P1
P2
P3
P4

P1
P2
P3
P4
CENTRALIZED COORDINATOR

Application Server

Lock Request

Coordinator

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Coordinator

Lock Request

Acknowledgement

Application Server

Partitions

P1
P2
P3
P4

P1
P2
P3
P4
CENTRALIZED COORDINATOR

Commit Request

Application Server

Safe to commit?

Coordinator

Partitions

P1
P2
P3
P4

P1
P2
P3
P4
**Centralized Coordinator**

**Commit Request**

**Application Server**

- **Coordinator**

  **Acknowledgement**

  **Safe to commit?**

**Partitions**

- P1
- P2
- P3
- P4
CENTRALIZED COORDINATOR

Middleware

Query Requests

Application Server

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Query Requests

Application Server

Middleware

Partitions

P1

P2

P3

P4

P1→ID: 1-100

P2→ID: 101-200

P3→ID: 201-300

P4→ID: 301-400
CENTRALIZED COORDINATOR

Middleware

Query Requests

Application Server

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?

Application Server

Partitions

P1

ID: 1-100

P2

ID: 101-200

P3

ID: 201-300

P4

ID: 301-400

P1→ID: 1-100

P2→ID: 101-200

P3→ID: 201-300

P4→ID: 301-400

Safe to commit?
DECENTRALIZED COORDINATOR

Application Server

Begin Request

Master Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Query Request

Master Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Safe to commit?

Master Node

Partitions

P1

P2

P3

P4
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

Application Server

Set A=2

Network

Application Server

Set B=7

Node 1

A=1

Node 2

B=8
**DISTRIBUTED 2PL**

- **Set A=2**
  - Node 1: A=2

- **Set B=7**
  - Node 2: B=7

Application Server

---

Node 1

Node 2

NETWORK
DISTRIBUTED 2PL

Application Server

Set A=2
Set B=9

Node 1

A=2

Node 2

B=7

Application Server

Set B=7
Set A=0

NETWORK
DISTRIBUTED 2PL

Waits-For Graph

Application Server

Set A=2
Set T_1 T_2
Set B=7
Set A=0
Set B=8

Node 1

Node 2

Network

Application Server

A=2
B=7
CONCLUSION

I have barely scratched the surface on distributed database systems...

It is hard to get this right.
NEXT CLASS

Distributed OLTP Systems
Replication
CAP Theorem
Real-World Examples