Introduction to Distributed Databases
ADMINISTRIVIA

Project #4 is due Sun Dec 11th @ 11:59pm
→ Zoom Q&A Session TONIGHT @ 8:00pm

Homework #5 is due Sun Dec 4th @ 11:59pm
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 distributed 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 must "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.
→ Must send messages between CPUs to learn about their current state.
SHARED DISK EXAMPLE

Application Server

Get Id=101

Node

Page ABC

Node

Storage
SHARED DISK EXAMPLE

Application Server

Node

Get Id=102

Page XYZ

Node

Storage
SHARED DISK EXAMPLE

Application Server

Get Id=101

Page ABC

Storage
Application Server

Node

Node

Node

Storage

update 101

Page ABC
Each DBMS instance has its own CPU, memory, and local disk. Nodes only communicate with each other via network.

→ Harder to scale capacity.
→ Harder to ensure consistency.
→ Better performance & efficiency.
SHARED NOTHING EXAMPLE

Application Server

Get Id=200

Node

P1→ID: 1-150

Node

P2→ID: 151-300
SHARED NOTHING EXAMPLE

Application Server

Node

Get Id=100
Get Id=200

Node

Get Id=200

Id=200

P1→ID: 1-150

P2→ID: 151-300
**SHARE NOTHINN EXAMPLE**

- **Application Server**

- Diagram showing three nodes:
  - Node 1: P1→ID:1-100
  - Node 2: P2→ID:201-300
  - Node 3: P3→ID:101-200
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?
Where does the application send queries?
How to execute queries on distributed data?
  → Push query to data.
  → Pull data to query.

How does the DBMS ensure correctness?

How do we divide the database across resources?

Next Class
HOMOGENOUS VS. HETEROGENEOUS

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 HETEROGENEOUS ARCHITECTURE

Application Server

Get Id=101

Router (mongos)

Router (mongos)

Config Server (mongod)

Shards (mongod)

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

Application Server

Get Id=101

Router (mongos)

Router (mongos)

Config Server (mongod)

Shards (mongod)

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

CMU-DB
DATA TRANSPARENCY

Applications should not be required to know where data is physically located in a distributed DBMS.

→ Any query that run on a single-node DBMS should produce the same result on a distributed DBMS.

In practice, developers need to be aware of the communication costs of queries to avoid excessively "expensive" data movement.
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.

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

**Table1**

```
SELECT * FROM table
```

**Table2**

**Partitions**

Ideal Query:
**VERTICAL PARTITIONING**

Split a table's attributes into separate partitions.
Must store tuple information to reconstruct the original record.

```sql
CREATE TABLE foo (   attr1 INT,   attr2 INT,   attr3 INT,   attr4 TEXT );
```

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuple#2</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuple#3</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuple#4</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
<td></td>
</tr>
</tbody>
</table>
VERTICAL PARTITIONING

Split a table's attributes into separate partitions.
Must store tuple information to reconstruct the original record.

```
CREATE TABLE foo (  
    attr1 INT,  
    attr2 INT,  
    attr3 INT,  
    attr4 TEXT  
);  
```
HORIZONTAL PARTITIONING

Split a table's tuples into disjoint subsets based on some partitioning key and scheme.
→ Choose column(s) that divides the database equally in terms of size, load, or usage.

Partitioning Schemes:
→ Hashing
→ Ranges
→ Predicates
## HORIZONTAL PARTITIONING

### Table 1

<table>
<thead>
<tr>
<th>Partition Key</th>
<th>Value</th>
<th>Year</th>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
<td>XXX</td>
<td>2022-11-29</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2022-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2022-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2022-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2022-11-29</td>
</tr>
</tbody>
</table>

**Partitioning Key**

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

### Ideal Query:

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

Application Server

Storage

Node

Id=1
Id=2

Id=3
Id=4

Get Id=1
LOGICAL PARTITIONING

Application Server

Storage

Node

Get Id=3

Id=1

Id=2

Id=1

Id=2

Id=3

Id=4

Id=3

Id=4
LOGICAL PARTITIONING

Application Server

Node

Id=1
Id=2

Get Id=3
Get Id=2

Node

Id=3
Id=4

Get Id=2

Storage

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

Application Server

Get Id=1

Get Id=3

Get Id=3
# HORIZONTAL PARTITIONING

## Ideal Query:

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

<table>
<thead>
<tr>
<th>Partitioning Key</th>
<th>Table1</th>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a XXX 2022-11-29</td>
<td>P2</td>
</tr>
<tr>
<td>102</td>
<td>b XXY 2022-11-28</td>
<td>P4</td>
</tr>
<tr>
<td>103</td>
<td>c XYZ 2022-11-29</td>
<td>P3</td>
</tr>
<tr>
<td>104</td>
<td>d XXY 2022-11-27</td>
<td>P2</td>
</tr>
<tr>
<td>105</td>
<td>e XYY 2022-11-29</td>
<td>P1</td>
</tr>
</tbody>
</table>

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

**Table 1**

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>Partition Key</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
<td>XXX</td>
<td>2022-11-29</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
<td>XXY</td>
<td>2022-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XYZ</td>
<td>2022-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYX</td>
<td>2022-11-27</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
<td>XYY</td>
<td>2022-11-29</td>
</tr>
</tbody>
</table>

**Partitioning Key**

- $\text{hash}(a) \% 5 = P4$
- $\text{hash}(b) \% 5 = P3$
- $\text{hash}(c) \% 5 = P5$
- $\text{hash}(d) \% 5 = P1$
- $\text{hash}(e) \% 5 = P3$

**Ideal Query:**

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

hash(key1)

P1

P2

P3

0.5

1

0
CONSISTENT HASHING

hash(key1) → P1
hash(key2) → P3

0.5
**CONSISTENT HASHING**

- `hash(key1)`: A hash function assigning a key to a point on a circle.
- `hash(key2)`: Another hash function for comparison.
- **Points P1, P2, P3**: These points represent servers or nodes in a distributed system.
- **Circles and Arrows**: Indicate the movement of keys based on their hash values, showing how consistent hashing works to balance load.
If hash(key) = P4
CONSISTENT HASHING
CONSISTENT HASHING

Replication Factor = 3
Replication Factor = 3

hash(key1)
CONSISTENT HASHING

Replication Factor = 3

hash(key1)
SINGLE-NODE VS. DISTRIBUTED

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

A **distributed**txn accesses data at one or more partitions.
→ Requires expensive coordination.
TRANSACTION 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

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.

Standardized protocol from 1990s: **X/Open XA**
CENTRALIZED COORDINATOR

Application Server

Lock Request

Coordinator

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Coordinator

Lock Request

Application Server

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Application Server

Coordinator

Lock Request

Acknowledgement

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Commit Request

Safe to commit?

Application Server

Coordinator

Partitions

P1

P2

P3

P4
**CENTRALIZED COORDINATOR**

Commit Request

Coordinator

Acknowledgement

Safe to commit?

Application Server

Partitions

P1

P2

P3

P4
**Centralized Coordinator**

- **Commit Request** from Application Server
- **Coordinator**:
  - Acknowledgement
  - **Safe to commit?**
- **Partitions**:
  - P1
  - P2
  - P3
  - P4
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
P2
P3
P4
DECENTRALIZED COORDINATOR

Application Server

Begin Request

Leader Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Leader Node

Query Request

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Safe to commit?

Leader 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

Node 1

A=1

Node 2

Set B=7

Application Server

NETWORK

B=8
DISTRIBUTED 2PL

Application Server

Set A=2

Node 1

A=2

Network

Node 2

B=7

Application Server

Set B=7
DISTRIBUTED 2PL

Application Server  
Node 1  
Set A=2  
Set B=9  
A=2  
APPLICATION

NETWORK

Application Server  
Node 2  
Set B=7  
Set A=0  
B=7

CMU-DB 15-445/645 (Fall 2022)
DISTRIBUTED 2PL

Waits-For Graph

Application Server

Set A=2

T1

Set B=7

=7

T2

=0

Application Server

A=2

Node 1

= NETWORK =

B=7

Node 2
I have barely scratched the surface on distributed database systems...

It is **hard** to get this right.
PROJECT #4 – CONCURRENCY CONTROL

You will add support for concurrent transactions using two-phase locking in BusTub!
→ Deadlock Detection
→ Hierarchical Locking (Table, Tuple)
→ Multiple Isolation Levels
→ Aborts/Rollbacks

You do not need to worry about logging txns to disk.

https://15445.courses.cs.cmu.edu/fall2022/project4/

Prompt: A dramatic and vibrant painting of a giant eye in the clouds looking down on a field of grazing sheep with padlocks as their heads.
PROJECT #3 - TASKS

Lock Manager
→ Maintain internal lock table and queues.
→ Track the growing/shrinking phases of txns.
→ Notify waiting txns when their locks are available.

Deadlock Detector:
→ Build the waits-for graph and deterministically identify what txn to kill off to break deadlocks

Execution Engine
→ Modify Project #3 executors to support txn requests.
PROJECT #3 - LEADERBOARD

We have designed the Terrier benchmark to measure who has the fastest BusTub implementation!

Tasks:
→ UpdateExecutor
→ Predicate Pushdown
THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you pull in the latest changes from the BusTub main branch.

Post your questions on Piazza or come to TA office hours.

Compare against our [solution in your browser](#)!
PLAGIARISM WARNING

Your project implementation must be your own work.
→ You may not copy source code from other groups or the web.
→ Do not publish your implementation on Github.

Plagiarism will not be tolerated. See CMU's Policy on Academic Integrity for additional information.
NEXT CLASS

Distributed OLTP Systems
Replication
CAP Theorem
Real-World Examples