Lecture #21

Intro to Distributed Databases

FALL 2023  Prof. Andy Pavlo  •  Prof. Jignesh Patel
**ADMINISTRIVIA**

**Homework #5** is due Sunday Dec 3rd @ 11:59pm

**Project #4** is due Sunday Dec 10th @ 11:59pm

Upcoming Special Lectures:
→ **SingleStore** (Monday Dec 4th over Zoom)
→ **Systems Speedrun Lecture** (Wednesday Dec 6th)

**Final Exam** is Tuesday Dec 12th @ 8:30am.
UPCOMING EVENTS

**pgVector** (ML↔DB Seminar)
→ Monday November 20\(^{th}\) @ 4:30pm

**Chroma** (ML↔DB Seminar)
→ Monday November 27\(^{th}\) @ 4:30pm
Databases are hard.
Distributed databases are harder.
Databases are hard.
Distributed databases are harder.
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 Nothing
- Shared Disk
- Shared Memory
Each DBMS node has its own CPU, memory, and local disk.
Nodes only communicate with each other via network.
→ Better performance & efficiency.
→ Harder to scale capacity.
→ Harder to ensure consistency.
SHARED NOTHING EXAMPLE

Catalog Meta-Data

Application Server

Node

P1→ID: 1–150

Node

P2→ID: 151–300
**SHARED NOTHING EXAMPLE**

Application Server

Catalog Meta-Data

Node

Get Id=200

Node

P1→ID: 1-150

P2→ID: 151-300
**SHARED NOTHING EXAMPLE**

Application Server

Catalog

Data

Get Id=100
Get Id=200

Node

P1→ID: 1-150

Get Id=200

Id=200

Node

P2→ID: 151-300

Get Id=200
SHARED NOTHING EXAMPLE

Catalog Meta-Data

Application Server

Node

P1→ID: 1-150

Node

P2→ID: 151-300
SHARED NOTHING EXAMPLE

Catalog Meta-Data

Application Server

Node  
P1→ID: 1-100

Node  
P3→ID: 101-200

Node  
P2→ID: 201-300
Nodes access a single logical disk via an interconnect, but each have their own private memories.

→ Scale execution layer independently from the storage layer.
→ Nodes can still use direct attached storage as a slower/larger cache.
→ This architecture facilitates data lakes and serverless systems.
SHARED DISK EXAMPLE

Catalog Meta-Data

Node

Storage

Page ABC

Get Id=101

Application Server
**SHARED DISK EXAMPLE**

- **Catalog Meta-Data**
- **Node**
- **Storage**

- Application Server
- Node
- Page XYZ
- Get Id=102
- Page XYZ
SHARED DISK EXAMPLE

Catalog Meta-Data

Application Server

Get Id=101

Node

Page ABC

Node

Node

Storage
**SHARED DISK EXAMPLE**

- **Catalog Meta-Data**
- **Node**
- **Node**
- **Node**
- **Node**
- **Storage**

*Update 101*

*Page ABC*

Application Server
Nodes access a common memory address space via a fast interconnect. → Each node has a global view of all the in-memory data structures. → Can still use local memory / disk for intermediate results.

This looks a lot like shared-everything. Nobody does this.
EARLY DISTRIBUTED DATABASE SYSTEMS

- **MUFFIN** – UC Berkeley (1979)
- **SDD-1** – CCA (1979)
- **System R** – IBM Research (1984)
- **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.
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

Table 1

Table 2

Partitions

Ideal Query:

```
SELECT * FROM table1
```
NAÏVE TABLE PARTITIONING

Ideal Query:

SELECT * FROM table1
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  
);  
```
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 );

**Partition #1**

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple#2</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
</tr>
<tr>
<td>Tuple#3</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
</tr>
<tr>
<td>Tuple#4</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
</tr>
</tbody>
</table>

**Partition #2**

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple#2</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#4</td>
<td>attr4</td>
</tr>
</tbody>
</table>
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

Partitioning Key

**Table**

<table>
<thead>
<tr>
<th>101</th>
<th>a</th>
<th>XXX</th>
<th>2022-11-29</th>
</tr>
</thead>
<tbody>
<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>

**Partitions**

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

**Ideal Query:**

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

Partitioning Key

Table

<table>
<thead>
<tr>
<th>101</th>
<th>a</th>
<th>XXX</th>
<th>2022-11-29</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>b</td>
<td>XXX</td>
<td>2022-11-28</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
<td>XXX</td>
<td>2022-11-29</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
<td>XYY</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>

Partitions

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

Ideal Query:

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

**Partitioning Key**

<table>
<thead>
<tr>
<th>Table</th>
<th>Partition Key</th>
<th>Hash Modulo 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>101 a</td>
<td>XXX</td>
<td>P2</td>
</tr>
<tr>
<td>102 b</td>
<td>XXY</td>
<td>P4</td>
</tr>
<tr>
<td>103 c</td>
<td>XYZ</td>
<td>P3</td>
</tr>
<tr>
<td>104 d</td>
<td>XXY</td>
<td>P2</td>
</tr>
<tr>
<td>105 e</td>
<td>XYY</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Ideal Query:**

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

### Ideal Query:

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

Application Server

Node

Storage

Get Id=1

Node

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

Node
Id=3
Id=4

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

Get Id=3
LOGICAL PARTITIONING

Application Server

Node

Storage

Get Id=3
Get Id=2

Id=1
Id=2

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

Id=3
Id=4
PHYSICAL PARTITIONING

Application Server

Get Id=1

Node

Id=1
Id=2

Id=3
Id=4
PHYSICAL PARTITIONING

Application Server

Node

Id=1
Id=2

Get Id=3

Node

Id=3
Id=4
**HORIZONTAL PARTITIONING**

**Partitioning Key**

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

**Ideal Query:**
```
SELECT * FROM table
WHERE partitionKey = ?
```
HORIZONTAL PARTITIONING

Partitioning Key

<table>
<thead>
<tr>
<th>Table</th>
<th>Partitioning Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>a</td>
</tr>
<tr>
<td>102</td>
<td>b</td>
</tr>
<tr>
<td>103</td>
<td>c</td>
</tr>
<tr>
<td>104</td>
<td>d</td>
</tr>
<tr>
<td>105</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>XXX</td>
</tr>
<tr>
<td>102</td>
<td>XXY</td>
</tr>
<tr>
<td>103</td>
<td>XYZ</td>
</tr>
<tr>
<td>104</td>
<td>XYX</td>
</tr>
<tr>
<td>105</td>
<td>XYY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>P4</td>
</tr>
</tbody>
</table>

Ideal Query:

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

hash(key1)

P1

P2

P3

1

0

0.5
CONSISTENT HASHING

$\text{hash(key1)}$

$\text{hash(key2)}$

P1

P2

P3

0.5

1.0
CONSISTENT HASHING

New Partition
If hash(key) = P4

New Partition

CONSISTENT HASHING
CONSISTENT HASHING

New Partition
CONSISTENT HASHING

New Partition
CONSISTENT HASHING

Replication Factor = 3
**CONSISTENT HASHING**

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.

Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.
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

Coordinator

Lock Request

Application Server

Partitions

P1
P2
P3
P4
Coordinating a distributed application involves a centralized Coordinator. The Coordinator receives Lock Request messages from Application Servers and sends Acknowledgement messages in response. Each partition (P1, P2, P3, P4) is managed by the Coordinator to ensure consistent access and synchronization across the distributed system.
CENTRALIZED COORDINATOR

Commit Request

Coordinator

Partitions

Application Server

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Application Server -> Coordinator

Commit Request

Safe to commit?

Partitions:
- P1
- P2
- P3
- P4
CENTRALIZED COORDINATOR

Commit Request

Coordinator

Acknowledgement

Safe to commit?

Application Server

Partitions

P1

P2

P3

P4
CENTRALIZED COORDINATOR

Middleware

Query Requests

Application Server

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

P2

P3

P4

P1→ID: 1–100
P2→ID: 101–200
P3→ID: 201–300
P4→ID: 301–400
Centralized Coordinator

Middleware

Safe to commit?

Partitions

P1

P2

P3

P4

Application Server

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

Leader Node

Safe to commit?

Partitions

P1

P2

P3

P4
OBSERVATION

We have assumed that the nodes in our distributed systems are running the same DBMS software. But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.
FEDERATED DATABASES

Distributed architecture that connects disparate DBMSs into a single logical system.
→ Expose a single query interface that can access data at any location.

This is hard and nobody does it well
→ Different data models, query languages, limitations.
→ No easy way to optimize queries
→ Lots of data copying (bad).
FEDERATED DATABASE EXAMPLE

- Application Server
- Middleware
  - Query Requests
  - Connectors
  - Back-end DBMSs
    - MySQL
    - MongoDB
    - redis
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 (Permanent + Ephemeral).
→ 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

A=2

Node 1

NETWORK

Set B=7

B=7

Node 2

Application Server
DISTRIBUTED 2PL

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

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

NETWORK
DISTRIBUTED 2PL

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

NODE 1
A=2

NETWORK

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

Application Server
DISTRIBUTED 2PL

Waits-For Graph

Application Server

Set A=2

Set

T₁

T₂

=7

=0

Application Server

Set B=7

Set

Node 1

Node 2

A=2

B=7

NETWORK
CONCLUSION

We 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