CS 6400 A

Database Systems Concepts and Design

Lecture 24

11/19/25

Agenda

1. Parallel DBMS

2. Distributed DBMS

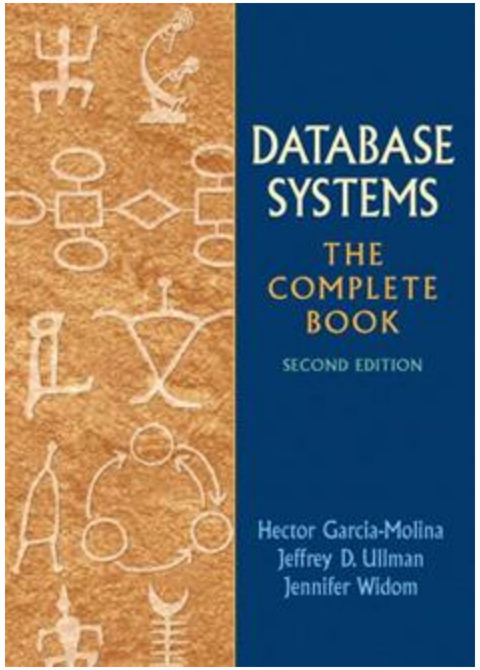
Reading Materials

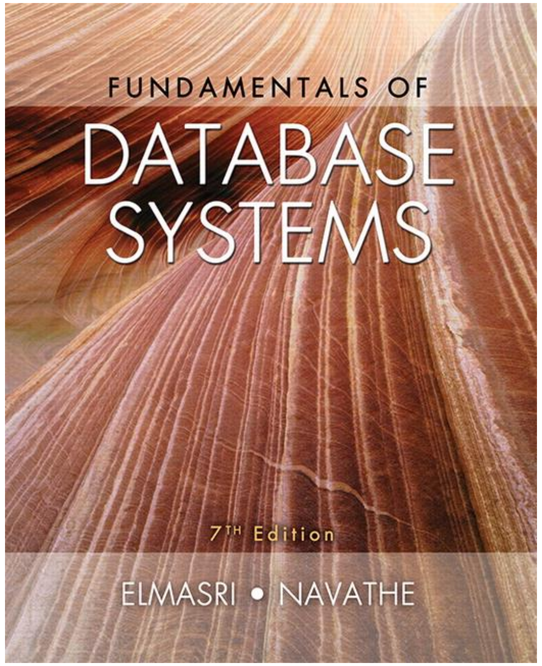
Database Systems: The Complete Book (2nd edition)

- Chapter 20: Parallel and Distributed Databases

Fundamental of Database Systems (7th Edition)

- Chapter 23: Distributed Database Concepts





1. Parallel DBMS

Parallel vs. Distributed DBMS

Parallel Databases:

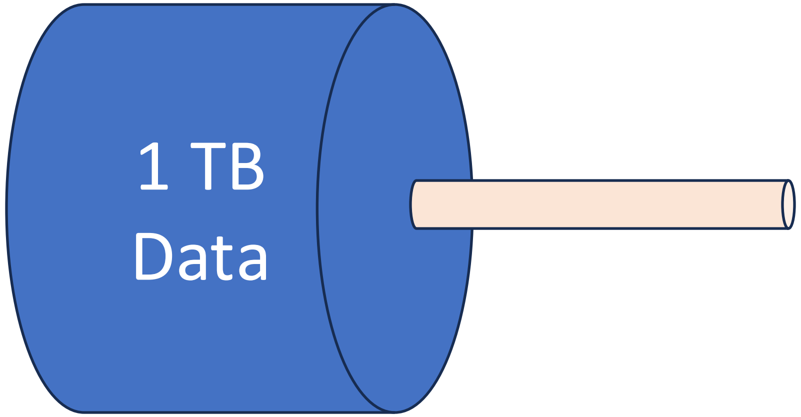
- Parallelization of various operations
  - e.g., loading data, building indexes, evaluating queries
- Data may or may not be distributed initially
- Distribution is governed by performance consideration

Distributed Databases:

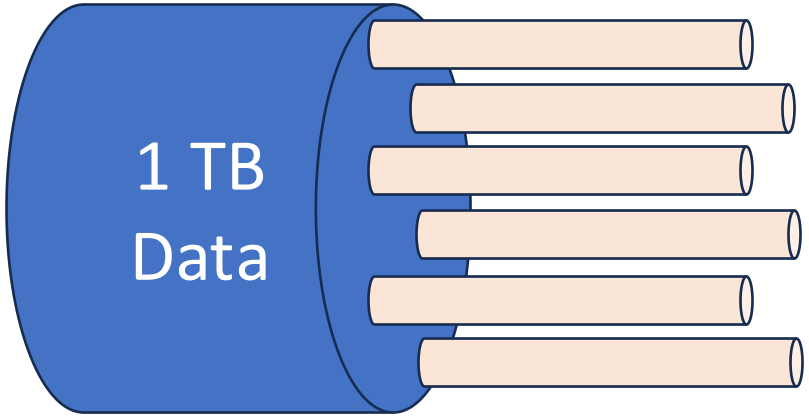
- Data is physically stored across different sites
  - Each site is typically managed by an independent DBMS
- Location of data and autonomy of sites have an impact on query optimization, concurrency control and recovery
- Distribution also governed by other factors
  - Increased availability for system crash
  - Local ownership and access

Benefits of Parallelism

Parallelism: divide a big problem into many smaller ones to be solved in parallel



At 10 MB/s
1.2 days to scan

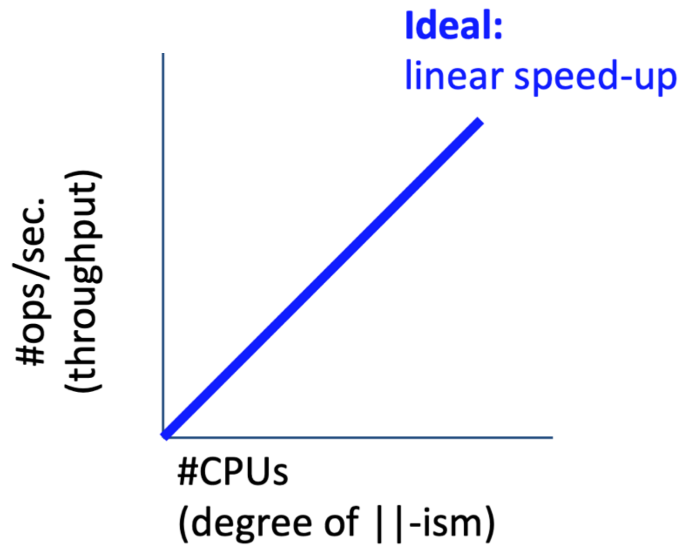


1,000 x parallel
1.5 minute to scan

|| Terminologies

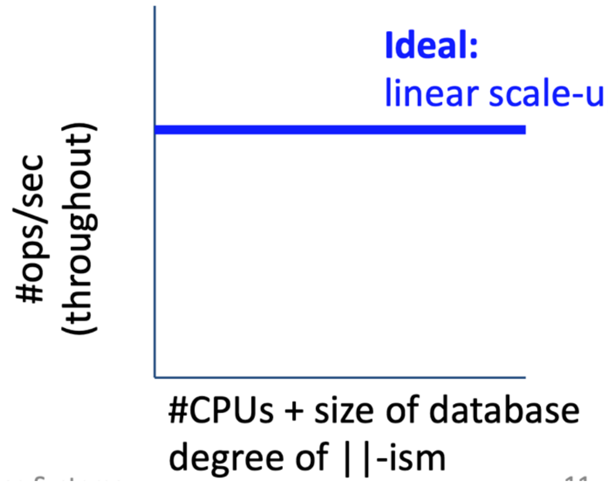
Speed-up

- More resources means proportionally less time for given amount of data



Scale-up

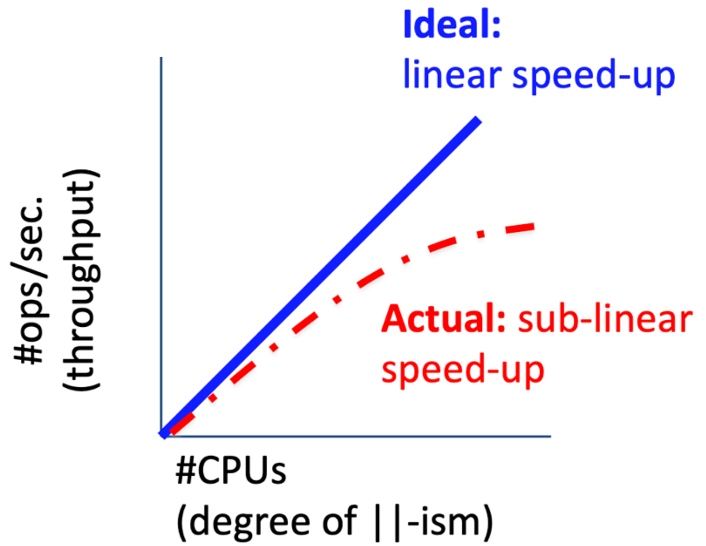
- If resources increased in proportion to increase in data size, time is constant.

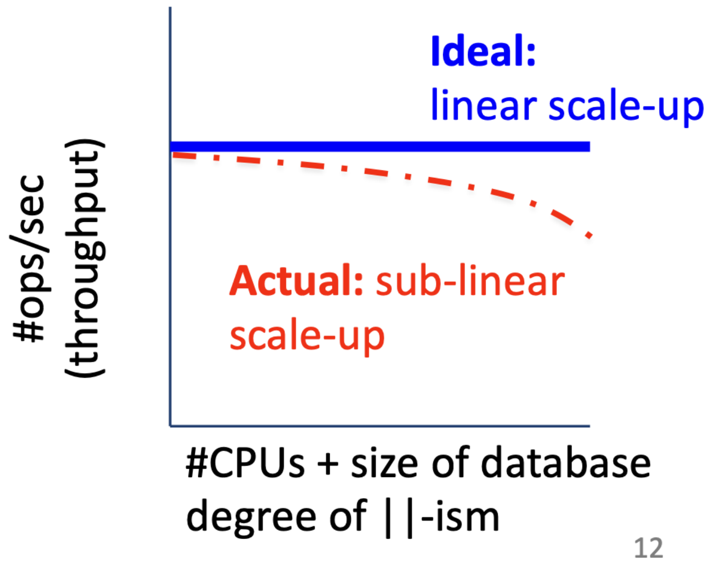


|| Terminologies

In practice, due to overheads in parallel processing:

- **Start-up cost**: Starting the operation on many processor, might need to distribute data
- **Interference**: Different processors may compete for the same resources
- **Skew**: The slowest processor (e.g. with a huge fraction of data) may become the bottleneck





Models of Parallelism

Units: a collection of processors

- Think hundreds or thousands of processors
- assume always have local cache
- may or may not have local memory or disk (next)

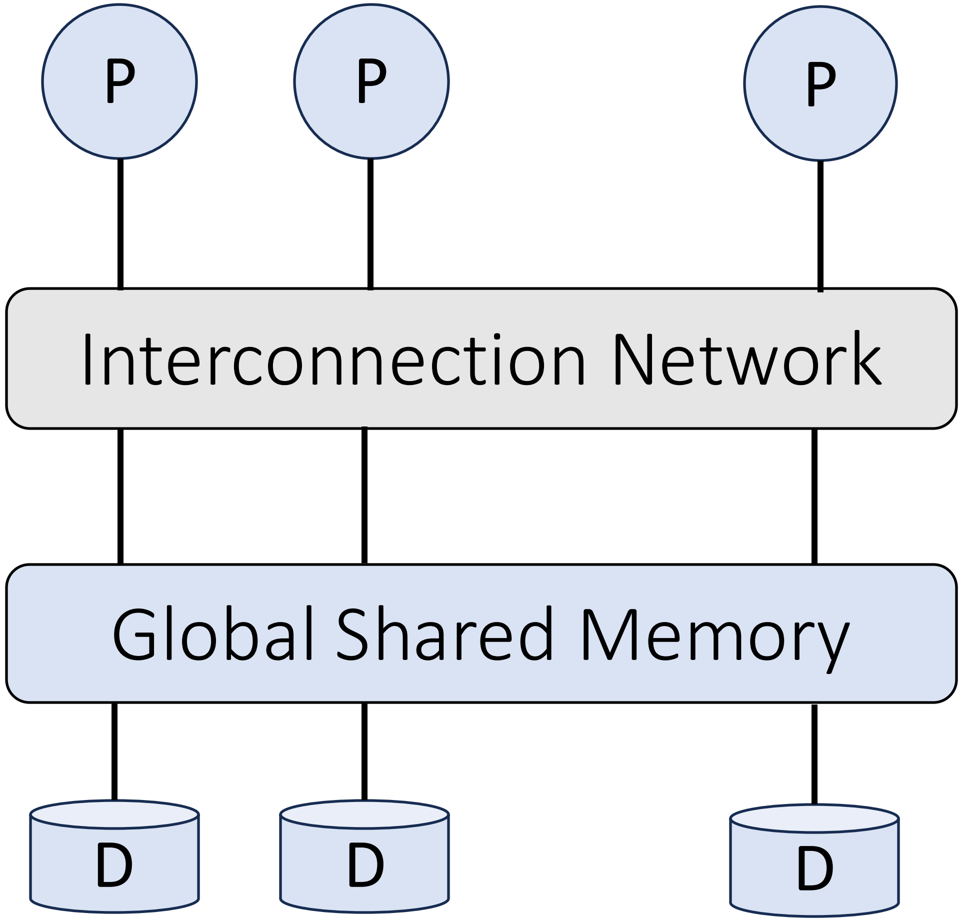
A communication facility to pass information among processors

- a shared bus or a switch

Different architecture:

- Whether memory AND/OR disk are shared
- 3 main groups: shared-memory, shared-disk, shared-nothing

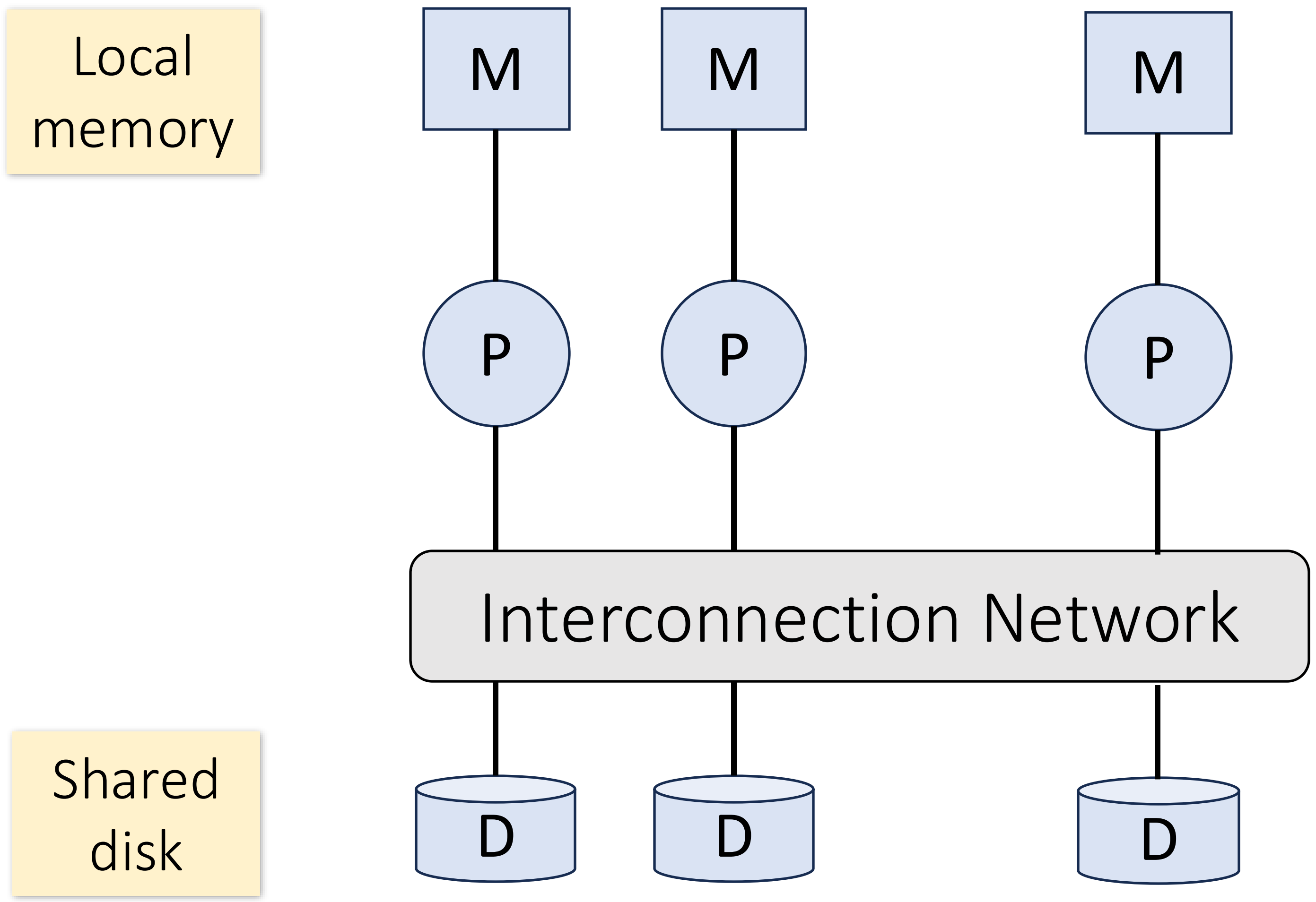
Shared-Memory Architecture



Shared memory

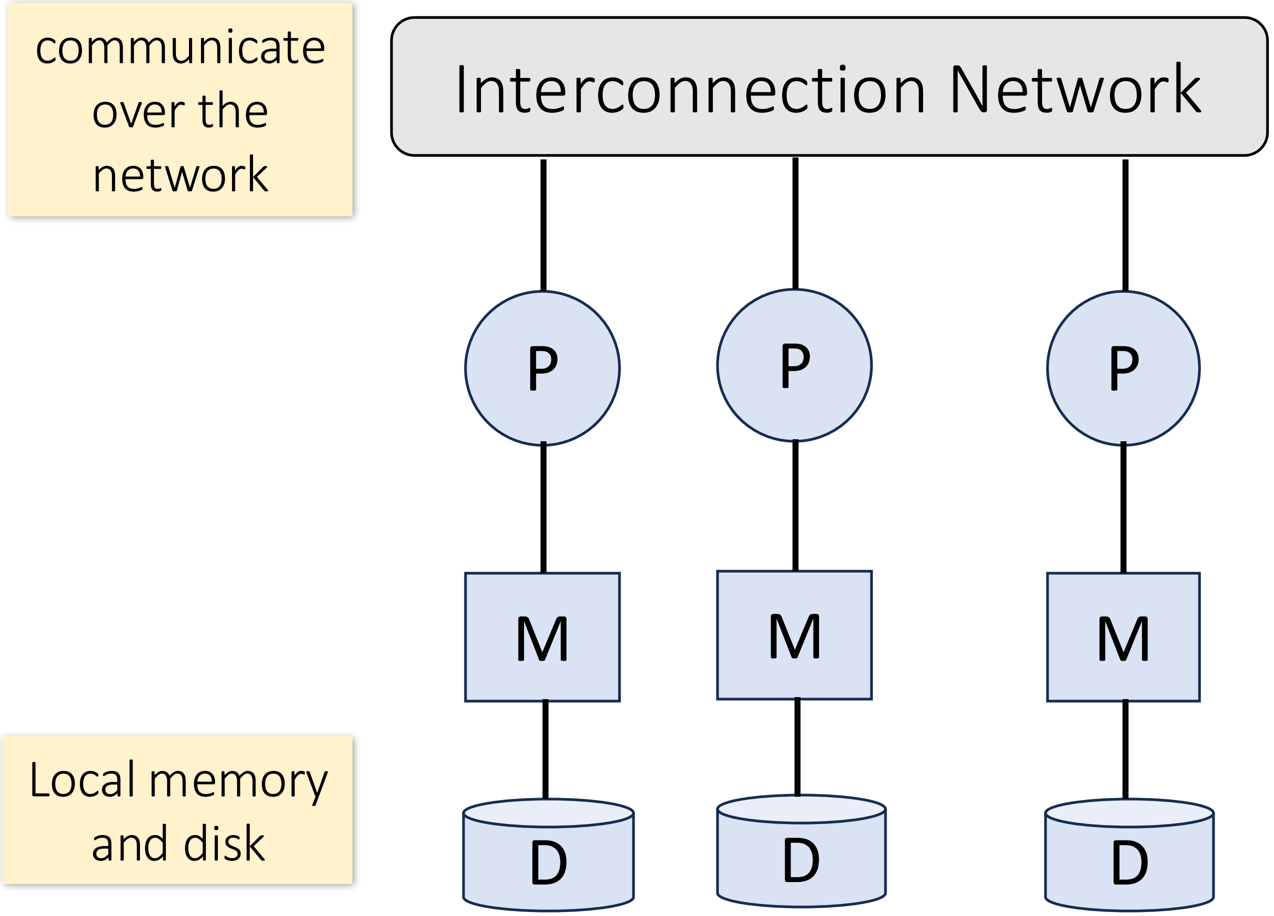
- e.g., NUMA (non uniform memory access)
- Easy to program
- Low communication overhead due to shared memory
- Difficult to scale up (memory contention) and expensive to build

Shared-Disk Architecture



- Centralized storage system (e.g., SAN or NAS), but compute is distributed
- Better scalability than shared memory, but still subject to contention of disk/network bandwidth

Shared-Nothing Architecture

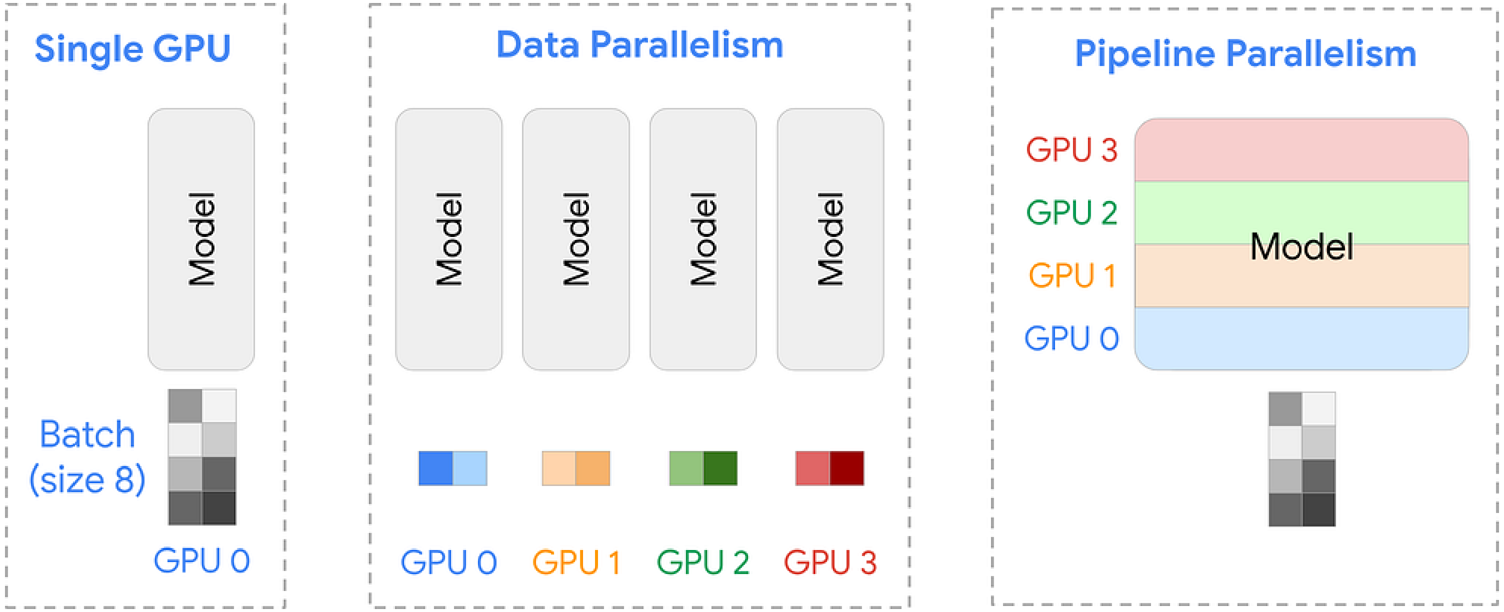


- Excellent horizontal scalability; relatively inexpensive to build
- Minimal resource contention but higher communication overhead
- Hard to program and design parallel algos

* We will assume this architecture by default

Types of Parallelism

- **Pipelining:** each machine does one component of the calculation, then passes the result on to another machine
- **Partitioning:** each machine runs the same computations on a different set of data.



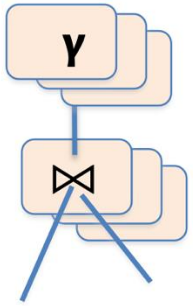
The same concepts are also used in distributed model training

Image source: https://selamjie.medium.com/the-frontier-of-pipeline-parallelism-an-overview-d4264cc9f877

Types of DBMS Query Parallelism

Intra-operator parallelism

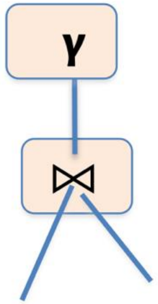
- get all machines working to compute a given ope (scan, sort, join)
- Achieved via partitioning
- OLAP (decision support)



*Our focus

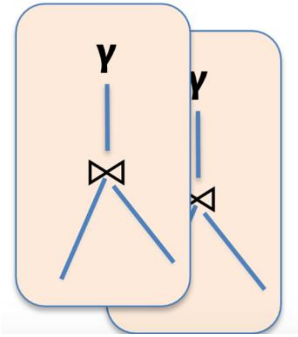
Inter-operator parallelism

- each operator may run concurrently on a different site
- Achieved via pipelining
- For both OLAP and OLTP



Inter-query parallelism

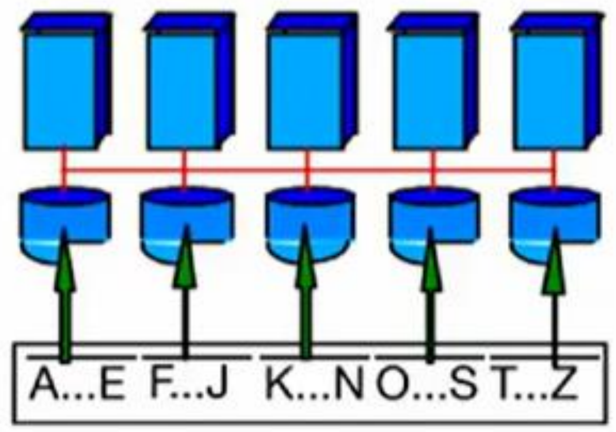
- different queries run on different sites
- For OLTP



Ack: Slide by Prof. Dan Suciu

Common Data Partitioning Schemes

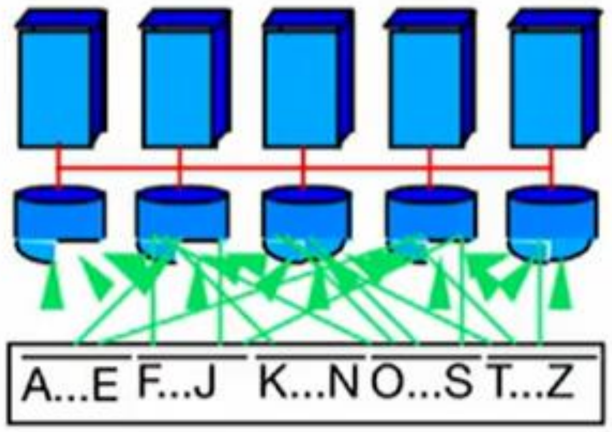
Range



Good for:

- Point look up
- Range queries
- Parallel SMJ

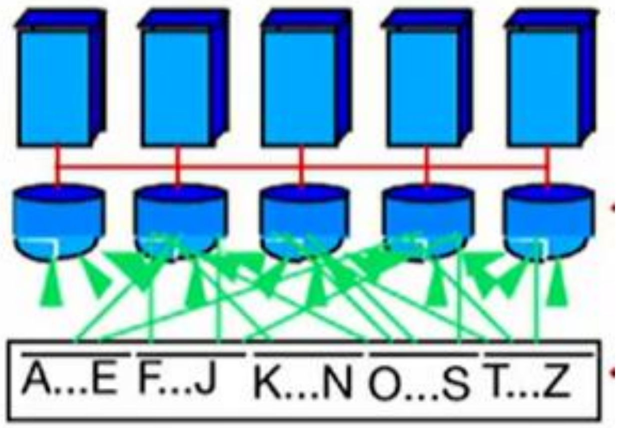
Hash



Good for:

- Point look up (but not for range queries)
- Parallel HJ

Round-Robin



Good for:

- Spreading the load
- When the entire relation is accessed

Shared disk and memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

Ack: Berkeley CS186

In-class Exercise

```
SELECT *
FROM   students
WHERE  name = 'Jane Doe'
```

Assume that we have 5 machines and a 1000 page **students(sid, name, gpa)** table. Assume pages are 1KB.

- How many IOs will it take to execute the above query under round-robin partitioning?
- Suppose that we hash partition on the **name** column instead. How many IOs will the query take?
- Assume that an IO takes 1ms and the network cost is negligible. How long will the query take if the data is round-robin partitioned and if the data is hash partitioned on the **name** column.

Parallel Algorithms - Sorting

A simple idea:

- Have each CPU sort the part of the relation that is on its local disk
- Merge the sorted results   Performance bottleneck

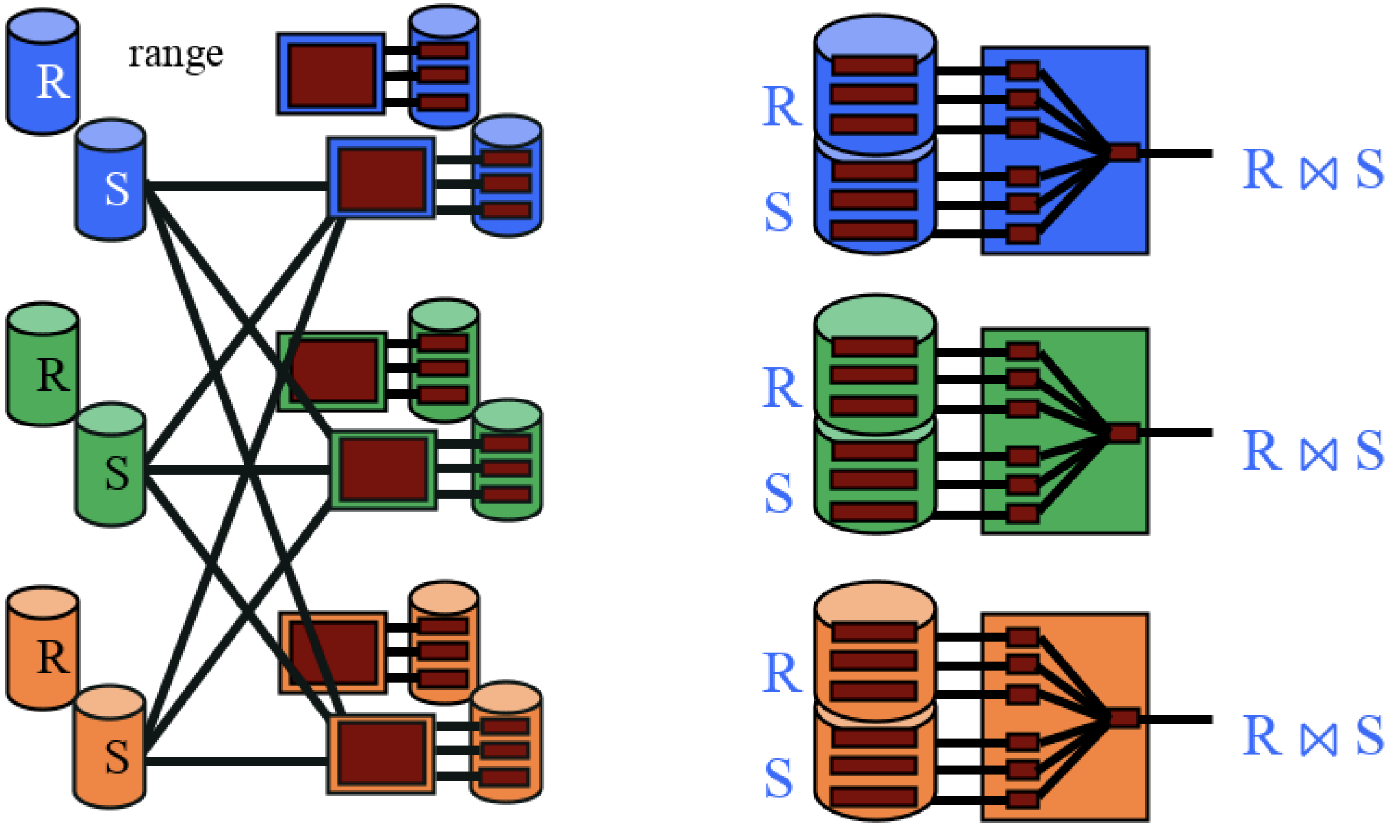
A better idea:

- Redistribute relation using range partitioning   1 pass
- Perform local sort on each machine

Parallel Algorithms – Sort Merge Join

Two Steps:

- Range partition each table **using the same ranges** on the join column
- Perform local sort merge join on each machine

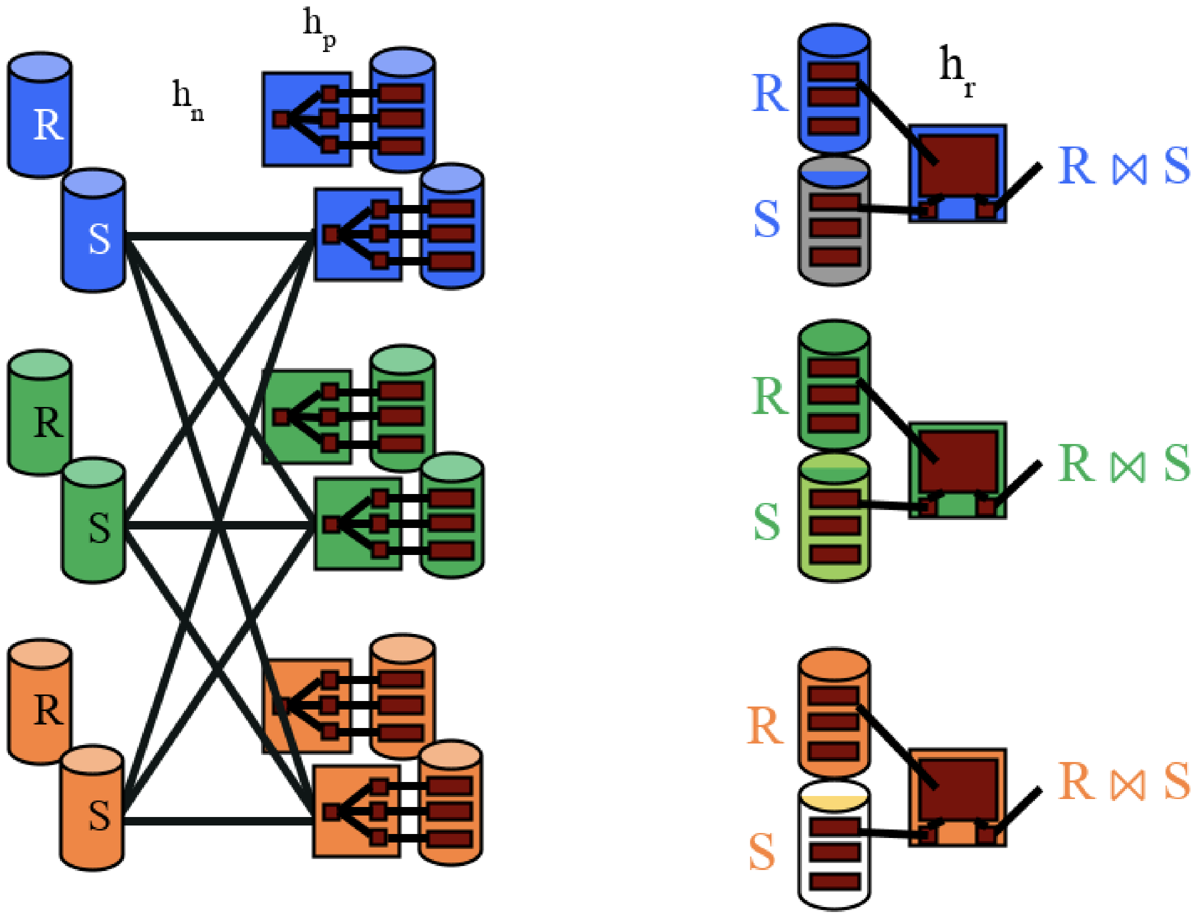


Ack: Figures from Berkeley CS186

Parallel Algorithms – Hash Join

Two steps

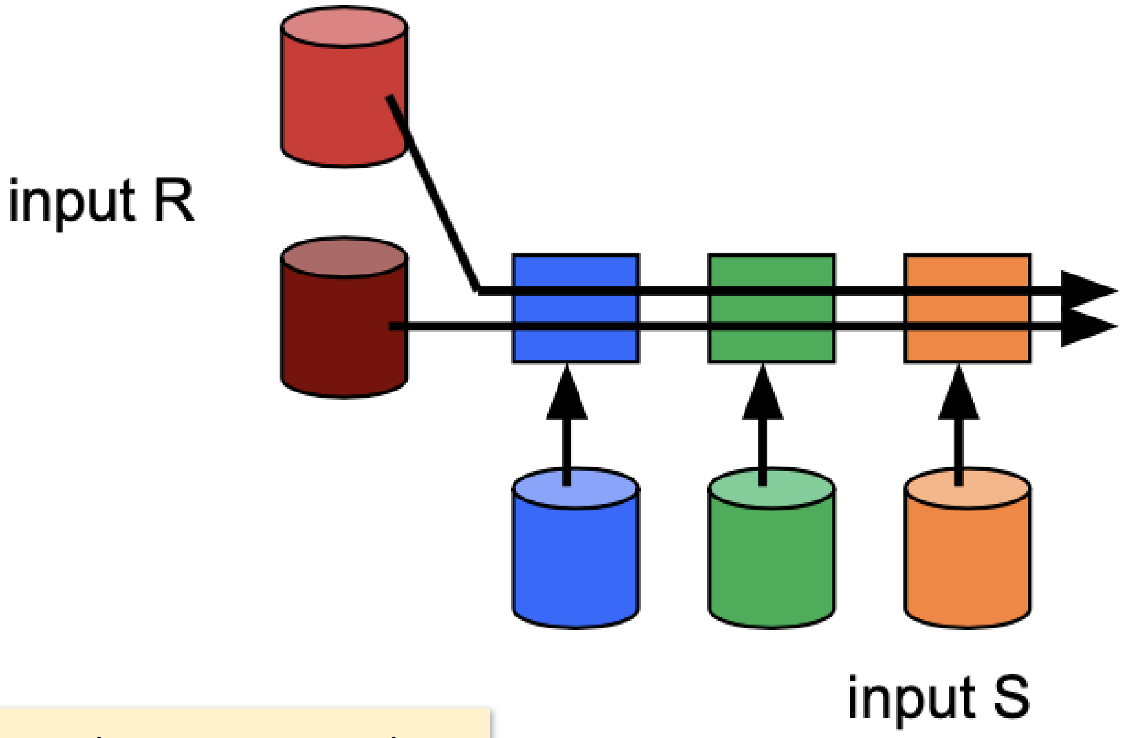
- Hash partition each table **using the same hash function** on the join column
- Perform local grace hash join on each machine



Ack: Figures from Berkeley CS186

Parallel Algorithms – Broadcast Join

- Sometimes, one join table is tiny and another table is huge.
- Too expensive to hash/range partition the large table
- “Broadcast” the small table to every machine; each machine will then perform a local join



More compute, less network

Ack: Figures from Berkeley CS186

Parallel DBMS vs MR

- Many commonalities:
  - Designed for large-scale data processing
  - Use data partitioning
- Reading for assignment 3

A Comparison of Approaches to Large-Scale Data Analysis

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Erik Paulson
University of Wisconsin
epaulson@cs.wisc.edu

Alexander Rasin
Brown University
alexr@cs.brown.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

David J. DeWitt
Microsoft Inc.
dewitt@microsoft.com

Samuel Madden
M.I.T. CSAIL
madden@csail.mit.edu

Michael Stonebraker
M.I.T. CSAIL
stonebraker@csail.mit.edu

ABSTRACT

There is currently considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis [17]. Although the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years, some have called MR a dramatically new computing model [8, 17]. In this paper, we describe and compare both paradigms. Furthermore, we evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a benchmark consisting of a collection of tasks that we have run on an open source version of MR as well as on two parallel DBMSs. For each task, we measure each system's performance for various degrees of parallelism on a cluster of 100 nodes. Our results reveal some interesting trade-offs. Although the process to load data into and tune the execution of parallel DBMSs took much longer than the MR system, the observed performance of these DBMSs was strikingly better. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel databases*

General Terms

Database Applications, Use Cases, Database Programming

1. INTRODUCTION

Recently the trade press has been filled with news of the revolution of "cluster computing". This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of low-end servers instead of deploying a smaller set of high-end servers. With this rise of interest in clusters has come a proliferation of tools for programming them. One of the earliest and best known such tools in MapReduce (MR) [8]. MapReduce is attractive because it provides a simple model through which users can express relatively sophisticated distributed programs, leading to significant interest in the educational community. For example, IBM and Google have announced plans to make a 1000 processor MapReduce cluster available to teach students distributed programming.

Given this interest in MapReduce, it is natural to ask "Why not use a parallel DBMS instead?" Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Aster Data, Netezza, DATAllegro (and therefore soon Microsoft SQL Server via Project Madison), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high-level programming environment and parallelize readily. Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Inspired by this question, our goal is to understand the differences between the MapReduce approach to performing large-scale data analysis and the approach taken by parallel database systems. The two classes of systems make different choices in several key areas. For example, all DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format. Other differences also include how each system provides indexing and compression optimizations, programming models, the way in which data is distributed, and query execution strategies.

The purpose of this paper is to consider these choices, and the trade-offs that they entail. We begin in Section 2 with a brief review of the two alternative classes of systems, followed by a discussion in Section 3 of the architectural trade-offs. Then, in Section 4 we present our benchmark consisting of a variety of tasks, one taken from the MR paper [8], and the rest a collection of more demanding tasks. In addition, we present the results of running the benchmark on a 100-node cluster to execute each task. We tested the publicly available open-source version of MapReduce, Hadoop [1], against two parallel SQL DBMSs, Vertica [3] and a second system from a major relational vendor. We also present results on the time each system took to load the test data and report informally on the procedures needed to set up and tune the software for each task.

In general, the SQL DBMSs were significantly faster and required less code to implement each task, but took longer to tune and load the data. Hence, we conclude with a discussion on the reasons for the differences between the approaches and provide suggestions on the best practices for any large-scale data analysis engine.

Some readers may feel that experiments conducted using 100

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

2. Distributed DBMS

Parallel vs Distributed DBMS

Parallel Database:

- Nodes are physically close to each other.
- Nodes are connected via high-speed LAN (fast, reliable communication fabric).
- The communication cost between nodes is assumed to be small. As such, one does not need to worry about nodes crashing or packets getting dropped when designing internal protocols.

Distributed Database:

- Nodes can be far from each other.
- Nodes are potentially connected via a public network, which can be slow and unreliable.
- The communication cost and connection problems cannot be ignored (i.e., nodes can crash, and packets can get dropped).

Storing Data in a Distributed DBMS

A single relation may be partitioned or fragmented across several sites

- typically at sites where they are most often accessed

The data can be replicated as well – when the relation is in high demand or for robustness

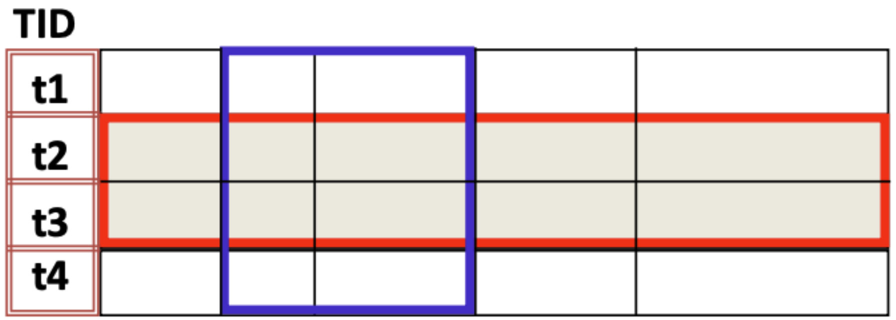
Storing Data in a Distributed DBMS

Horizontal Fragmentation (Sharding):

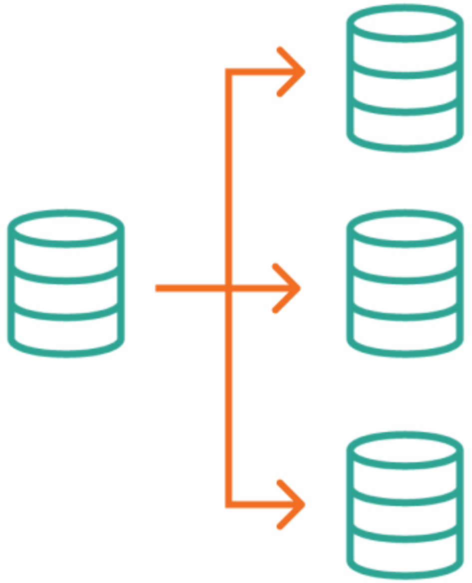
- Usually disjoint
- Can often be identified by a selection query
- To retrieve the full relation, need a union

Vertical Fragmentation:

- Identified by projection queries
- Typically unique TIDs added to each tuple
- TIDs replicated in each fragments
- Ensures that we have a Lossless Join



Replication



Storing multiple copies of a relation

Motivation

- Increased availability: If a site that contains a replica goes down, we can find the same data at other sites
- Faster query evaluation: Queries can execute faster by using a local copy of a relation instead of going to a remote site.

Two types of replication: synchronous vs asynchronous

- how replicas are kept current when the relation is modified

Partitioning vs Sharding vs Replication

- Partitioning: Splitting a large table into smaller pieces; horizontal (split rows) or vertical (split columns); within a single database server

- Sharding: Distribute data across multiple servers/machines (each called a shard); each shard is an independent database; horizontal scalability

- Replication: Creating copies (replicas) of the same data across multiple servers

Updating Distributed Data

Synchronous Replication: All copies of a modified relation (or fragment) must be updated before the modifying transaction commits

- Always updated but expensive commit protocols (2 Phase Commit)
- By "voting" - Write Quorum (W) + Read Quorum (R) > Total Replicas (N)

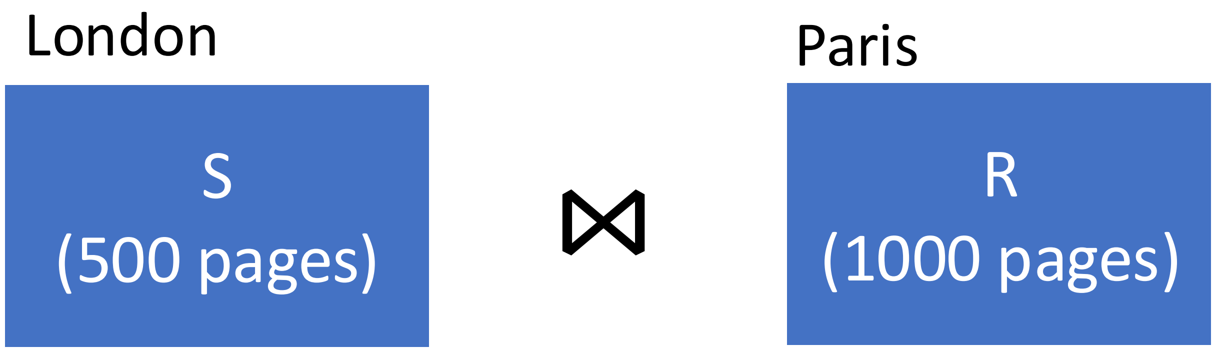
Asynchronous Replication: Copies of a modified relation are only periodically updated; different copies may get out-of-sync in the meantime

- More efficient – many current products follow this approach
- Primary site (one master copy) or peer-to-peer (multiple master copies)

Joins in Distributed DBMS

Can be very expensive if relations are stored at different sites!

Goal: Ship R to London, then do join with S at London.



Semijoin ($\ltimes$)

London

S
(500 pages)

Paris

R
(1000 pages)

1. At London, project S onto join columns and ship this to Paris
2. At Paris, join S-projection with R
   - Result is the reduction of R w.r.t. S (only these tuples are needed)
3. Ship reduction of R to back to London
4. At London, join S with reduction of R

Tradeoff the cost of computing and shipping projection for cost of shipping full R relation

Bloomjoin

London	Paris
S (500 pages)	R (1000 pages)

1. At London, compute a bit-vector of some size k:
   - Hash column values into range 0 to k-1
   - If some tuple hashes to p, set bit p to 1 (p from 0 to k-1)
   - Ship bit-vector to Paris
2. At Paris, hash each tuple of R similarly
   - discard tuples that hash to 0 in S's bit-vector
   - Result is called reduction of R w.r.t S
3. Ship "bit-vector-reduced" R to London
4. At London, join S with reduced R

Distributed Query Optimization

Similar to centralized optimization, but have differences

- Communication costs must be considered. If we have several copies of a relation, must decide which copy to use
- Local site autonomy must be respected
- New distributed join methods should be considered

Query site constructs global plan, with suggested local plans describing processing at each site

- If a site can improve suggested local plan, free to do so

Distributed Transaction

A given transaction is submitted at one site, but it can access data at other sites.

When a transaction is submitted at some site, the transaction manager at that site breaks it up into a collection of *sub-transactions* that can be executed at different sites.

New questions:

- Distributed CC: How can locks for objects stored across several sites be managed?
- Distributed Recovery: how to ensure transaction atomicity when data is distributed across sites?

Distributed Concurrency Control

Lock management can be distributed across sites in many ways:

- Centralized: A single site is in charge of handling lock and unlock requests for all objects.
- Primary copy: One copy of each object is designated as the primary copy.
  - All requests to lock or unlock a copy of this object are handled by the lock manager at the site where the primary copy is stored.
- Fully distributed: Requests to lock or unlock a copy of an object stored at a site are handled by the lock manager at the site where the copy is stored.

Distributed Recovery

Two new issues:

- New kinds of failure, e.g., network/remote site failures
- If “sub-transactions” of a transaction execute at different sites, all or none must commit

Need a commit protocol to achieve this

- Goal: Ensures atomicity (all-or-nothing commits) for transactions spanning multiple sites in a distributed database.
- Most widely used: Two Phase Commit (2PC)

This is not to be confused with 2PL!

Two-phase Commit (2PC)

A log is maintained at each site – as in a centralized DBMS – commit protocol actions are additionally logged

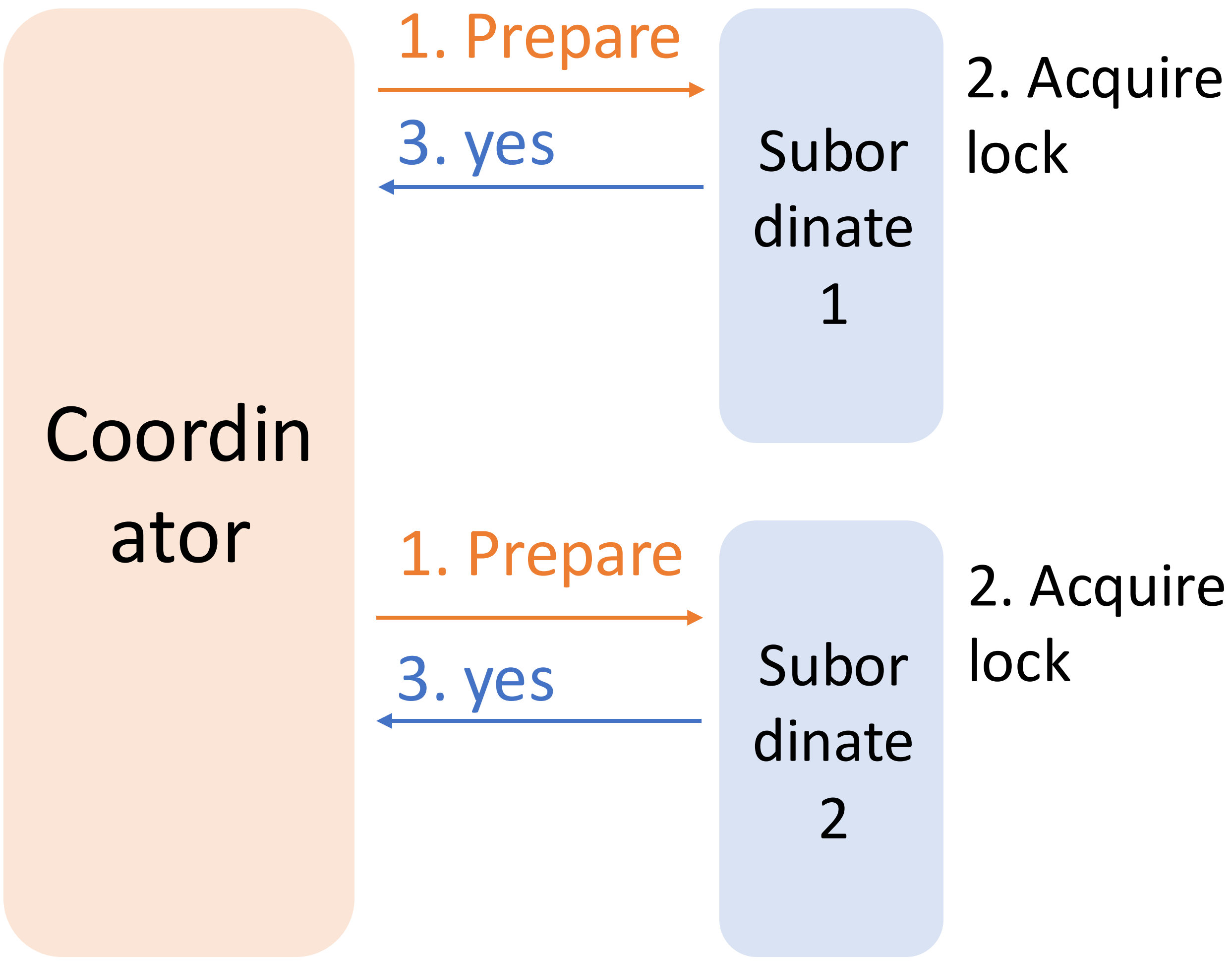
Site at which transaction originates is coordinator

Other sites at which it executes are subordinates

- w.r.t. coordination of this transaction

Two-phase Commit (2PC)

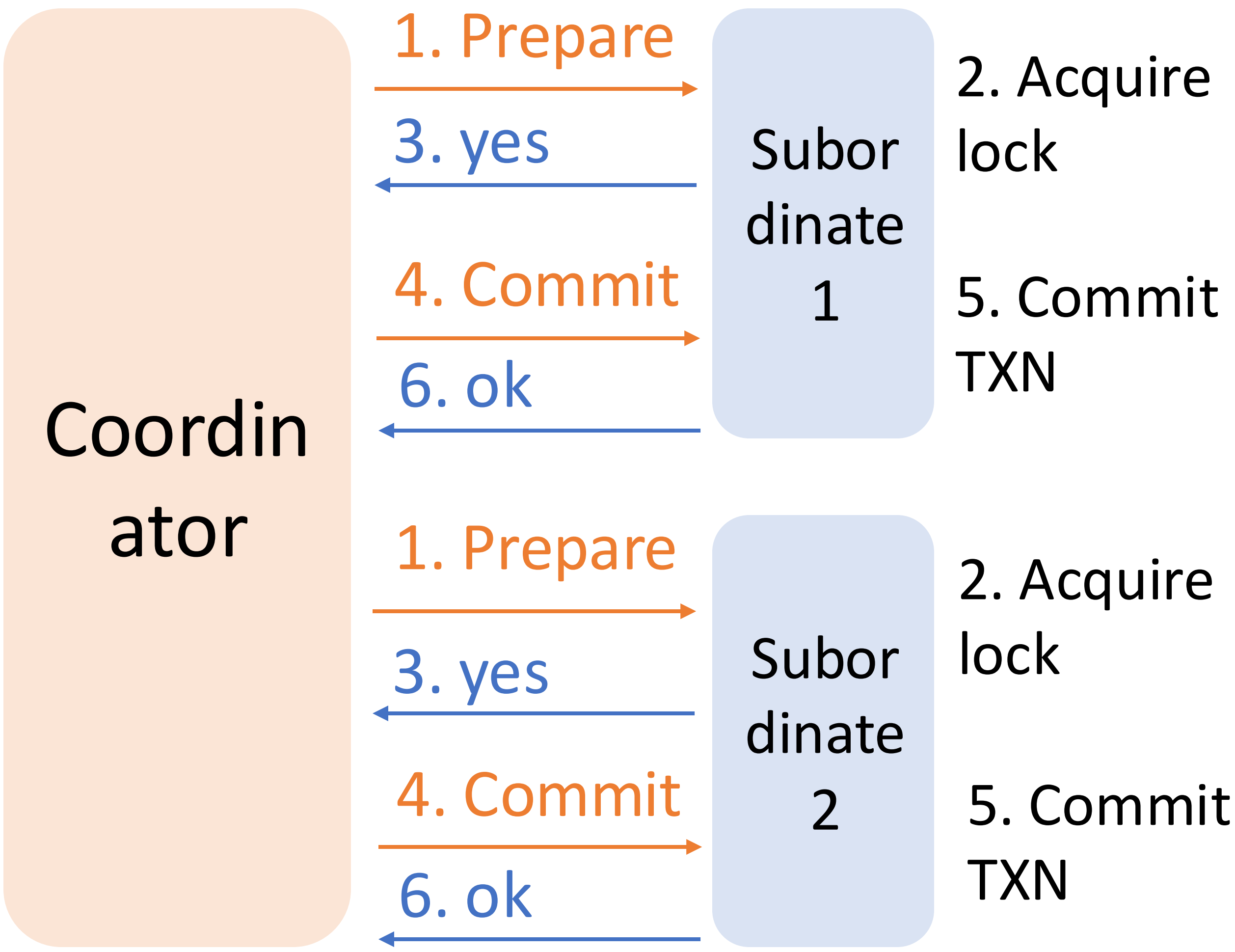
Prepare Phase: asking other nodes whether they can commit the proposed transaction.



Two-phase Commit (2PC)

Prepare Phase: asking other nodes whether they can commit the proposed transaction.

Commit Phase: commanding other nodes to either commit or abort the proposed transaction



2PC comments

Two rounds of communication

- voting => termination
- Both initiated by the coordinator

Any site (coordinator or subordinate) can unilaterally decide to abort a transaction

- but consensus needed to commit

Every message reflects a decision by the sender

- to ensure that this decision survives failures, it is first recorded in the local log and is force-written to disk

Weakness of 2PC

2PC is often called a **"blocking"** atomic commit protocol (or **"anti-availability"** protocol) because all nodes/members must be up for it to work.

- In particular, if a coordinator dies, all nodes would have to wait to hear the final decision (commit or abort)
- Participant nodes cannot simply time out and abort the transaction because it promised to follow the final decision from the coordinator

We will see how Google Spanner mitigate this drawback