ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ ΚΥΒΕΡΝΟΑΣΦΑΛΕΙΑ ΚΑΙ ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ
MSc CYBERSECURITY AND DATA SCIENCE
DEPT OF INFORMATICS
UNIVERSITY OF PIRAEUS

Διαχείριση Μεγάλων Δεδομένων

Big Data Management

Lecture 2 – DBMS Architectures

From Centralized to Cloud Big Data Processing

Nikos Pelekis (npelekis@unipi.gr)

ver. 11.2022

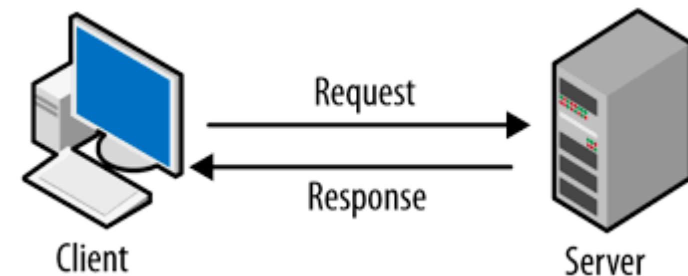
Outline

- Centralized Database Systems
  - Server System Architectures
- Parallel Systems
- Distributed Systems
  - Network Types
- Cloud Systems
- Apache Hadoop Ecosystem

Centralized Database Systems

- Database Software Runs on **a single computer system**
- Single-user system → pc, mobile
  - Embedded databases, e.g., SQL Lite, H2

- Multi-user systems also known as client server systems.
  - Service requests received from client systems
  - Processes by a central database server



Thin vs Thick Clients

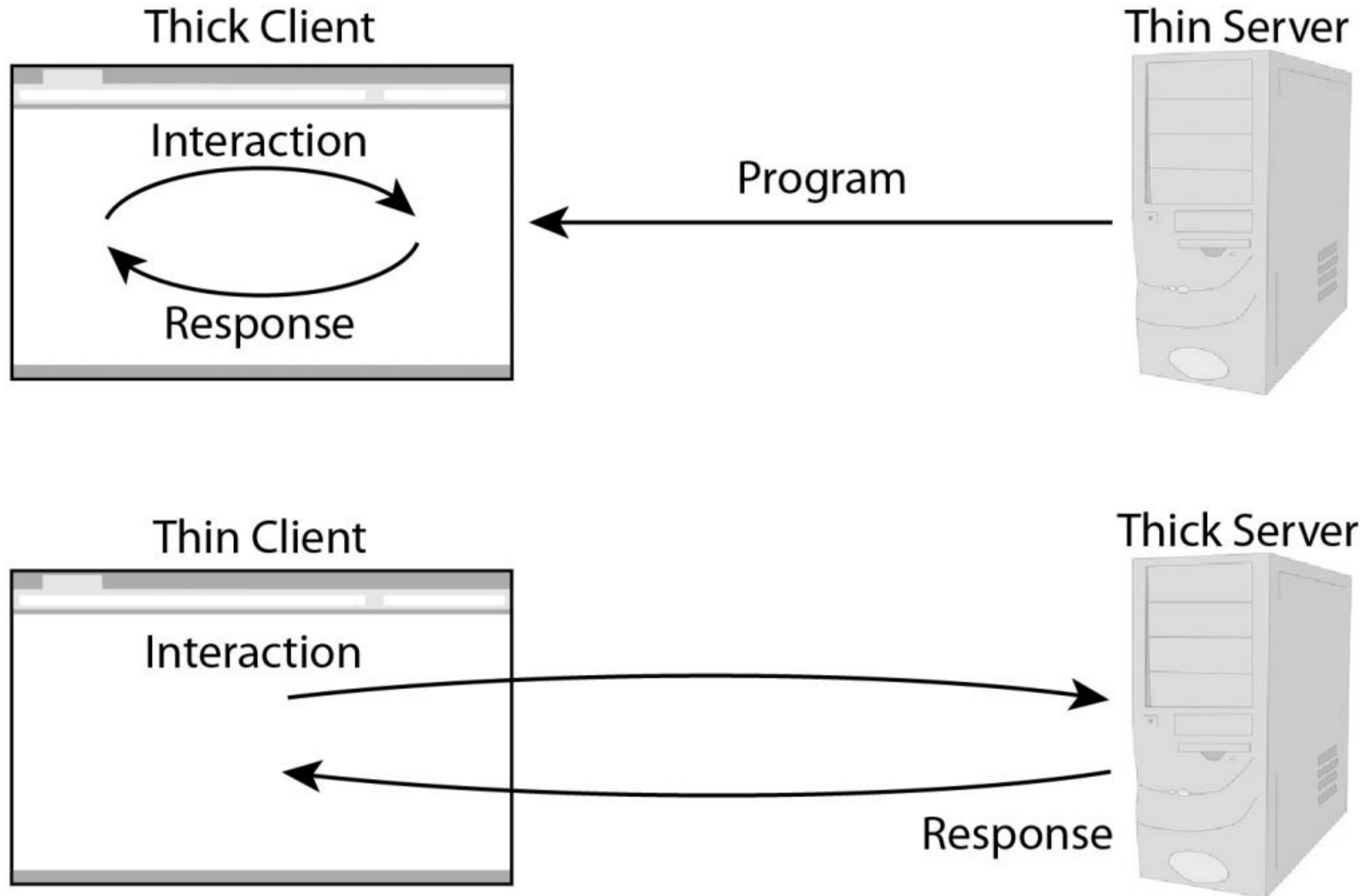


image credits: wikimedia.com

DB Server System Architecture

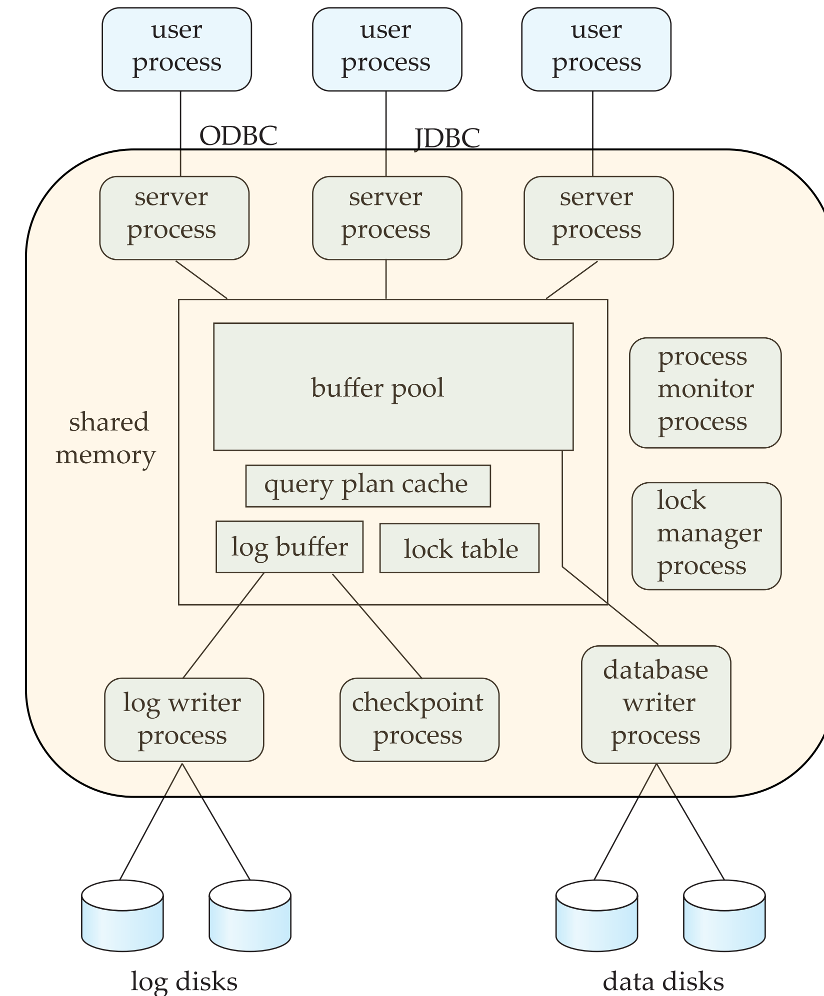
- Server systems can be broadly categorized into two kinds:
  - **Transaction servers**
    - Widely used in relational database systems, and
  - **Data servers**
    - Parallel data servers used to implement high-performance transaction processing systems

Transaction Servers

- Also called **query server systems** or SQL server systems
  - Clients send requests to the server
  - Transactions are executed at the server
  - Results are shipped back to the client.
- Requests are specified in SQL, and communicated to the server through a remote procedure call (RPC) mechanism.
  - Transactional RPC allows many RPC calls to form a transaction.
- Applications typically use ODBC/JDBC APIs to communicate with transaction servers

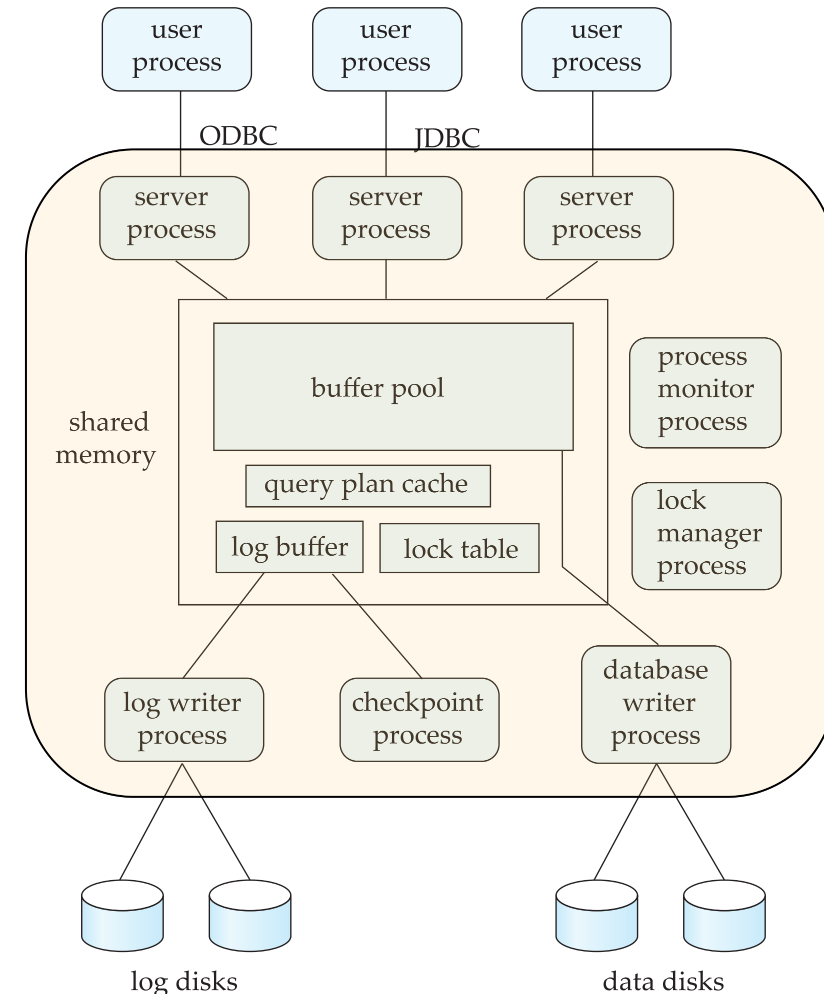
Transaction Servers

- A typical transaction server consists of multiple processes accessing data in **shared memory**
- Shared memory contains shared data
  - Buffer pool
  - Lock table
  - Log buffer
  - Cached query plans (reused if same query submitted again)
- All database processes can access shared memory
- Server processes
  - These receive user queries (transactions), execute them and send results back



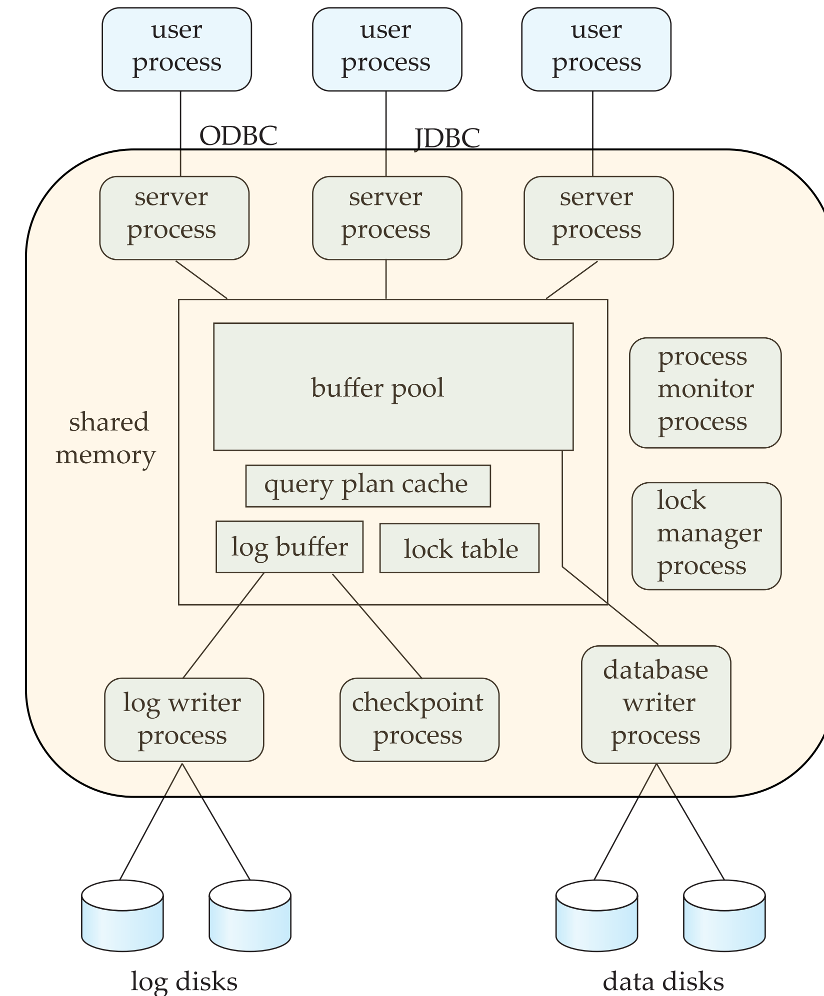
Transaction Servers

- Database writer process
  - Output modified buffer blocks to disks continually
- Log writer process
  - Server processes simply add log records to log record buffer
  - Log writer process outputs log records to stable storage.
- Checkpoint process
  - Performs periodic checkpoints
- Process monitor process
  - Monitors other processes, and takes recovery actions if any of the other processes fail
    - E.g. aborting any transactions being executed by a server process and restarting it



Transaction Servers

- Lock manager process
  - To avoid overhead of interprocess communication for lock request/grant, each database process operates directly on **the lock table**
  - **Lock table.** A table with locked items (e.g., tables, records) and processes in the queue
  - Lock manager process still used for deadlock detection (e.g., a never-ending process)
- To ensure that no two processes are accessing the same data structure at the same time, databases systems implement **mutual exclusion**



Data Servers/Data Storage Systems

- Data items are shipped to clients where processing is performed
  - Updated data items written back to server
- Earlier generation of data servers operated in disk **pages** containing multiple data items (records)
- Current generation data servers (also called data storage systems) work in units of **data items**
  - Commonly used data item formats include text, JSON, XML, or binary data

Data Servers/Storage Systems (Cont.)

- Prefetching
  - Prefetch items that may be used soon
- Adaptive lock granularity
  - Lock granularity escalation
    - switch from finer granularity (e.g. tuple) lock to coarser
  - Lock granularity de-escalation
    - Start with coarse granularity to reduce overheads, switch to finer granularity in case of more concurrency conflict at server

Data Servers (Cont.)

- Data Caching
  - Data can be cached at client even in between transactions
  - But check that data is up-to-date before it is used (cache coherency)
  - Check can be done when requesting lock on data item

PARALLEL DATABASES

Parallel Systems

- Parallel database systems consist of **multiple processors** and **multiple disks** connected by a fast interconnection network.
- **Motivation**: handle workloads beyond what a single computer system can handle
  - E.g., Multi transaction processing, handling user requests at web-scale
  - Data intensive processing, Online Analytical Processing (Data Warehouse), ML support on very large amounts of data

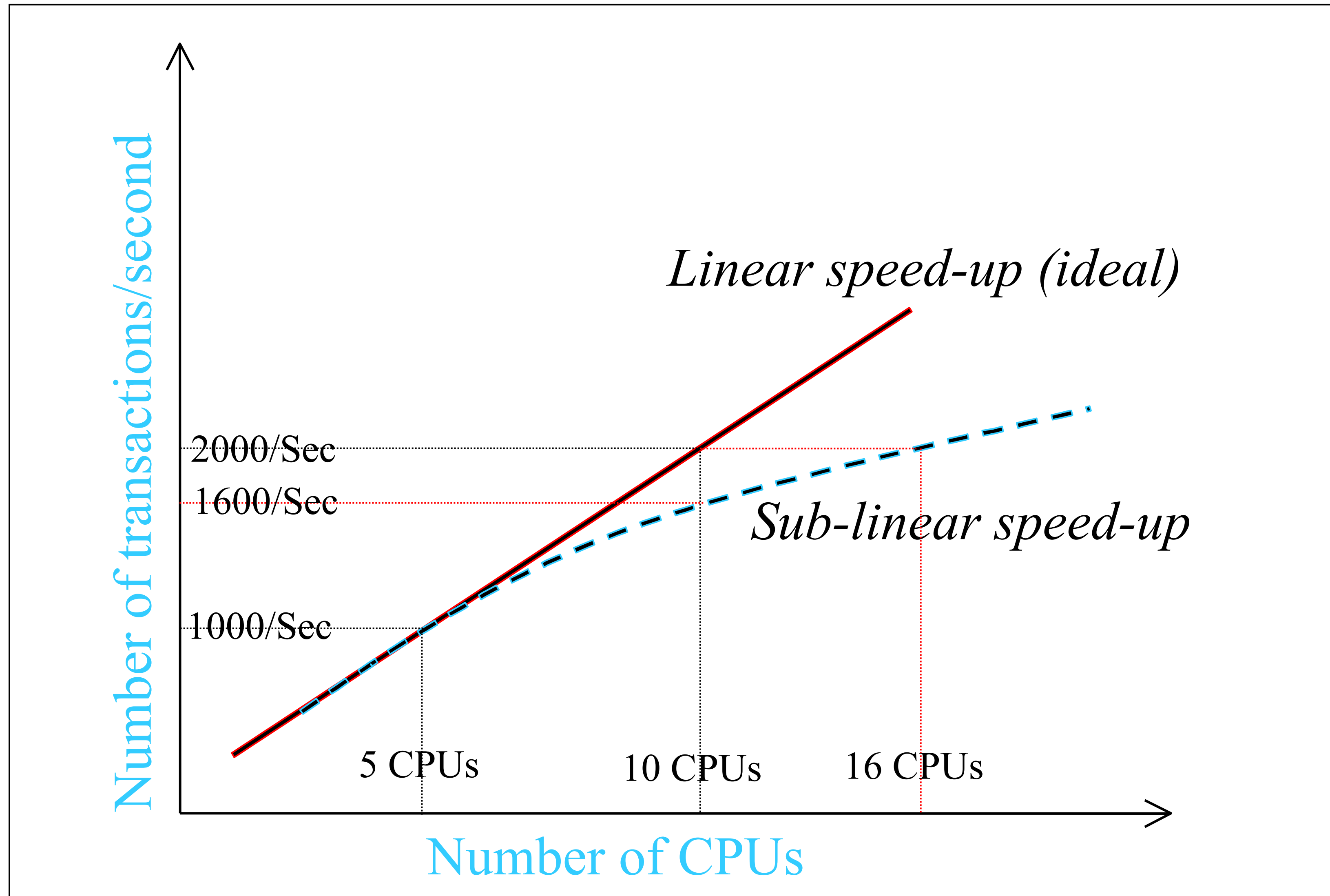
Parallel Systems (Cont.)

- A coarse-grain **parallel machine consists of a small number** of powerful processors
  - E.g., Intel's May 8, 2004 cancellation of its *Tejas* and *Jayhawk* processors, which is generally cited as the end of **frequency scaling** as the dominant computer architecture paradigm.
    - Multi Core CPUs are now everywhere (in desktop PCs, as well)
- A massively parallel or fine grain parallel machine **utilizes thousands of smaller processors**.
- What are the benefits?
  - **throughput** --- the number of tasks that can be completed in a given time interval **increases**
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted **decreases**

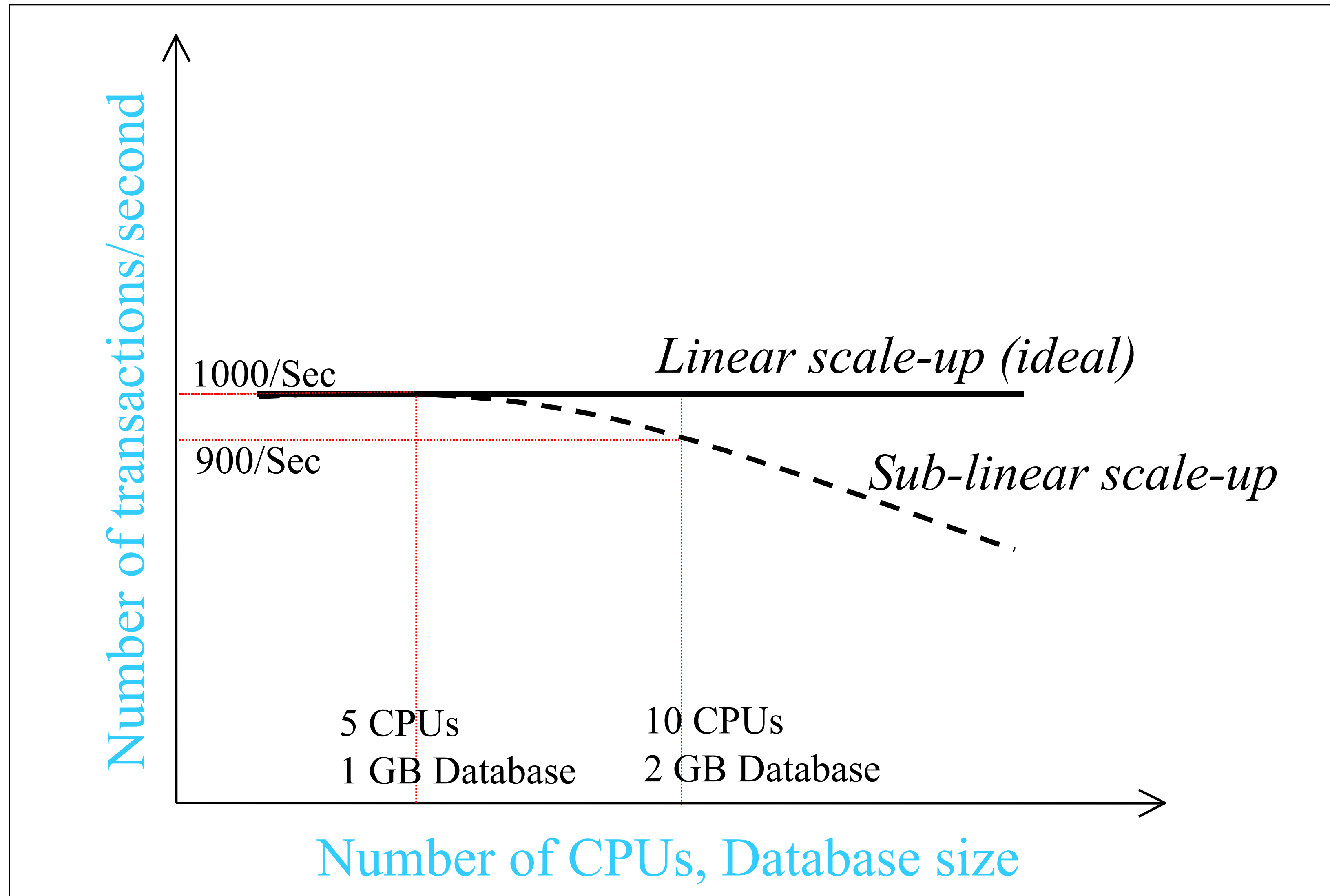
How to measure the benefits
Speed-Up vs Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is *N-times* larger.
  - Measured by: ***speedup = small system elapsed time / large system elapsed time***
  - Speedup is linear if equation equals N.
- **Scaleup**: increase the size of both the problem and the system
  - N-times larger system used to perform a N-times larger job
  - Measured by: ***scaleup = small system small problem elapsed time / big system big problem elapsed time***
  - Scale up is linear if equation equals 1.

Speedup



Scaleup



Batch and Transaction Scaleup

- Batch scaleup:
  - A single large job; typical of most decision support queries and scientific simulation.
  - Use a N-times larger computer on N-times larger problem.
- Transaction scaleup:
  - Numerous small queries submitted by independent users to a shared database; typical transaction processing and timesharing systems.
  - N-times as many users submitting requests (hence, N-times as many requests) to a N-times larger database, on an N-times larger computer.
  - Well-suited to parallel execution.

Factors Limiting Speedup and Scaleup

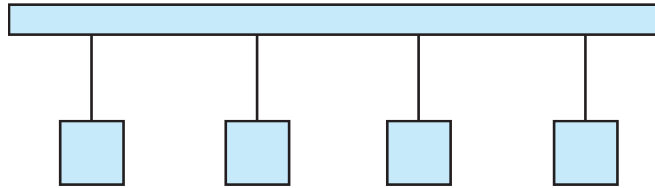
Speedup and scaleup are often sublinear due to:

- **Startup/sequential costs**: Cost of starting up multiple processes, and sequential computation before/after parallel computation
  - May dominate computation time, if the degree of parallelism is high
  - Suppose $p<1$ (e.g., 95%) is the **parallelizable** proportion of computation and $n>1$ the speedup enhancement (e.g. 2X faster)
  - Amdahl's law: speedup limited to: $1 / [(1-p)+(p/n)]$
  - Gustafson's law: scaleup limited to: $1 / [n(1-p)+p]$
- **Interference**: Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.
- **Skew**: Increasing the degree of parallelism increases the variance in service times of parallelly executing tasks. Overall execution time determined by slowest of parallelly executing tasks.

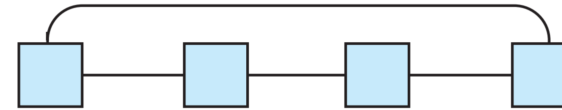
Interconnection Network Architectures

- **Bus**. System components send data on and receive data from a single communication bus;
  - Does not scale well with increasing parallelism.
- A **ring** network is a network topology in which each node connects to exactly two other nodes
- **Mesh**. Components are arranged as nodes in a grid, and each component is connected to all adjacent components
- **Hypercube**. Components are numbered in binary; components are connected to one another if their binary representations differ in exactly one bit.
- **Tree-like Topology**. Widely used in data centers today

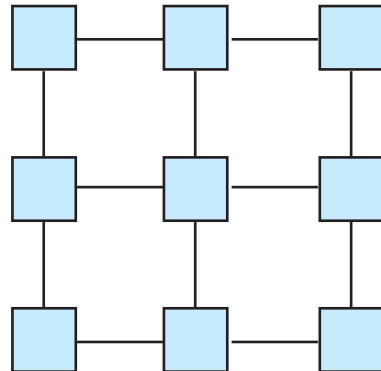
Interconnection Architectures



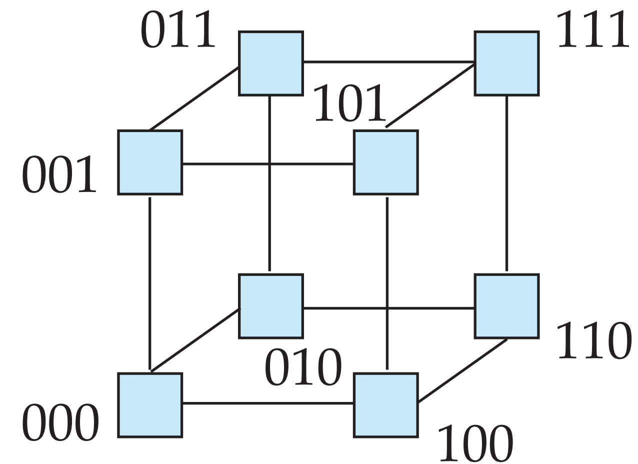
(a) bus



(b) ring



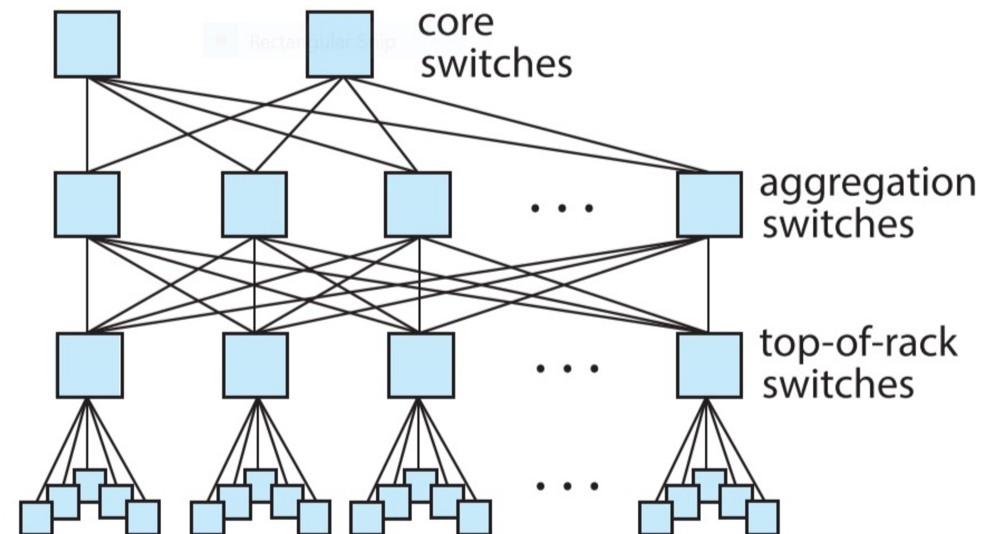
(c) mesh



(d) hypercube

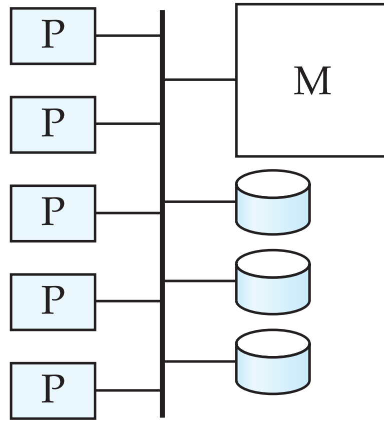
Interconnection Network Architectures

- Tree-like or Fat-Tree Topology: widely used in data centers today
  - Top of rack switch for approx 40 machines in rack
  - Each top of rack switch connected to multiple aggregation switches.
  - Aggregation switches connect to multiple core switches.
- Data center fabric

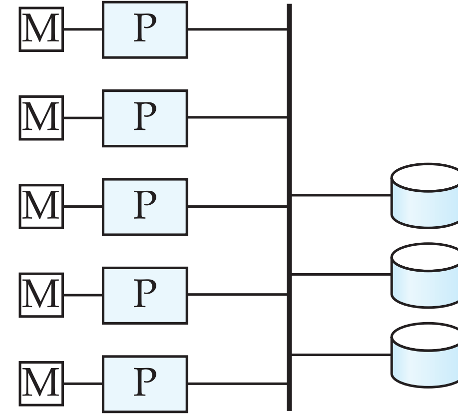


(e) tree-like topology

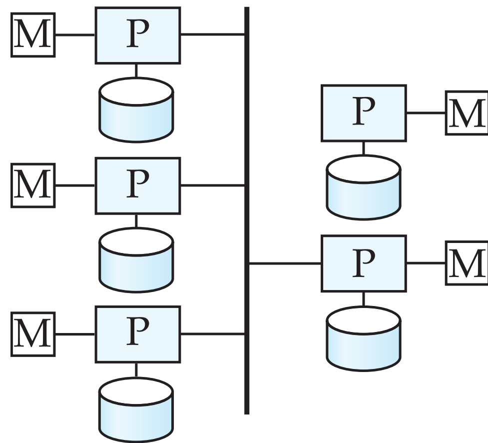
Parallel Database Architectures



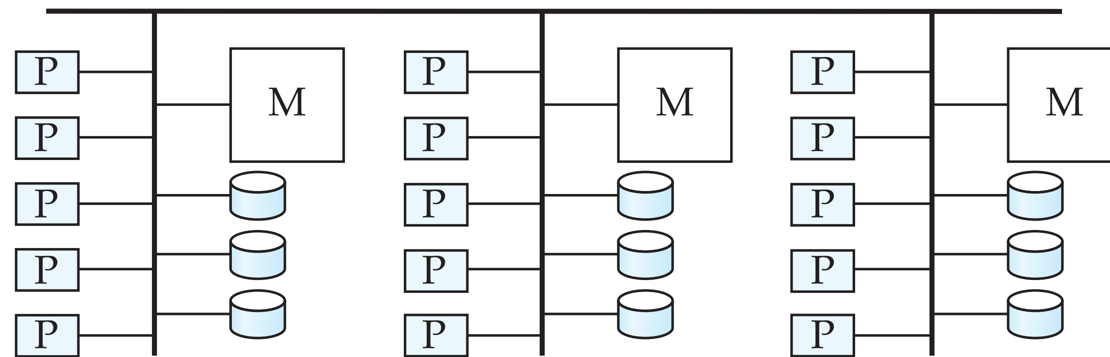
(a) shared memory



(b) shared disk



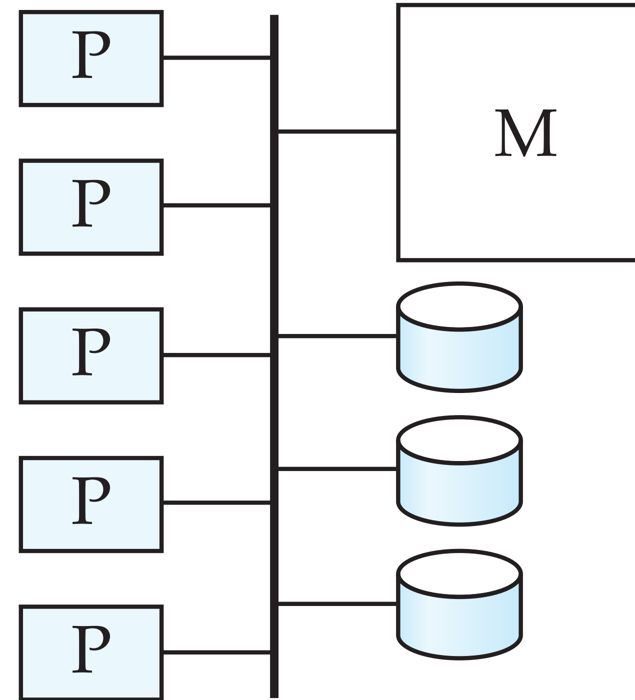
(c) shared nothing



(d) hierarchical

Shared Memory

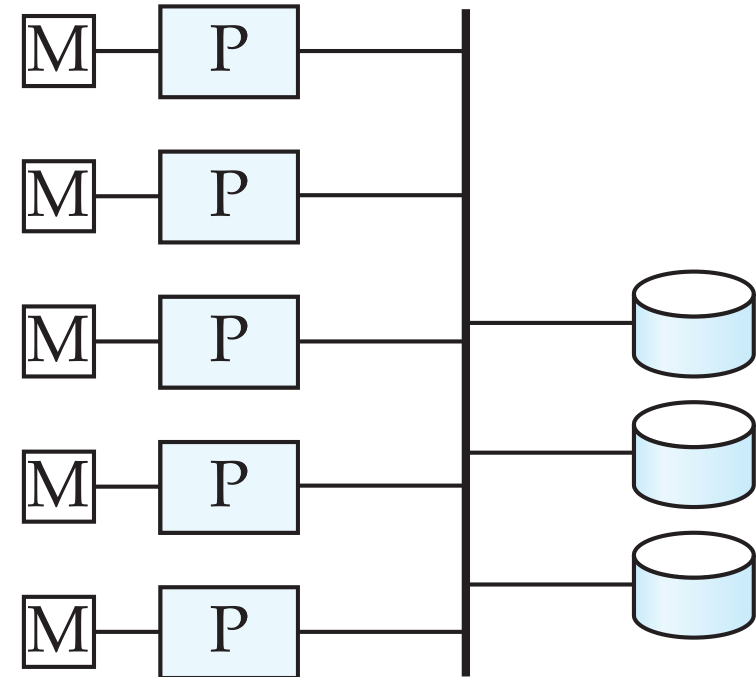
- Processors (or processor cores) and disks have access to a common memory
  - Via a bus in earlier days, through an interconnection network today
- Extremely efficient communication between processors
- Downside: shared-memory architecture is not scalable beyond 64 to 128 processor cores
  - Memory interconnection network becomes a bottleneck



(a) shared memory

Shared Disk

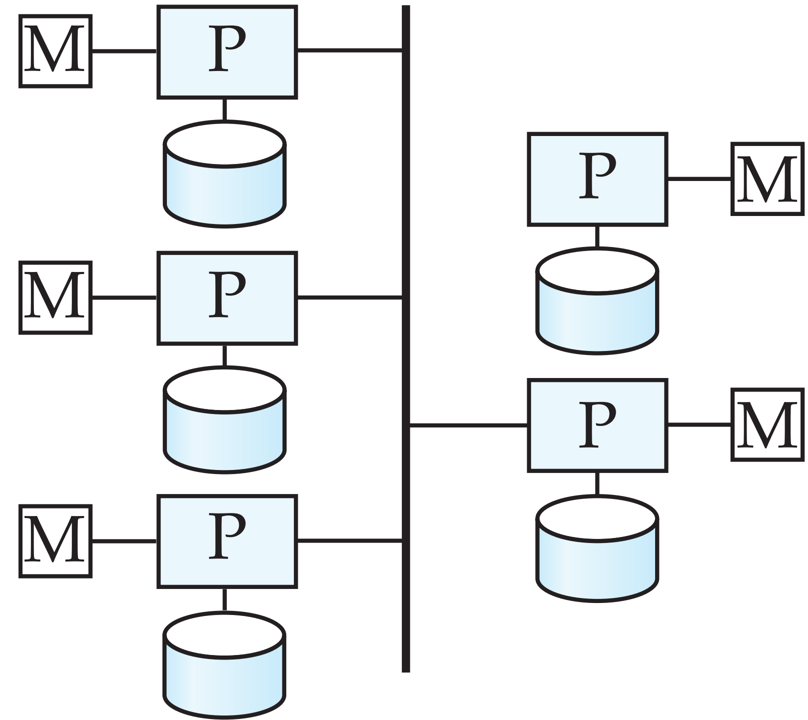
- All processors can directly access all disks via an interconnection network, but the processors have private memories.
  - Architecture provides a degree of fault-tolerance — if a processor fails, the other processors can take over its tasks
    - the data of the failed processor is resident on disks that are accessible from all processors.
- Downside: bottleneck now occurs at interconnection to the disk subsystem.



(b) shared disk

Shared Nothing

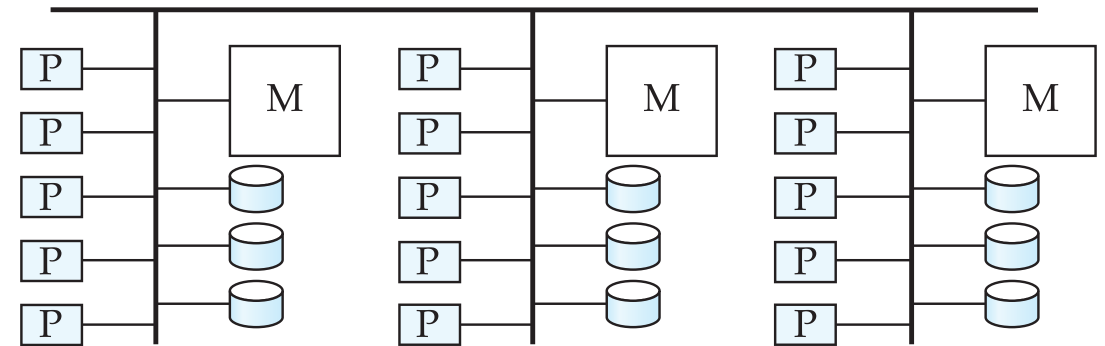
- Node consists of a processor, memory, and one or more disks
- All communication via interconnection network
- Can be scaled up to thousands of processors without interference.
- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.



(c) shared nothing

Hierarchical

- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.
  - Top level is a shared-nothing architecture
    - With each node of the system being a shared-memory system
  - Alternatively, top level could be a shared-disk system
    - With each node of the system being a shared-memory system

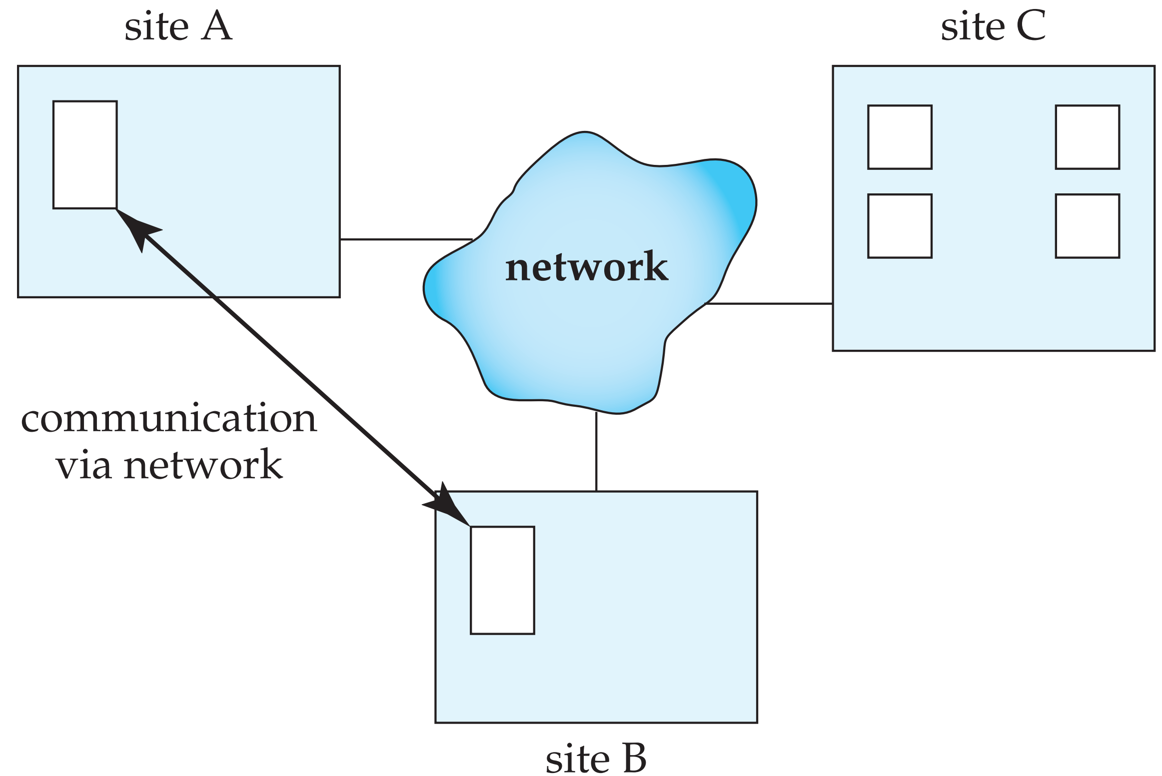


(d) hierarchical

DISTRIBUTED DATABASES

Distributed Systems

- Data spread over multiple machines (also referred to as sites or nodes).
- Local-area networks (LANs)
- Wide-area networks (WANs)
  - Higher latency



Distributed vs Parallel

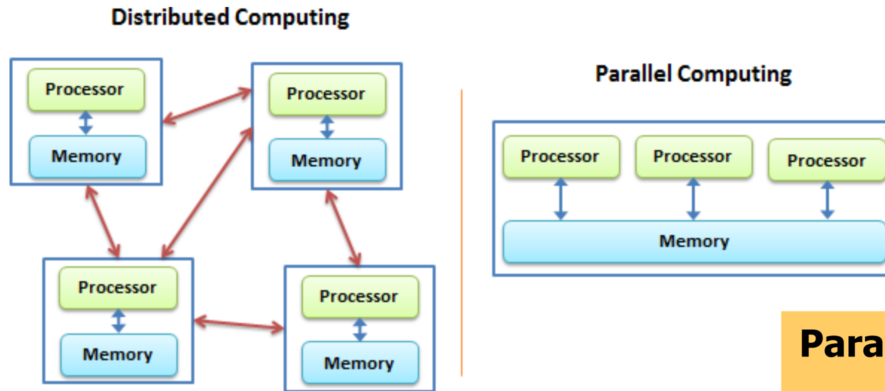


Image: https://www.oreilly.com/library/view/distributed-computing-in/9781787126992/7478b64c-8de4-4db3-b473-66e1d1fcba77.xhtml

Parallel	Distributed
Many operations are performed simultaneously	System components are located at different locations
Single computer	Multiple computers
Multiple processors perform multiple operations	Multiple computers perform multiple operations
It may have shared or distributed memory	It have only distributed memory
Processors communicate with each other through bus	Computer communicate with each other through message passing.
Improves the system performance	Improves system scalability, fault tolerance and resource sharing capabilities

https://www.geeksforgeeks.org/difference-between-parallel-computing-and-distributed-computing/

Distributed Databases

- **Homogeneous** distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution
- **Heterogeneous** distributed databases
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality
- Differentiate between local transactions and global transactions
  - A local transaction accesses data in the single site at which the transaction was initiated.
  - A global transaction either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.

Data Integration and Distributed Databases

- Data integration between multiple distributed databases
- Benefits:
  - Sharing data – users at one site able to access the data residing at some other sites.
  - Autonomy – each site is able to retain a degree of control over data stored locally.

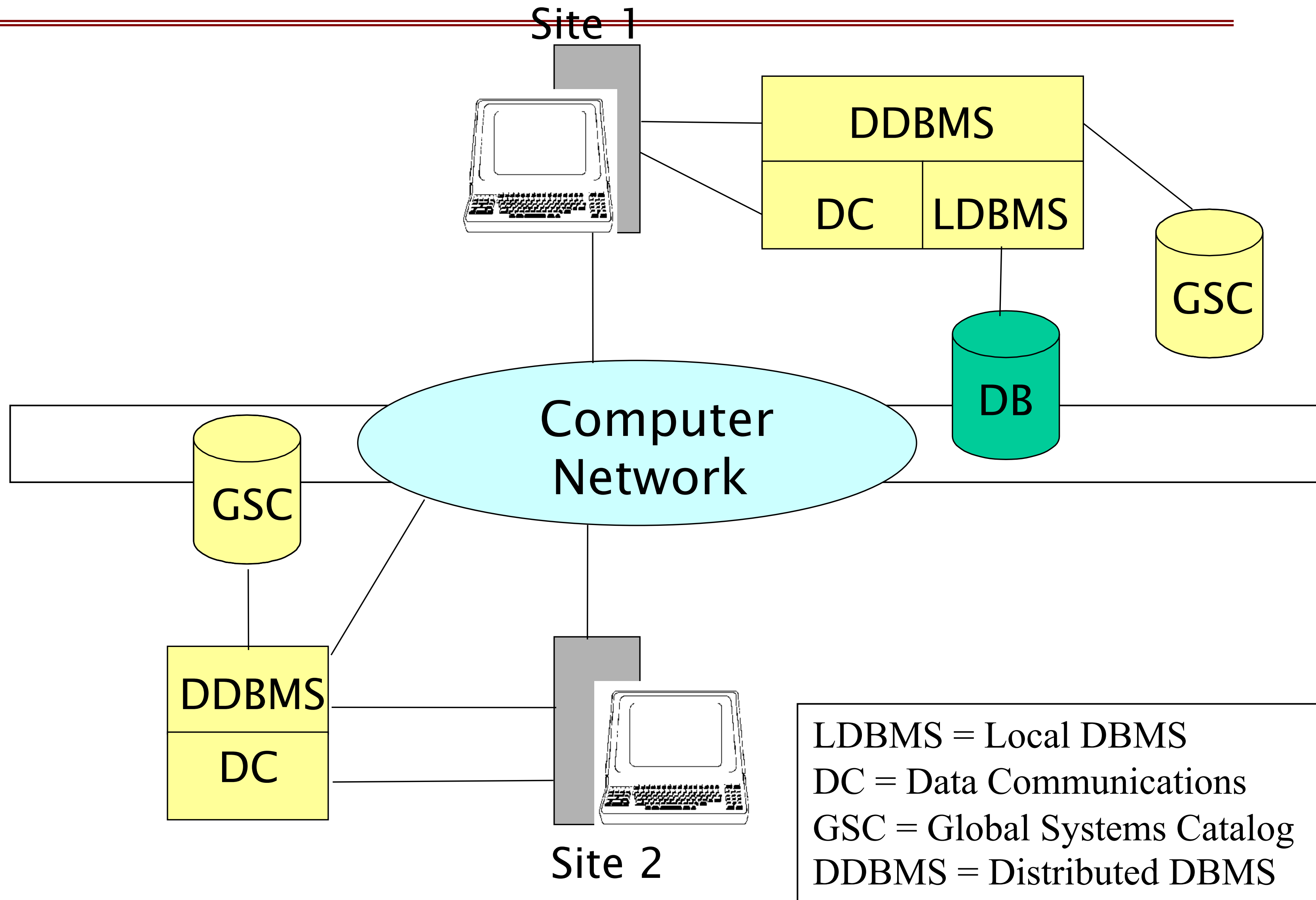
Availability

- Network partitioning
- Availability of system
  - If all nodes are required for system to function, failure of even one node stops system functioning.
  - Higher system availability through redundancy
    - data can be replicated at remote sites, and system can function even if a site fails.

Implementation Issues for Distributed Databases

- Atomicity needed even for transactions that update data at multiple sites
- The two-phase commit protocol (2PC) is used to ensure atomicity
  - Basic idea: each site executes transaction until just before commit, and then leaves final decision to a coordinator
  - Each site must follow decision of coordinator, even if there is a failure while waiting for coordinators decision
- Distributed concurrency control (and deadlock detection) required
- Data items may be replicated to improve data availability

COMPONENTS OF A DDBMS



ISSUES

- Data Partitioning
  - *How are data partitioned in nodes?*
- Data Replication
  - *Where are data located?*
- Catalog Management
  - *Where does the DB catalog reside?*
- Distributed Transactions
  - *How do transactions commit changes in multiple nodes?*
- Distributed Queries
  - *How are Queries executed over multiple nodes?*

WHY PARTITIONING DATA?

- Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks, on multiple nodes (computers)
  - Partitioning across nodes
  - Same techniques can be used across disks on a node
- **Partitioning methods**
  - Horizontal : different rows of a tables into different nodes .
  - Vertical: different columns into different nodes.

HORIZONTAL DATA PARTITIONING

ACCOUNT	CUSTOMER	BRANCH	BALANCE
200	JONES	Athens	1000.00
324	GRAY	London	200.00
345	SMITH	Athens	23.17
350	GREEN	London	340.14
400	ONO	London	500.00
456	KHAN	Athens	333.00

Horizontal Partitioning: Consists of a Restriction on a Relation.

e.g., $(\sigma_{\text{branch = 'Athens'}}\ \text{Account})$

HORIZONTAL DATA PARTITIONING

Athens BRANCH

ACCT NO.	CUSTOMER	BRANCH	BALANCE
200	JONES	Athens	1000.00
345	SMITH	Athens	23.17
456	KHAN	Athens	333.00

London BRANCH

ACCT NO.	CUSTOMER	BRANCH	BALANCE
324	GRAY	London	200.00
350	GREEN	London	340.14
400	ONO	London	500.00

HOW TO PARTITION?

- Round-robin:
  - Send the ***ith tuple*** inserted in the relation to node (***i mod n***).
- Hash partitioning:
  - Choose one or more attributes as the partitioning key.
  - Choose hash function ***h*** with range ***0…n - 1***
  - Let ***i*** denote result of hash function ***h*** applied to the partitioning key of a tuple. Send tuple to node ***i***.
- Range Partitioning :
  - Choose an attribute as the partitioning key.
  - Select a partition by determining if the partitioning key is within a certain range.
- List partitioning:
  - A partition is assigned a list of values.
    - E.g., Greece, Italy, Germany
  - If the partitioning key has one of these values, the partition is chosen.

VERTICAL DATA PARTITIONING

S#	NAME	SITE	PHONE NO	LOGIN	PASSWORD
200	JONES	Athens	0208-500-9000	JON200T	XXYY22
324	GRAY	London	0208-545-7528	GRA324S	ZZEE56
456	KHAN	Athens	0208-500-5821	KHA456T	KJTR78

Vertical Partitioning: Consists of a Projection on a Relation.

e.g., $(\Pi_{\text{S\#, NAME, SITE, PHONE NO}} \text{ Student})$

Remember how Column Stores store the data!!!

VERTICAL DATA PARTITIONING

Customer ADMINISTRATION

S#	NAME	SITE	PHONE NO.
200	JONES	Athens	0208-500-9000
324	GRAY	London	0208-545-7528
456	KHAN	Athens	0208-500-5821

System ADMINISTRATION

S#	LOGIN-ID	PASSWORD
200	JON200T	XXYY22
324	GRA324S	ZZEE56
456	KHA456T	KJTR78

Replication

- Goal: availability despite failures
- Data replicated at 2, often 3 nodes
- Unit of replication typically a **partition** (tablet)
- Requests for data at failed node automatically routed to a replica
- Partition table with each tablet replicated at two nodes

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

DISTRIBUTED CATALOGUE MANAGEMENT

Centralised Global Catalogue

- One site maintains the full global catalogue.
- All changes to any local system catalogue are propagated to the site maintaining the global catalogue.
- Bad performance, single point of failure, compromises site autonomy.

Distributed Catalogue

- There is no physical global catalogue. Each time a remote data item is required, the catalogues from ALL other sites are examined for the item.
- Severe performance penalties.

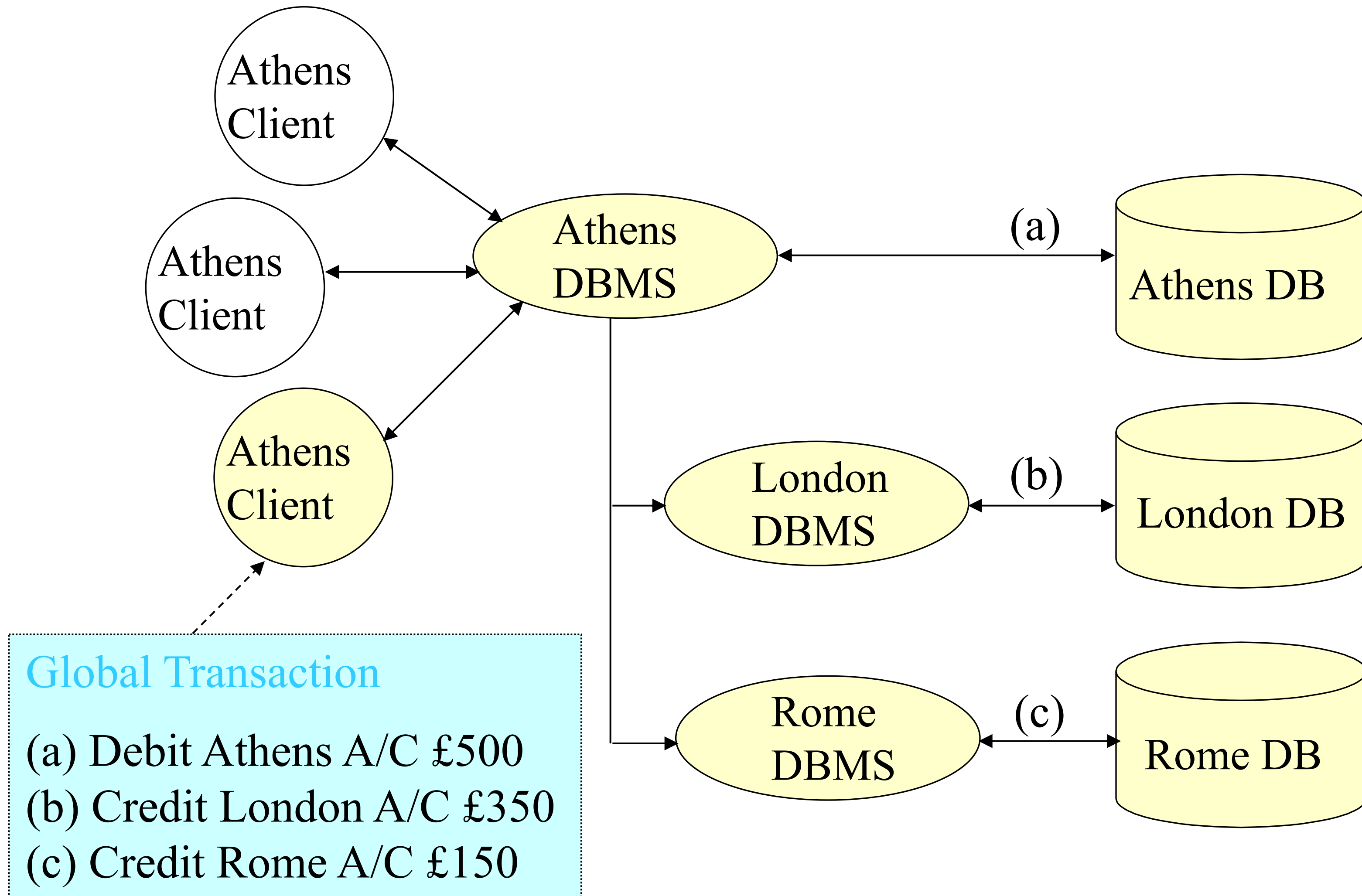
Replicated Global Catalogue

- Each site maintains its own global catalogue.
- Although this greatly speeds up remote data location, it is very inefficient to maintain. A detail of every data item added, changed or deleted locally has to be propagated to ALL other sites .

Local-Master Catalogue

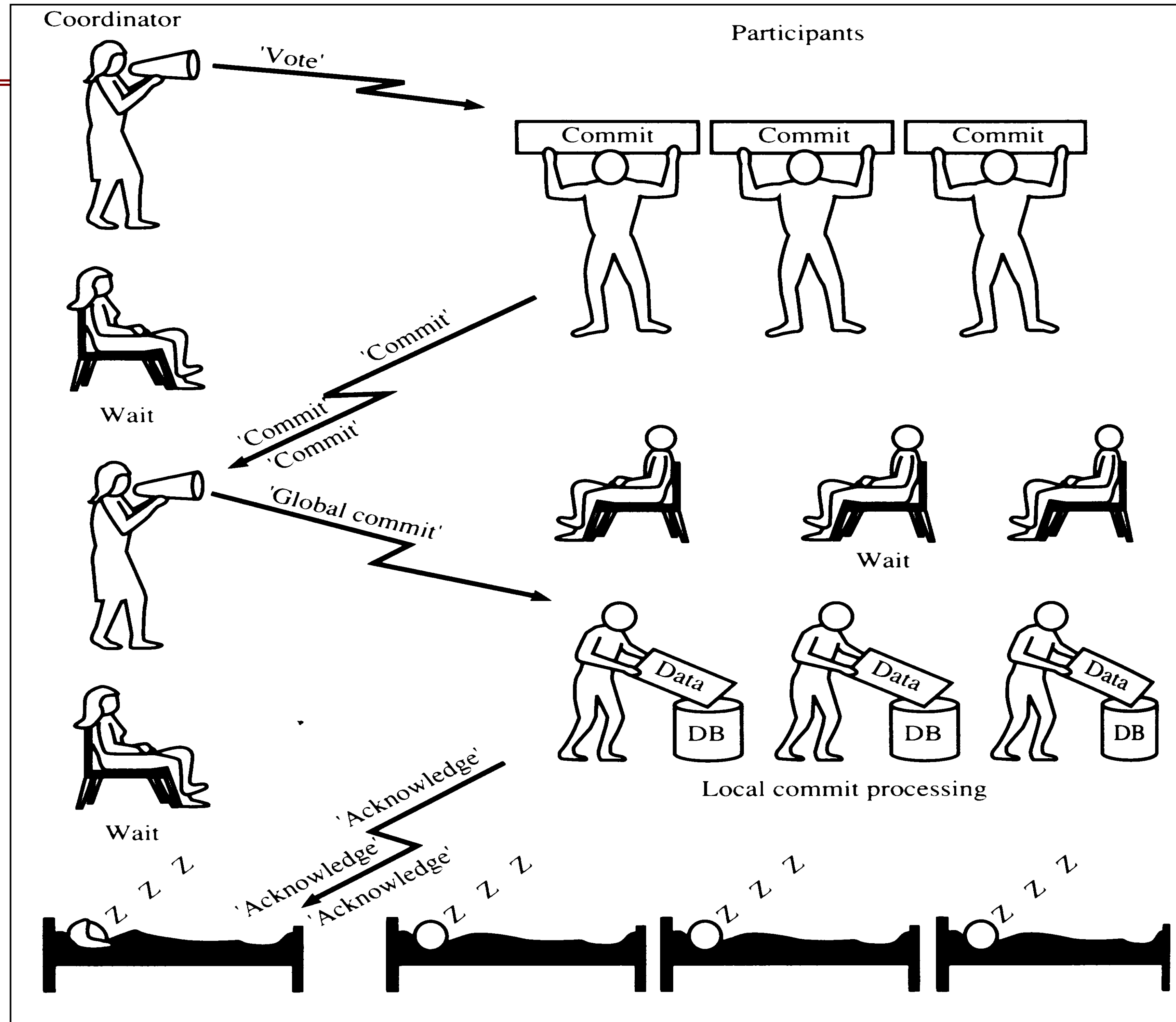
- Each site maintains both its local system catalogue as well as a catalogue of all of its data items that are replicated at other sites.
- Avoids compromising site autonomy, is fairly efficient, and is not a single point of failure

DISTRIBUTED TRANSACTIONS

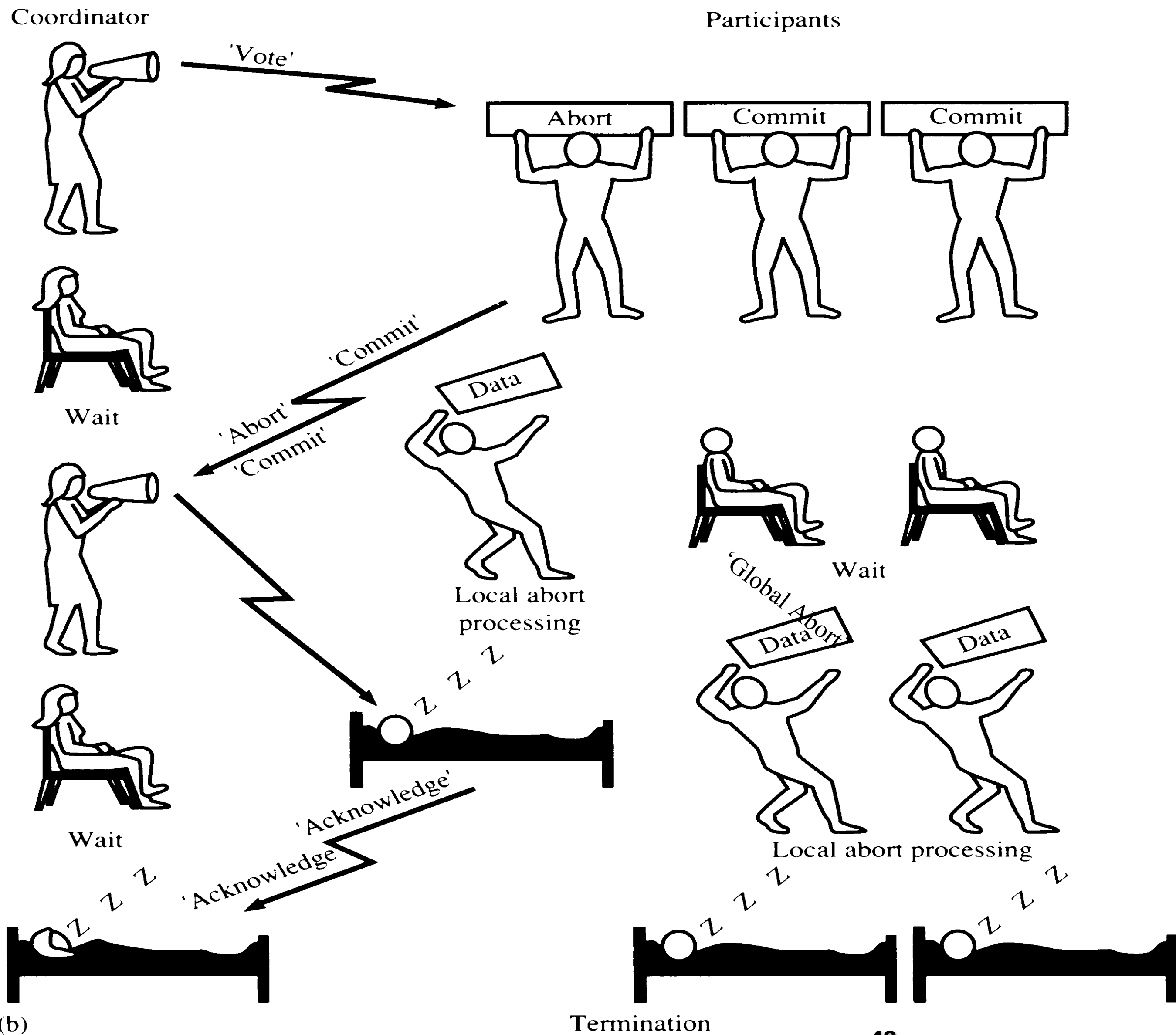


ATOMIC DISTRIBUTED TRANSACTION

TWO-PHASE COMMIT (2PC) - OK



TWO-PHASE COMMIT (2PC) - ABORT



2PC is not always appropriate:
3-phase commits and other transaction models based on **persistent messaging**, and **workflows**, are also used

Parallel (&Distributed) Query Processing

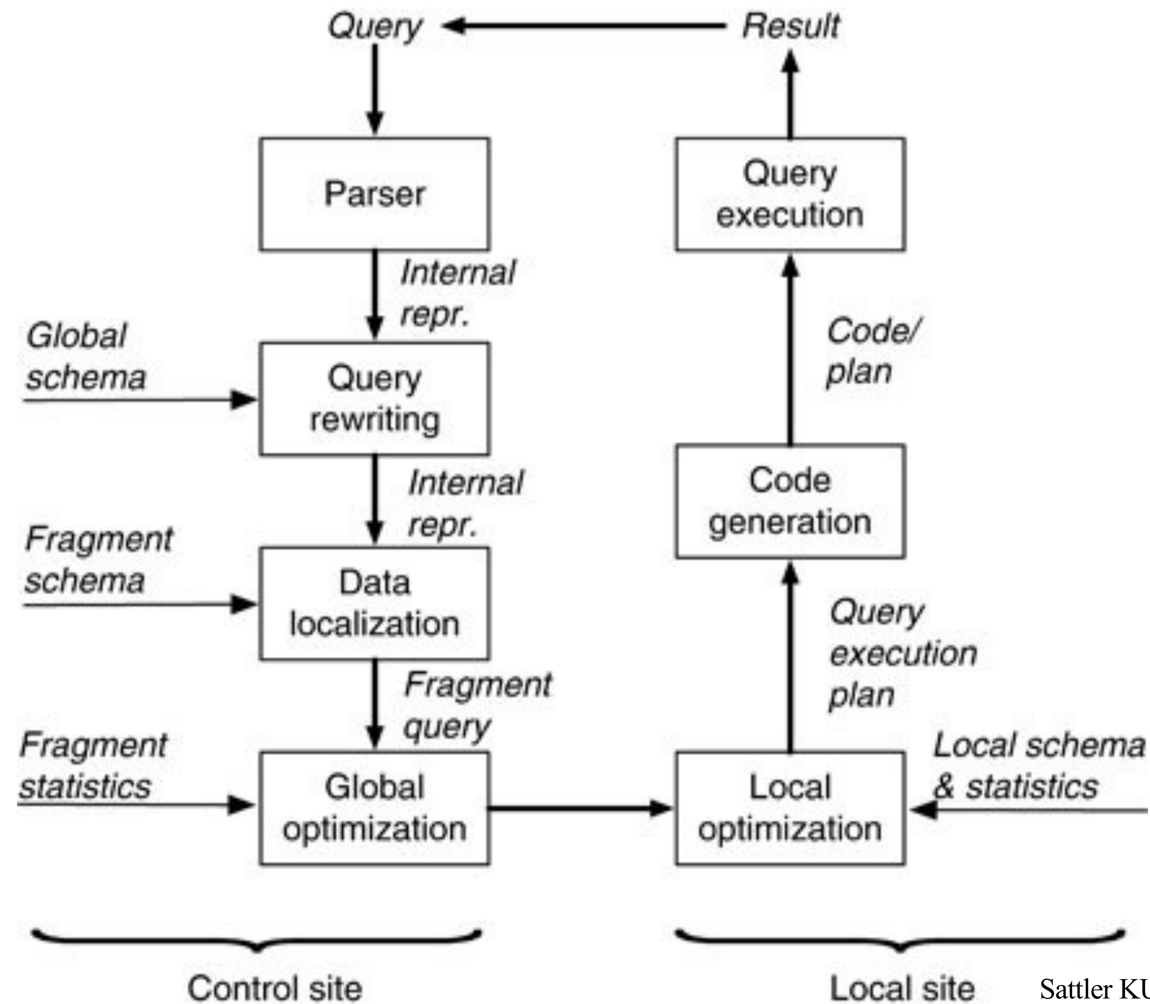
INTERQUERY PARALLELISM

- It is a form of parallelism where many different Queries or Transactions are executed in parallel with one another on many processors (nodes).

INTRAQUERY PARALLELISM

- It is the form of parallelism where a Single Query is broken in ‘sub-tasks’ and executed in parallel on many processors (nodes).

Distributed Query Processing - Plan



Sattler KU. (2009) Distributed Query Processing. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-39940-9_704

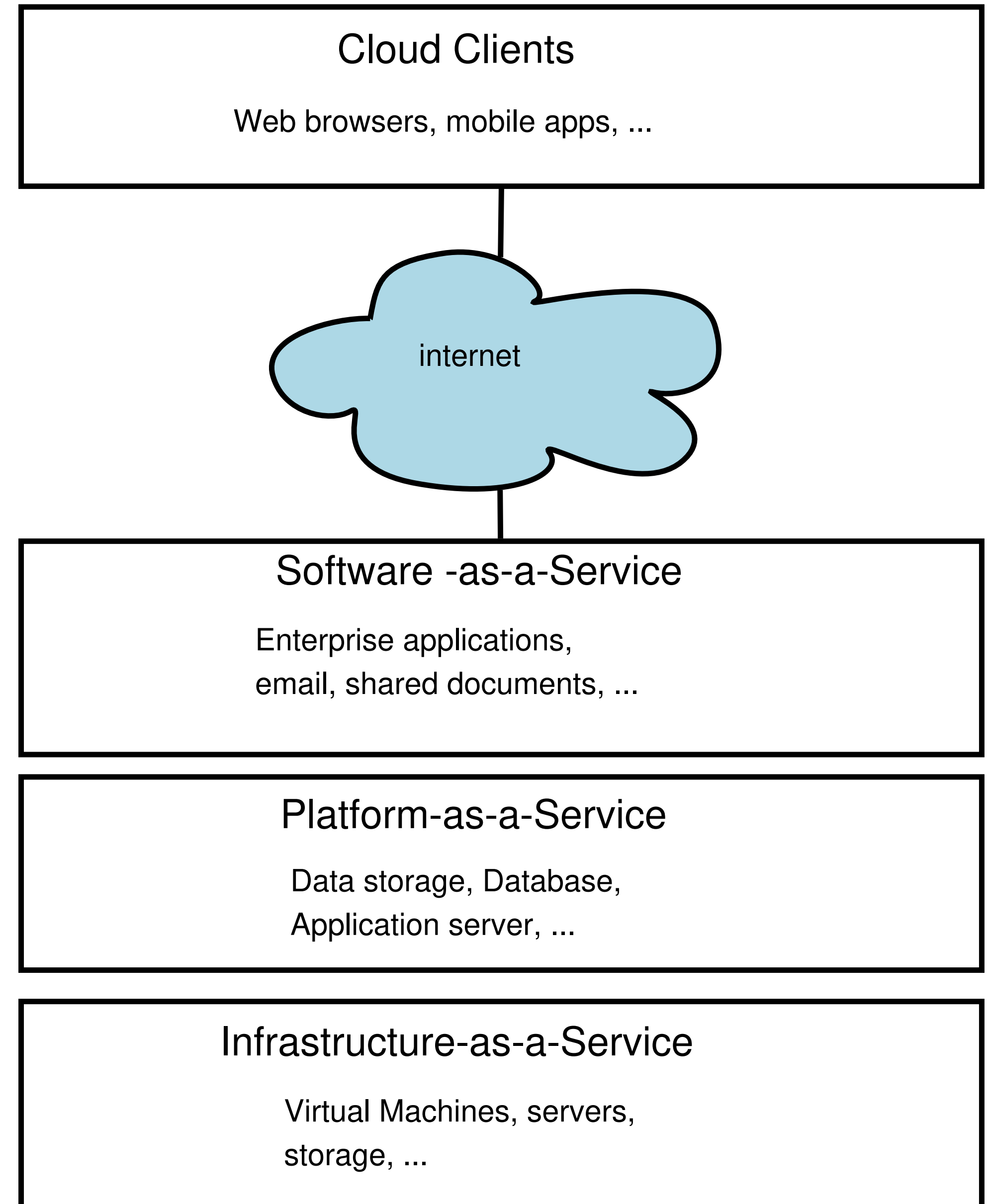
DISADVANTAGES OF DDBMSs

- ☹ Architectural complexity.
- ☹ Cost.
- ☹ Security.
- ☹ Integrity control more difficult.
- ☹ Lack of standards.
- ☹ Lack of experience.
- ☹ Database design more complex.

CLOUD DATABASES

Cloud Service Models

- On-demand provisioning and elasticity
  - ability to **scale out** at short notice and to release of unused resources for use by others



Cloud Databases

- A cloud database is a database that typically runs on a cloud computing platform and access to the database is provided as-a-service.
- Two deployment models:
  - Databases is deployed on a Virtual Machine - DB is maintained by the user independently
  - Database-as-a-service (DBaaS) : access to a database service, maintained by a cloud database provider

Scale up, scale out, scale in

- Scale up: Big iron
- Scale out: Commodity hardware
- "Scale in": Multiple apps / tenants / VMs on single machine

Felix Naumann Slides
https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/folien/WS0910/DBS_II/DBS2_12_WDM_MultiTenancy_FelixNaumann.pdf

DBaaS - Multi-Tenancy

- Run application for multiple clients („tenants“) on a **shared** database
  - Only makes sense if enough tenants can be served from one database
- Resource utilization increased, fewer/smaller hardware required
- Fewer processes/machines to manage

Analogies for Multi-Tenancy in Real Life

- Multiple clients hosted by one service provider
  - Multiple tenants hosted in one building complex
- Code (to be executed)
  - Utilities (gas, water, electricity, waste)
- Data (with services)
  - Storage space (furniture, basement, garage)

Felix Naumann Slides
https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/folien/WS0910/DBS_II/DBS2_12_WDM_MultiTenancy_FelixNaumann.pdf

Back to SaaS: Implementation Options

- Four analogies…
  - Dedicated Virtual Machine (house)
  - Shared machine (hotel room)
  - Shared Process (apartment)
  - Shared Table (youth hostel)

Felix Naumann Slides
https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/folien/WS0910/DBS_II/DBS2_12_WDM_MultiTenancy_FelixNaumann.pdf

Shared Machine

- One dedicated database process for each tenant
- Does not scale beyond a few tenants per server (overhead!)
- Applications with small number of tenants

Shared Process

- Single database process manages several tenants (in separate table spaces)
- Scales to many tenants (regarding overhead)
- Migration of a tenant to a different process is simple (moving all data files)

Shared Table

- Data from many tenants in the same table
  - Add tenant_id column, must be set for each access (by application or by dbms)
- But: sometimes tenants require additional columns
  - Extend schema with generic columns; database needs to efficiently support rows with many NULL values
- Advantage: everything is pooled
  - Processes, memory, connections, prepared statements
  - New tenants can be created by DML (not DDL) operations
- Disadvantage: isolation is very weak
  - Query optimization, statistics, cost estimation more difficult (data from other tenants can influence estimates)
  - Table scans more expensive, caching etc not as effective
  - Tenant migration by extracting data from the operational system

Schema Flexibility in SaaS

- Each tenant uses common base schema plus optional (private or shared) extensions
- All mapped into one (multi-tenant) physical schema by a dedicated Mapper in the database
- Schema evolution must be possible during operation, without intervention by DBA

Schema Management

Two alternatives:

- Database manages the schema (evolution through DDL operations):
  - Private tables
  - Extension tables
  - Sparse columns
- Application manages the schema (evolution through DML operations)
  - XML Columns
  - Universal tables
  - Pivot tables

Running example:
simple account table

Account	
Account	Name

Private Tables

- Each tenant uses private set of tables

Account_1

Account	Name	Hospital	Beds
1	Acme	St Mary	135
2	Gump	State	1042

Healthcare Domain

Account_2

Account	Name	Dealers
1	Big	65

Automotive Domain

Account_3

Account	Name
1	Ball

- Works well for small number of tables and tenants
  - Constant overhead per table
  - each table is small, storage space not used effectively

Private Tables – Query Transformation

- Tenant 1:

```
SELECT Beds FROM Account
WHERE Hospital='State'
```

```
⇒SELECT Beds FROM Account_1
  WHERE Hospital='State'
```

Account_1

Account	Name	Hospital	Beds
1	Acme	St Mary	135
2	Gump	State	1042

- Tenant 2:

```
SELECT Name FROM Account
WHERE Dealers>50
```

```
⇒SELECT Name FROM Account_2
  WHERE Dealers>50
```

Account_2

Account	Name	Dealers
1	Big	65

Extension Tables

- Keep common data in base table with explicit Tenant-ID and row-ID
- Each tenant‘s extension gets its own table, join on tenant-ID and row-ID at query time
- Number of table still proportional to number of tenants (but additional tables are rather small)

Account

Tenant	Row	Account	Name
1	1	1	Acme
1	2	2	Gump
2	1	1	Big
3	1	1	Ball

Account_Healthcare

Tenant	Row	Hospital	Beds
1	1	St Mary	135
1	2	State	1042

Account_Automotive

Tenant	Row	Dealers
2	1	65

Extension Tables – Query Transformation

- Tenant 1:

```
SELECT Name,Beds
FROM Account
WHERE Hospital='State'
```

```
⇒SELECT A.Name,
        H.Beds
  FROM Account A,
       Account_Healthcare H
  WHERE A.Tenant=1
    AND H.Tenant=1
    AND A.Row=H.Row
    AND H.Hospital='State'
```

Account

Tenant	Row	Account	Name
1	1	1	Acme
1	2	2	Gump
2	1	1	Big
3	1	1	Ball

Account_Healthcare

Tenant	Row	Hospital	Beds
1	1	St Mary	135
1	2	State	1042

Sparse Columns

- Each tuple has only a few out of many possible attributes (e.g., catalogues) $\Rightarrow$ many NULL values
- Store only attributes with values to avoid NULLs
  - store values with their attribute (identifier)
  - not widely supported in systems (Microsoft SQL Server)
- Add extensions as „sparse column“ to tables

Account

Tenant	Account	Name	SPARSE
1	1	Acme	0:St Mary, 1:135
1	2	Gump	0:State, 1:1042
2	1	Big	2:65
3	1	Ball	

Sparse Columns – Query Transformation

- ```
  CREATE TABLE Account (
  Tenant INT, Account INT, Name VARCHAR(100),
  Hospital VARCHAR(100) SPARSE,
  Beds INT SPARSE, Dealer INT SPARSE
  )
  ```

- Tenant 1:

  ```
  SELECT Name, Beds
  FROM Account
  WHERE Tenant=1
    AND Hospital='State'
  ```

System retrieves attribute ID and extracts values from SPARSE column

Account

Tenant	Account	Name	SPARSE
1	1	Acme	0:St Mary, 1:135
1	2	Gump	0:State, 1:1042
2	1	Big	2:65
3	1	Ball	
  ```

XML Columns

- Each table contains additional XML column for storing extensions as XML document

Account

Tenant	Account	Name	XMLData
1	1	Acme	<data> <hospital>St Mary</hospital> <bed>135</bed> </data>
1	2	Gump	<data> <hospital>State</hospital> <bed>1042</bed> </data>
2	1	Big	<data> <dealers>65</dealers> </data>
3	1	Ball	NULL

XML Columns – Query Transformation

- Tenant 1:

```
SELECT Name,Beds
FROM Account
WHERE Hospital=‘State‘
```

```
⇒SELECT NAME,
xml([data/bed])
FROM Account
WHERE Tenant=1
AND xmlexists(
‘$x[data/hospital=‘State‘]
PASSING XMLData as $x)
```

Account

Tenant	Account	Name	XMLData
1	1	Acme	<data> <hospital>St Mary</hospital> <bed>135</bed> </data>
1	2	Gump	<data> <hospital>State</hospital> <bed>1042</bed> </data>
2	1	Big	<data> <dealers>65</dealers> </data>
3	1	Ball	NULL

Universal Table

- Use wide table with generic VARCHAR columns
  - Application-specific mapping
  - Requires casting of non-textual values
  - Many NULL values, usually no indexes possible
- Used with huge number of extensions and many tenants

Universe

Tenant	Table	Col1	Col2	Col3	Col4	…	Col999
1	1	1	Acme	St Mary	135	…	NULL
1	1	2	Gump	State	1042	…	NULL
2	1	1	Big	65	NULL	…	NULL
3	1	1	Ball	NULL	NULL	…	NULL

Universal Table – Query Transformation

- Tenant 1:

```
SELECT Name,Beds
FROM Account
WHERE Hospital='State'

⇒SELECT Col2, TO_INTEGER(Col4)
  FROM Universe
  WHERE Tenant=1
     AND Table=1
    AND Col3='State'
```

Universe

Tenant	Table	Col1	Col2	Col3	Col4	…	Col999
1	1	1	Acme	St Mary	135	…	NULL
1	1	2	Gump	State	1042	…	NULL
2	1	1	Big	65	NULL	…	NULL
3	1	1	Ball	NULL	NULL	…	NULL

Pivot Tables

- Store data in 3-ary tables with column_ids and values
  - One tuple for each non-NULL attribute of original table
  - One pivot table for each type (int, string, …)
  - Eliminates the problem of many NULL values
  - No casts necessary, indexing possible

Pivot_String

Tenant	Table	Row	Col	String
1	1	1	2	Acme
1	1	1	3	St Mary
1	1	2	2	Gump
1	1	2	3	State
2	1	1	2	Big
3	1	1	2	Ball

Pivot_Int

Tenant	Table	Row	Col	Int
1	1	1	1	1
1	1	1	4	135
1	1	2	1	2
1	1	2	4	1042
2	1	1	1	1
2	1	1	3	65
3	1	1	1	1

Pivot Tables – Query Transformation

- Tenant 1:

```
SELECT Name,Beds
FROM Account
WHERE Hospital='State'
```

```
⇒SELECT S1.String,I.Int
FROM Pivot_Int I,
     Pivot_String S1,
     Pivot_String S2

WHERE I.Tenant=1
  AND S1.Tenant=1
  AND S2.Tenant=1
  AND S1.Table=1 AND S1.Col=2
  AND S2.Table=1 AND S2.Col=3
  AND I.Table=1 AND I.Col=4
  AND I.Row=S2.Row
  AND S1.Row=S2.Row
  AND S2.String='State'
```

Pivot_Int

Tenant	Table	Row	Col	Int
1	1	1	1	1
1	1	1	4	135
1	1	2	1	2
1	1	2	4	1042
2	1	1	1	1
2	1	1	3	65
3	1	1	1	1

Pivot_String

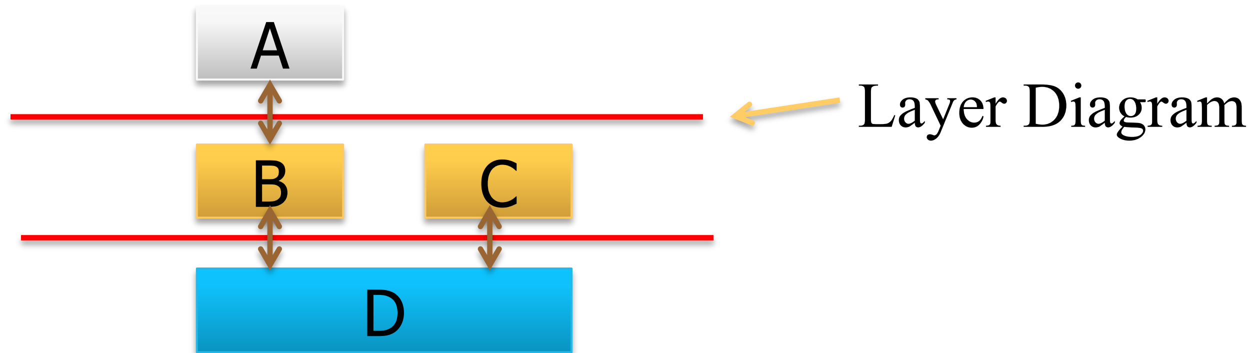
Tenant	Table	Row	Col	String
1	1	1	2	Acme
1	1	1	3	St Mary
1	1	2	2	Gump
1	1	2	3	State
2	1	1	2	Big
3	1	1	2	Ball

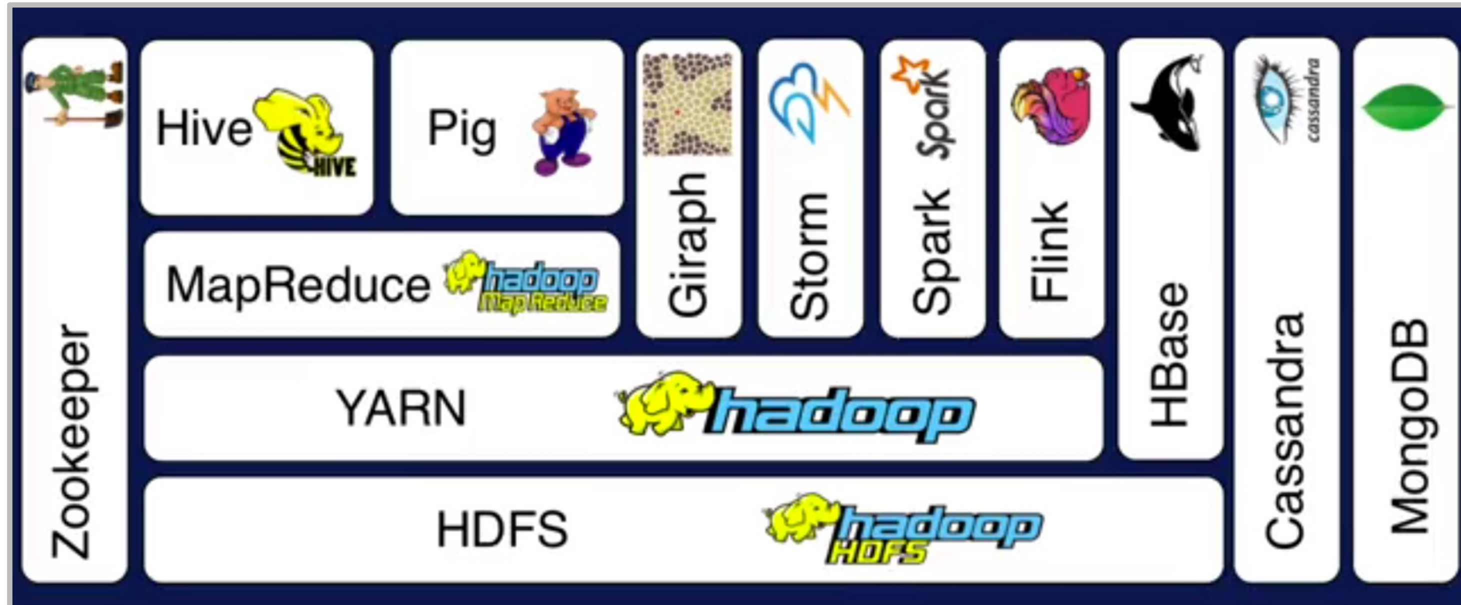
Summary: DB in the Cloud

- New **paradigm** to provide services (S,P,I)
- **SaaS** important part of cloud computing
- Different forms of **scaling**: up, out, in
- Re-use processes: **multiple tenants** on single database instance (but: isolation?)
- **Map** many **logical** schemas to a **single physical** schema, many mapping techniques (but: query transformation?)

APACHE HADOOP ECOSYSTEM

Hadoop Ecosystem





Hadoop HDFS

- Hadoop distributed File System (based on Google File System (GFS) paper, 2004)
  - Serves as the distributed file system for most tools in the Hadoop ecosystem
  - Scalability for large data sets
  - Reliability to cope with hardware failures
- HDFS good for:
  - Large files
  - Streaming data
- Not good for:
  - Lots of small files
  - Random access to files
  - Low latency access

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

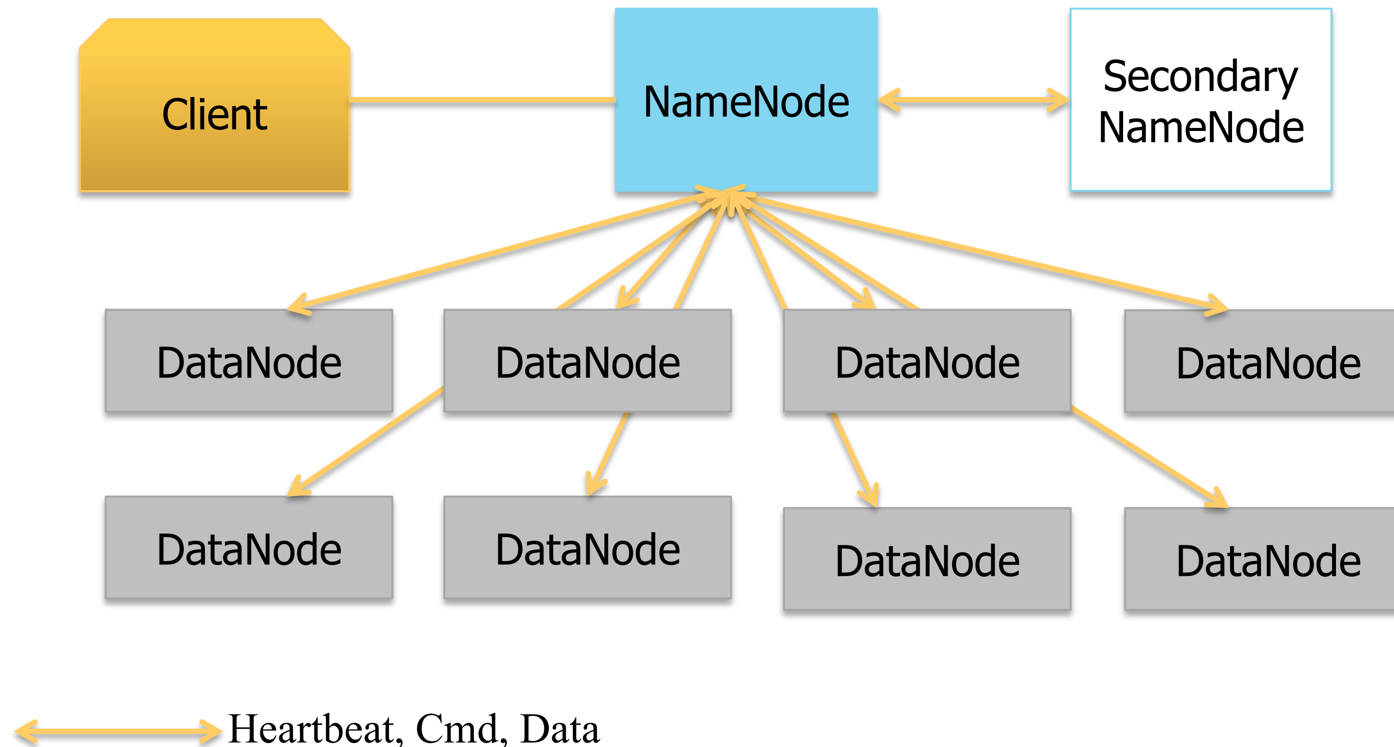
First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the

Design of Hadoop Distributed File System (HDFS)

- Master-Slave design
- Master Node
  - Single NameNode for managing metadata
- Slave Nodes
  - Multiple DataNodes for storing data
- Other
  - Secondary NameNode as a backup

HDFS Architecture

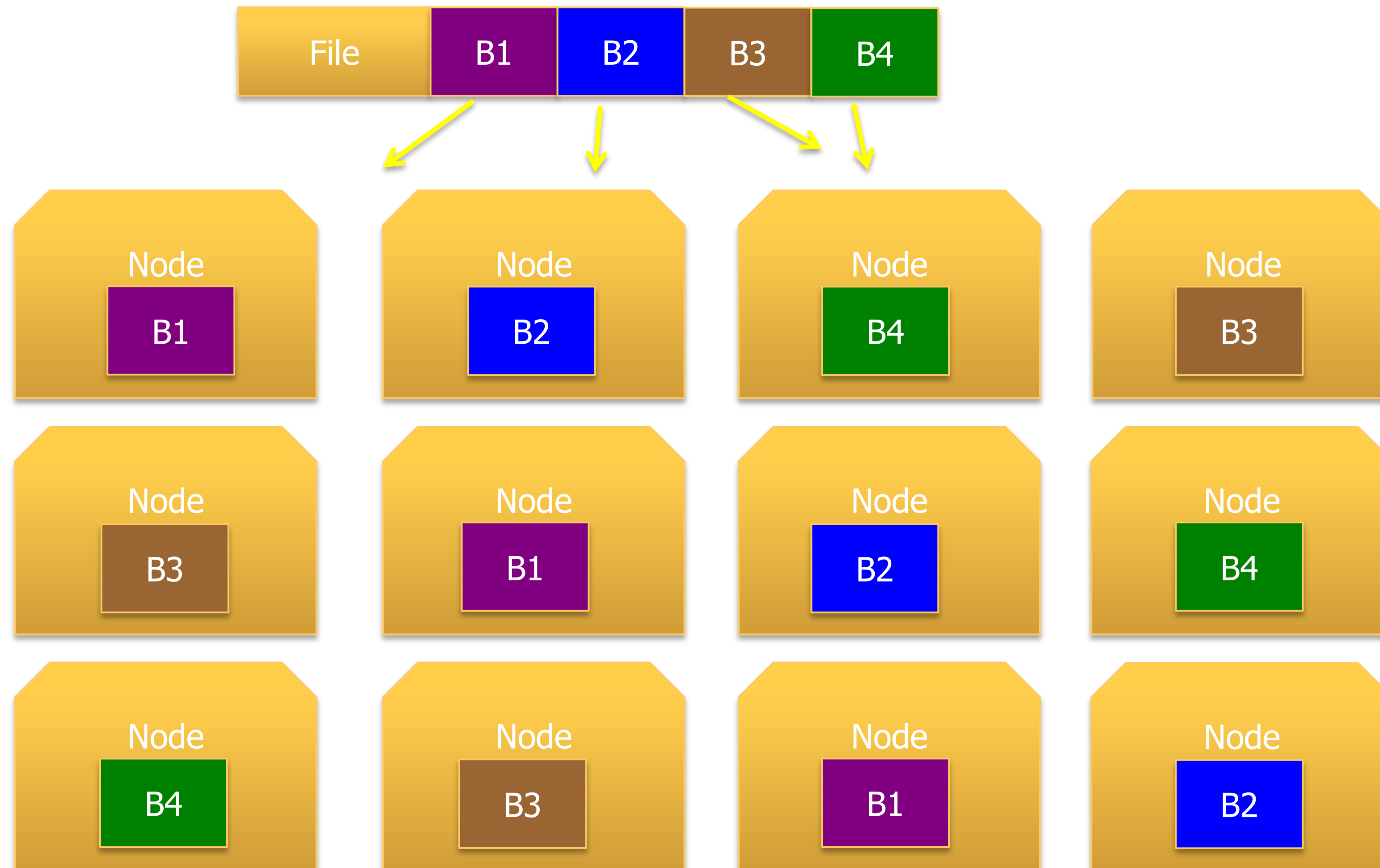
NameNode keeps the metadata, the name, location and directory
DataNode provide storage for blocks of data



Heartbeat, Cmd, Data

HDFS

What happens; if node(s) fail?
Replication of Blocks for fault tolerance



HDFS

- HDFS files are divided into blocks
  - It’s the basic unit of read/write
  - Default size is 64MB, could be larger (128MB)
  - Hence makes HDFS good for storing larger files
- HDFS blocks are **replicated** multiple times
  - One block stored at multiple locations, also at different racks (usually 3 times)
  - This makes HDFS storage fault tolerant and faster to read

Few HDFS Shell commands

Create a directory in HDFS

- `hadoop fs -mkdir /user/godil/dir1`

List the content of a directory

- `hadoop fs -ls /user/gpapas`

Upload and download a file in HDFS

- `hadoop fs -put /home/godil/file.txt    /user/godil/datadir/`
- `hadoop fs -get /user/godil/datadir/file.txt   /home/`

Look at the content of a file

- `Hadoop fs -cat /user/godil/datadir/book.txt`

Many more commands, similar to Unix

HBase

- NoSQL data store build on top of HDFS
- Based on the Google BigTable paper (2006)
- Can handle various types of data
- Stores large amount of data (TB,PB)
- Column-Oriented data store
- Big Data with random read and writes
- Horizontally scalable

MapReduce: Simple Programming for Big Data

- MapReduce is simple programming paradigm for the Hadoop ecosystem
- Traditional parallel programming requires expertise of different computing/systems concepts
  - examples: multithreads, synchronization mechanisms (locks, semaphores, and monitors)
  - incorrect use: can crash your program, get incorrect results, or severely impact performance
  - Usually not fault tolerant to hardware failure
- The MapReduce programming model greatly simplifies running code in parallel
  - you don't have to deal with any of above issues
  - only need to create, map and reduce functions

Based on Google's MR paper (2004)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then

Map Reduce Paradigm

- Map and Reduce are based on functional programming

<table>
<tr>
<td>

Map:
Apply a function to all the elements of List

```
list1=[1,2,3,4,5];
square x = x * x
list2=Map square(list1)
print list2

-> [1,4,9,16,25]
```

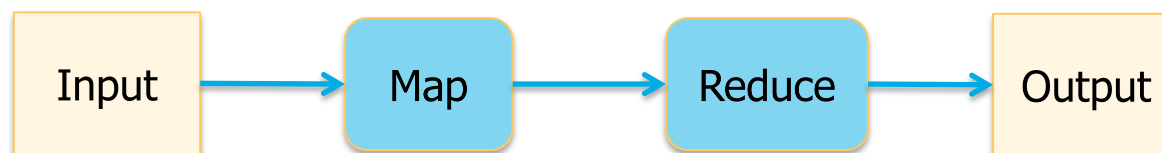
</td>
<td>

Reduce:
Combine all the elements of list for a summary

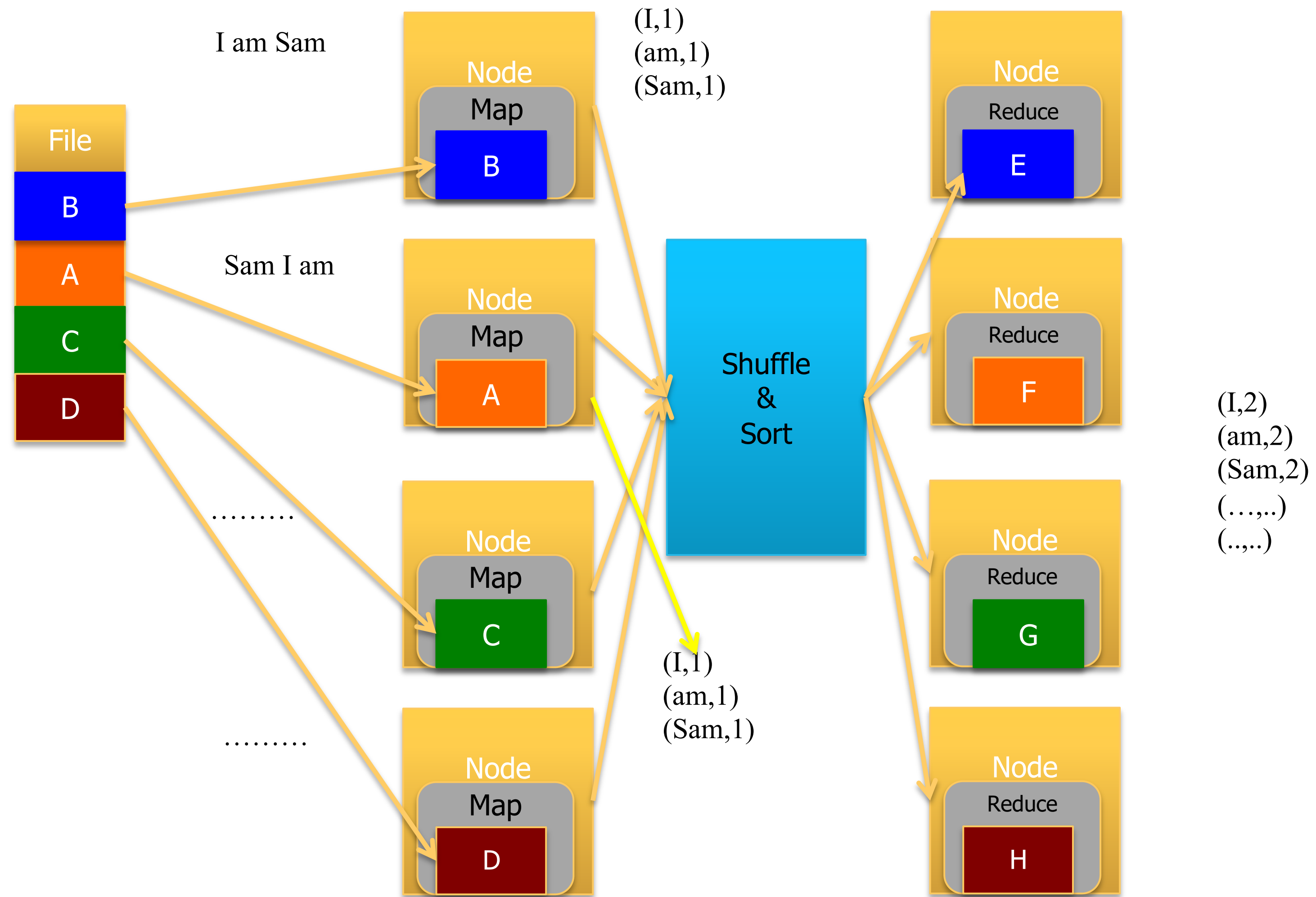
```
list2 = [1,4,9,16,25];
A = reduce (+) list1
Print A

-> 55
```

</td>
</tr>
</table>



MapReduce Word Count Example



Shortcoming of MapReduce

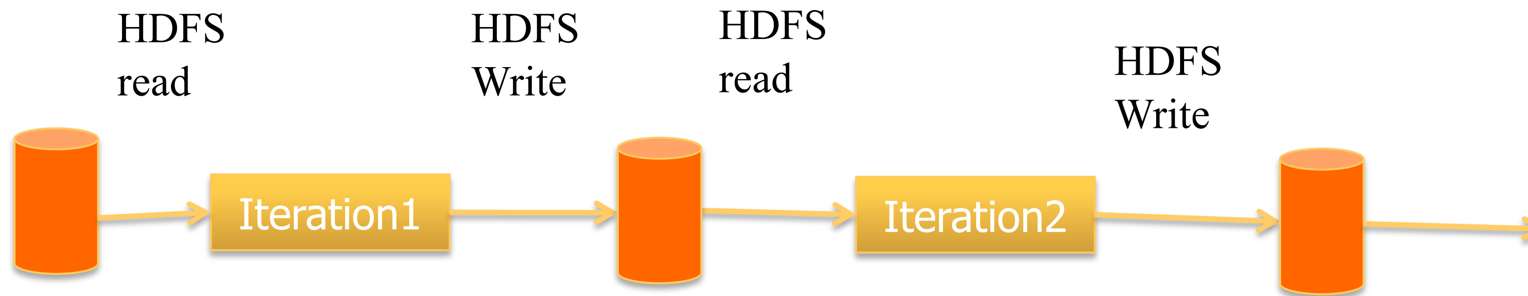
- Forces your data processing into Map and Reduce
  - Other workflows missing include join, filter, flatMap, groupByKey, union, intersection, …
- Based on “Acyclic Data Flow” from Disk to Disk (HDFS)
- Read and write to Disk before and after Map and Reduce (stateless machine)
  - Not efficient for iterative tasks, i.e. Machine Learning
- Only Java natively supported
  - Support for others languages needed
- Only for Batch processing
  - Interactivity, streaming data

One Solution is Apache Spark

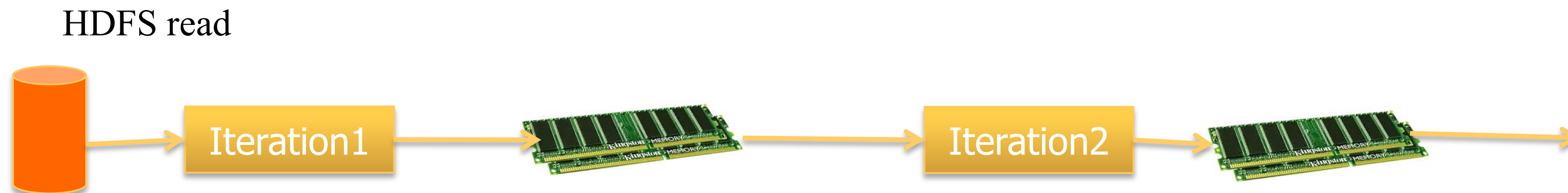
- A new general framework, which solves many of the short comings of MapReduce
- It is capable of leveraging the Hadoop ecosystem, e.g. HDFS, YARN, HBase, S3, …
- Has many other workflows, i.e. join, filter, flatMapdistinct, groupByKey, reduceByKey, sortByKey, collect, count, first…
  - (around 30 efficient distributed operations)
- In-memory caching of data (for iterative, graph, and machine learning algorithms, etc.)
- Native Scala, Java, Python, and R support
- Supports interactive shells for exploratory data analysis
- Spark API is extremely simple to use
- Developed at AMPLab UC Berkeley, now by Databricks.com

Spark Uses Memory instead of Disk

Hadoop: Use Disk for Data Sharing



Spark: In-Memory Data Sharing



Sort competition

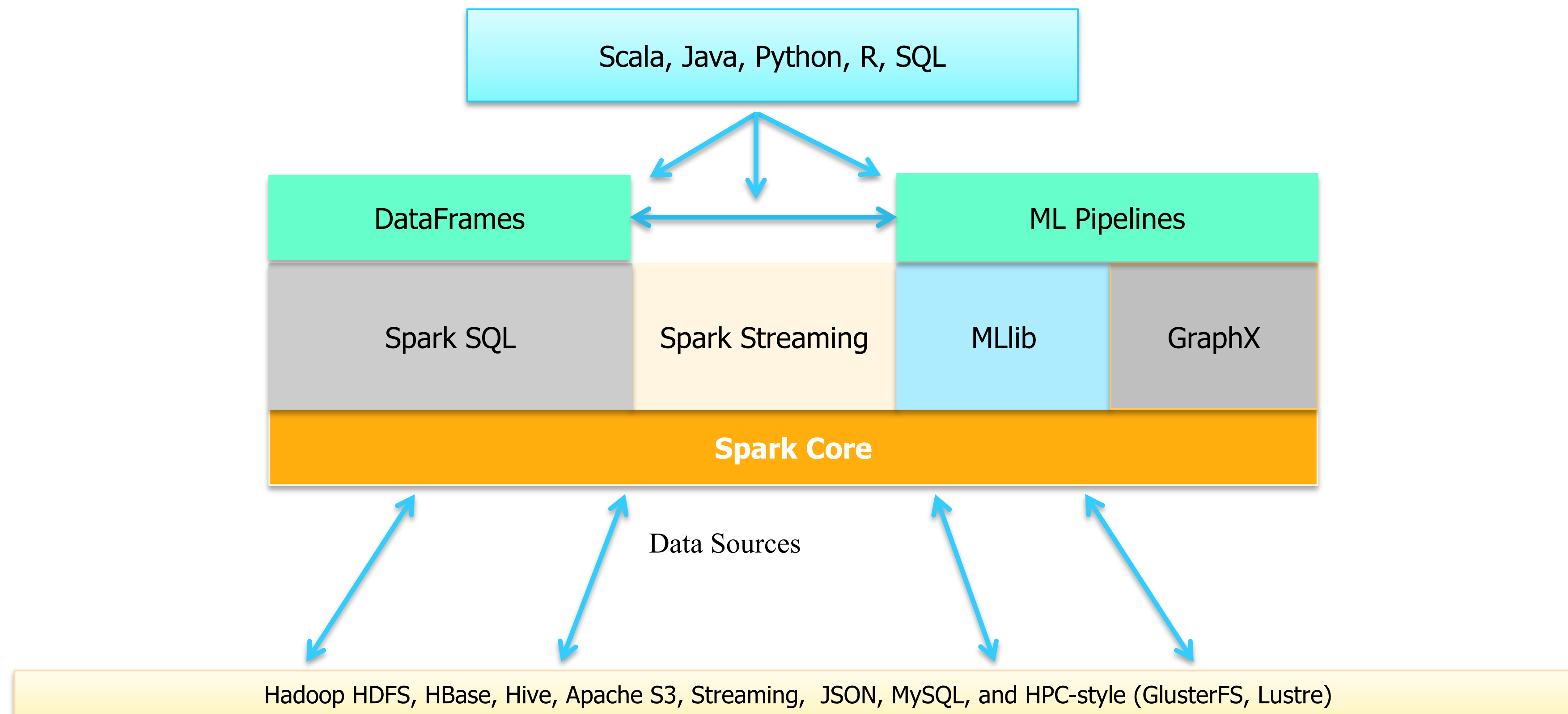
	Hadoop MR Record (2013)	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

Spark, 3x faster with 1/10 the nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)
http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Apache Spark

Apache Spark supports data analysis, machine learning, graphs, streaming data, etc. It can read/write from a range of data types and allows development in multiple languages.



Resilient Distributed Datasets (RDDs)

- RDDs (Resilient Distributed Datasets) is Data Containers
- All the different processing components in Spark share the same abstraction called RDD
- As applications share the RDD abstraction, you can mix different kind of transformations to create new RDDs
- Created by parallelizing a collection or reading a file
- Fault tolerant

DataFrames & SparkSQL

- DataFrames (DFs) is one of the other distributed datasets organized in named columns
- Similar to a relational database, Python Pandas Dataframe or R's DataTables
  - Immutable once constructed
  - Track lineage
  - Enable distributed computations
- How to construct Dataframes
  - Read from file(s)
  - Transforming an existing DFs(Spark or Pandas)
  - Parallelizing a python collection list
  - Apply transformations and actions

DataFrame example

```
// Create a new DataFrame that contains "students"
students = users.filter(users.age < 21)

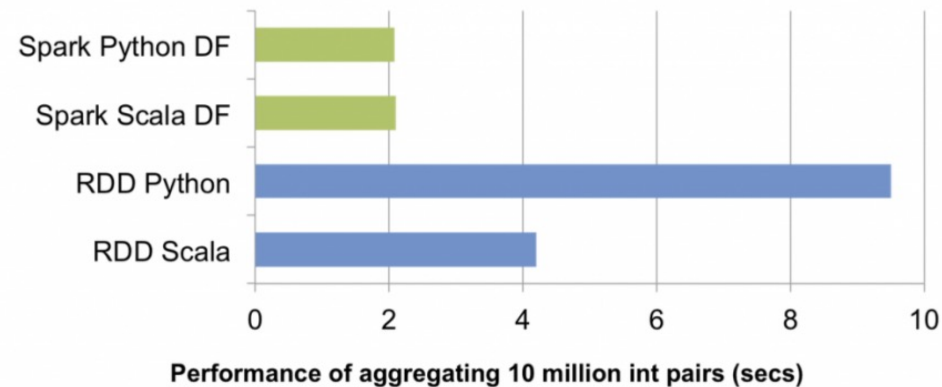
//Alternatively, using Pandas-like syntax
students = users[users.age < 21]

//Count the number of students users by gender
students.groupBy("gender").count()

// Join young students with another DataFrame called
logs
students.join(logs, logs.userId == users.userId,
"left_outer")
```

RDDs vs. DataFrames

- RDDs provide a low level interface into Spark
- DataFrames have a schema
- DataFrames are cached and optimized by Spark
- DataFrames are built on top of the RDDs and the core Spark API



More in the two upcoming LABS

Επιπλέον υλικό για μελέτη

- Main source of slides:
  - Silberschatz et al., “Database System Concepts”, 7th edition . Part 7: Parallel and Distributed Databases
  - DBMS Architectures - Data Science Lab @ University of Piraeus , Nikos Pelekis
  - MultiTenancy & SaaS by Felix Naumann https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/folien/WS0910/DBS_II/DBS2_12_WDM_MultiTenancy_FelixNaumann.pdf

- Google’s Map Reduce paper (2004): https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
- The Google File System paper (2003): https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

Thank you

GUNet2 eClass - Τμήμα Πληροφορικής

Χαρτοφυλάκιο χρήστη » CDS110- Big Data Management » Ταυτότητα Μαθήματος

Ενεργά εργαλεία

- Ανακοινώσεις
- Ασκήσεις
- Ατζέντα
- Έγγραφα
- Πληροφορίες Μαθήματος
- Σύνδεσμοι

Ανενεργά εργαλεία

- Ανταλλαγή Μηνυμάτων
- Γλωσσάριο
- Γραμμή μάθησης
- Εργασίες
- Ερωτηματολόγια
- Ηλεκτρονικό Βιβλίο
- Ομάδες Χρηστών
- Περιοχές Συζητήσεων

CDS110- Big Data Management

Περιγραφή

Introduction - review of relational and object-relational databases. Modern trends in database design. Non-traditional data types (text, multimedia, spatial information). Non-traditional database architecture (sensor networks, data streams, distributed, in the cloud). The “big data” era (MapReduce architecture, etc.). **Lab hours with** PostgreSQL, MongoDB, Spark (Batch Processing, Streaming, MLib).

Ταυτότητα Μαθήματος

- **Κωδικός:** CDS110
- **Εκπαιδευτές:** Γιάννης Θεοδωρίδης, Γιώργος Παπαστεφανάτος. Εργαστηριακοί Βοηθοί: Γ. Αλεξίου, Γ. Θεοδωρόπουλος, Σ. Μαρούλης
- **Σχολή - Τμήμα:** Μεταπτυχιακό "Κυβερνοασφάλεια και Επιστήμη Δεδομένων"
- **Τύπος:** Μεταπτυχιακό

CDS110- Big Data Management » Σύνδεσμοι

CDS110- Big Data Management

Σύνδεσμοι

Γενικοί σύνδεσμοι

- Ιστοσελίδα Data Science Lab. (DataStories)

Κατηγοριοποιημένοι σύνδεσμοι

Books

- Bailis P, et al. (eds.) (2015) Readings in Database Systems
- Codd EF (1990) The relational model for database management: version 2
- Liu L, Özsu MT (eds.) (2009) Encyclopedia of Database Systems

Papers

- Abadi D, et al. (2013) The Beckman report on database research
- Abadi D, et al. (2018) The Seattle report on database research
- Abiteboul S, et al. (2003) The Lowell database research self assessment
- Agrawal R, et al. (2008) The Claremont report on database research
- Codd EF (1970) A relational model of data for large shared data banks

Posts

- Big Data Architecture: A Complete and Detailed Overview
- HPI Genealogy of Relational Database Management Systems

Videos, Tutorials etc.

- Learn PostgreSQL Tutorial - Full Course for Beginners
- History of Databases

Search

All teams

CDS110: Big Data Managem...

Class Notebook
Assignments
Grades
Insights

Channels

General
MongoDB channel
PostgreSQL channel
Spark channel

General Posts Files

Team 3 Guests Meet

Welcome to CDS110: Big Data Management

Choose where you want to start

Upload Class Materials

Set up Class Notebook

IOANNIS THEODORIDIS 1/10 7:44 AM
Scheduled a meeting

CDS110: Big Data Management online lectures
Occurs every Monday @6:00 PM until 31/1/22

Reply