2. Distributed DBMS Architecture

Chapter 4

Distributed DBMS Architecture

Outline

- Introduction
- Top-Down Design of DDBMS Architecture
  - Schema and Distribution Transparency
- Bottom-up Design of DDBMS Architecture
  - Architectural Alternatives for DDBMSs
  - Reference Architectures for a DDBMS
- Global Directory/Dictionary

Outline

- ☞ Introduction
- ❖ Top-Down Design of DDBMS Architecture
  - ◆ Schema and Distribution Transparency
- ❖ Bottom-up Design of DDBMS Architecture
  - ◆ Architectural Alternatives for DDBMSs
  - ◆ Reference Architectures for a DDBMS
- ❖ Global Directory/Dictionary

Introduction

- Architecture defines the structure of the system
  - components identified
  - functions of each component defined
  - interrelationships and interactions between components defined

Reference Model（参考模型）

A conceptual framework whose purpose is to divide standardization work into manageable pieces and to show at a general level how these pieces are related to one another.

Three Approaches to Define a Reference Model

① Component-based

- Components of the system are defined together with the interrelationships between components.
- Good for design and implementation of the system.

② Function-based

- Classes of users are identified together with the functionality that the system will provide for each class.
- The objectives of the system are clearly identified. But how do you achieve these objectives?

③ Data-based

- Identify different types of data and specify the functional units that will realize and/or use data according to these views.
- The ANSI/SPARC architecture discussed below belongs to this category.

ANSI/SPARC Data-based Architecture

Users

External Schema

External View | External View | External View

Conceptual Schema

Conceptual View

Internal Schema

Internal View

Conceptual Schema （概念模式）

RELATION EMP [
KEY = {ENO}
ATTRIBUTES = {
ENO : CHARACER(9)
ENAME : CHARACER(15)
TITLE : CHARACER(10)
}
]

RELATION PAY [
KEY = {TITLE}
ATTRIBUTES = {
TITLE : CHARACER(10)
SAL : NUMERIC(6)
}
]

RELATION PROJECT [
KEY = {PNO}
ATTRIBUTES = {
PNO : CHARACER(7)
PNAME : CHARACER(20)
BUDGET : NEMERIC(7)
}
]

RELATION ASG [
KEY = {ENO,PNO}
ATTRIBUTES = {
ENO : CHARACER(9)
PNO : CHARACER(7)
RESP : CHARACER(10)
DUR : NUMERIC(6)
}
]

Internal Schema （内部模式）

```
RELATION EMP [
    KEY = {ENO}
    ATTRIBUTES = {
        ENO      : CHARACER(9)
        ENAME    : CHARACER(15)
        TITLE    : CHARACER(10)
    }
]
```

```
INTERNAL_REL EMP [
    INDEX ON ENO CALL EMINX
    FIEDLS = {
        HEADER :  BYTE(1)
        ENO    :  BYTE(9)
        ENAME  :  BYTE(15)
        TITLE  :  BYTE(10)
    }
]
```

External View（外部模式 ）– Example 1

Create a BUDGET view from the PROJ relation

CREAT VIEW BUDGET(PNAME, BUD)
AS **SELECT** PNAME, BUDGET
FROM PROJ

RELATION PROJECT [
KEY = {PNO}
ATTRIBUTES = {
PNO : CHARACER(7)
PNAME : CHARACER(20)
BUDGET : NEMERIC(7)
}
]

External View（外部模式 ）– Example 2

Create a Payroll view from relations EMP and PAY

CREAT VIEW		PAYROLL(ENO, ENAME, SAL)
AS	**SELECT**	EMP.ENO, EMP.ENAME, PAY.SAL
	FROM	EMP, PAY
	WHERE	EMP.TITLE = PAY.TITLE

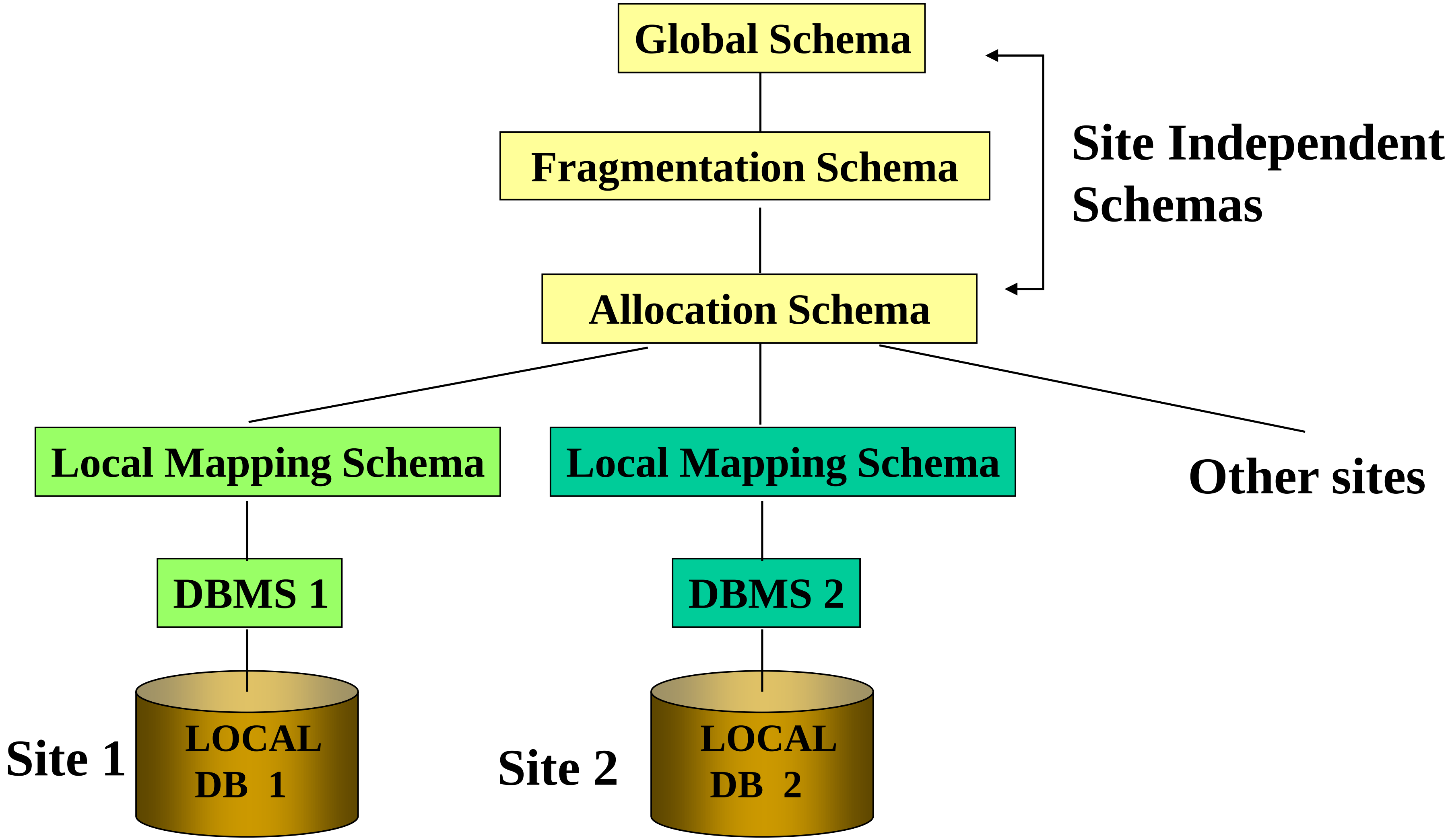
RELATION EMP [
  KEY = {ENO}
  ATTRIBUTES = {
    ENO : CHARACER(9)
    ENAME : CHARACER(15)
    TITLE : CHARACER(10)
  }
]

RELATION PAY [
  KEY = {TITLE}
  ATTRIBUTES = {
    TITLE : CHARACER(10)
    SAL : NUMERIC(6)
  }
]

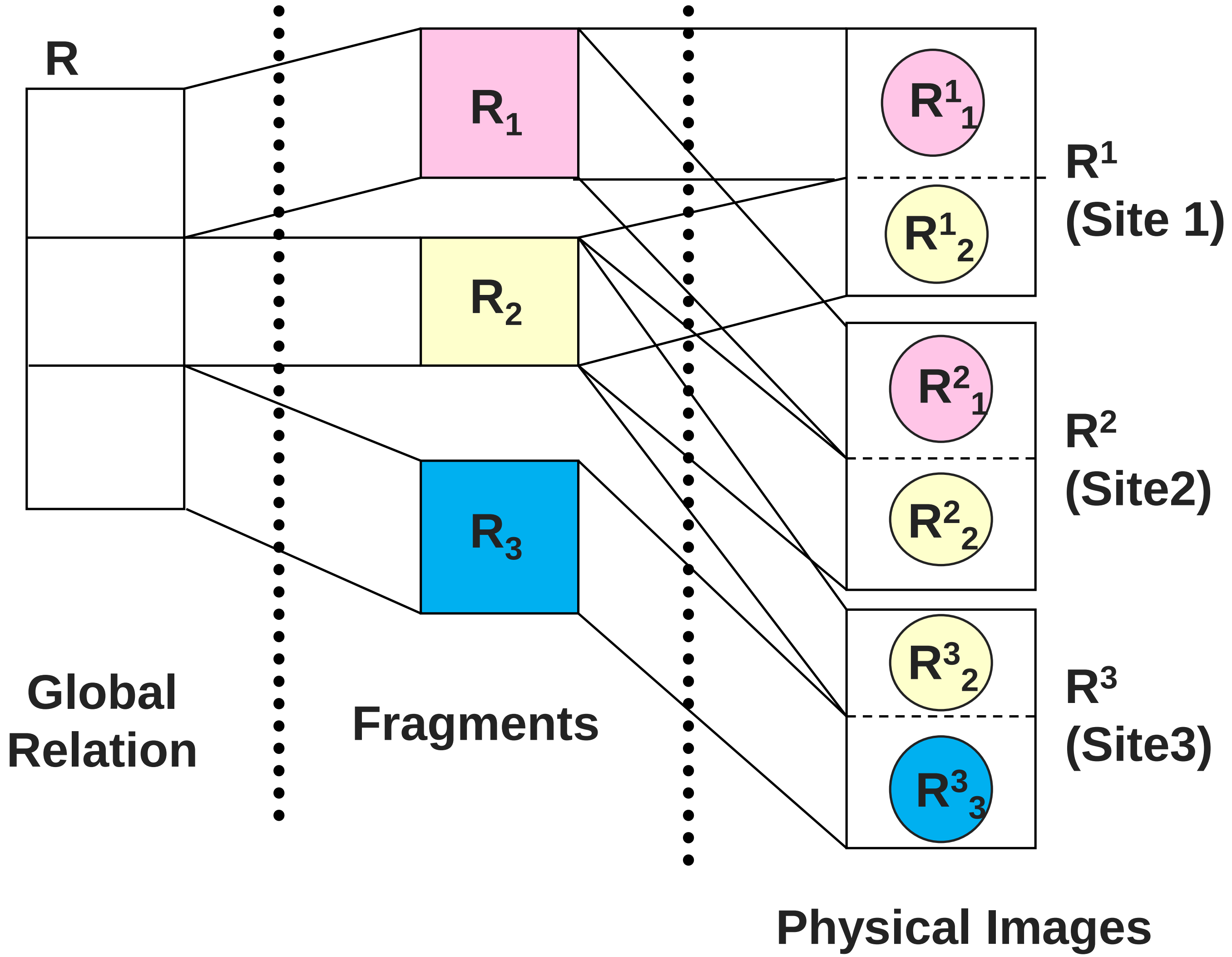
Outline

- Introduction
- Top-Down Design of DDBMS Architecture
  - Schema and Distribution Transparency
- Bottom-up Design of DDBMS Architecture
  - Architectural Alternatives for DDBMSs
  - Reference Architectures for a DDBMS
- Global Directory/Dictionary

Top-Down Classical DDBMS Architecture



Global Relations, Fragments, and Physical Images



DDBMS Schemas

- **Global Schema**: a set of global relations as if database were not distributed at all
- **Fragmentation Schema**: global relation is split into "non-overlapping" (logical) fragments. 1:n mapping from relation R to fragments R_i.
- **Allocation Schema**: 1:1 or 1:n (redundant) mapping from fragments to sites. All fragments corresponding to the same relation R at a site j constitute the physical image R^j. A copy of a fragment is denoted by R^j_i.
- **Local Mapping Schema**: a mapping from physical images to physical objects, which are manipulated by local DBMSs.

Motivation for this Architecture

- Separating the concept of data fragmentation from the concept of data allocation
- Fragmentation transparency
- Location transparency
- Explicit control of redundancy
- Independence from local databases allows local mapping transparency

Rules for Data Fragmentation

- Completeness

  All the data of the global relation must be mapped into the fragments.

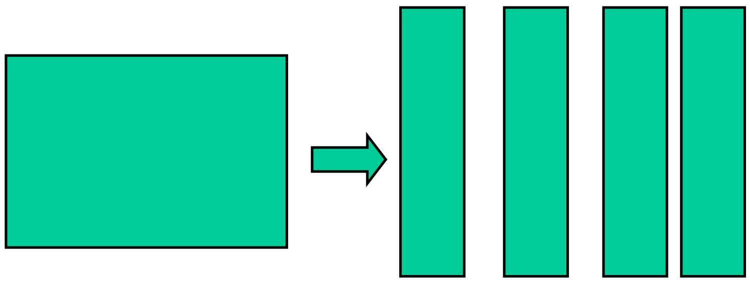
- Reconstruction

  It must always be possible to reconstruct each global relation from its fragments.

- Disjointedness

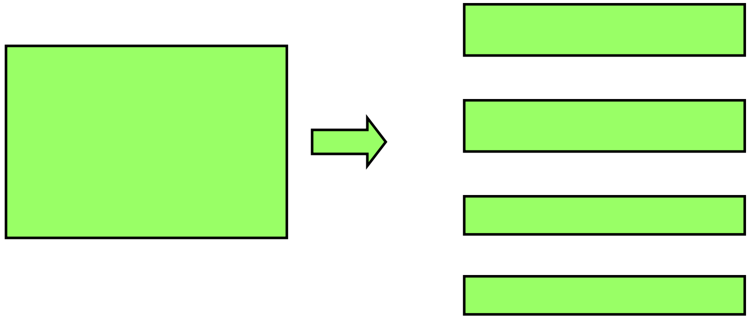
  It is convenient that fragments are disjoint, so that the replication of data can be controlled explicitly at the allocation level.

Types of Data Fragmentation



Vertical Fragmentation

- Projection on relation (subset of attributes)
- Reconstruction by join
- Updates require no tuple migration



Horizontal Fragmentation

- Selection on relation (subset of tuples)
- Reconstruction by union
- Updates may require tuple migration

Mixed Fragmentation

- A fragment is a Select-Project query on relation

Horizontal Fragmentation (水平划分)

- Partitioning the tuples of a global relation into subsets

Example:

Supplier (SNum, Name, City)

Horizontal Fragmentation can be:

$Supplier_1 = \sigma_{City = ``HK"} Supplier$

$Supplier_2 = \sigma_{City\ != \text{“HK”}} Supplier$

Reconstruction is possible:

$Supplier = Supplier_1 \cup Supplier_2$

- The set of predicates defining all the fragments must be complete, and mutually exclusive

Derived Horizontal Fragmentation

$Supplier_1 = \sigma_{City = ``HK''}\ Supplier$

$Supplier_2 = \sigma_{City\ != “HK”}\ Supplier$

- The horizontal fragmentation is derived from the horizontal fragmentation of another relation

Example:

Supply (SNum, PNum, DeptNum, Quan)

SNum is a supplier number

$\ltimes$ **semijoin operation**

$\mathbf{Supply_1 = Supply \ltimes_{SNum=SNum} Supplier_1}$

$\mathbf{Supply_2 = Supply \ltimes_{SNum=SNum} Supplier_2}$

The predicates defining derived horizontal fragments are:

(Supply.SNum = Supplier.SNum) and (Supplier. City = ``HK")

(Supply.SNum = Supplier.SNum) and (Supplier. City != ``HK")

Vertical Fragmentation (垂直划分)

- The vertical fragmentation of a global relation is the subdivision of its attributes into groups; fragments are obtained by projecting the global relation over each group

Example

EMP (ENum,Name,Sal,Tax,MNum,DNum)

A vertical fragmentation can be

$$\mathbf{EMP_1} = \Pi_{\text{ENum, Name, MNum, DNum}}\ \mathbf{EMP}$$

$$\mathbf{EMP_2} = \Pi_{\text{ENum, Sal, Tax}}\ \mathbf{EMP}$$

Reconstruction:

$$\mathbf{EMP} = \mathbf{EMP_1} \bowtie_{\mathbf{ENum = ENum}} \mathbf{EMP_2}$$

Distribution Transparency (分布透明)

- Different levels of distribution transparency can be provided by DDBMS for applications.

A Simple Application

Supplier(SNum, Name, City)

Horizontally fragmented into:

$Supplier_1 = \sigma_{City = \text{``HK''}}$ Supplier at Site1

$Supplier_2 = \sigma_{City != \text{“HK”}}$ Supplier at Site2, Site3

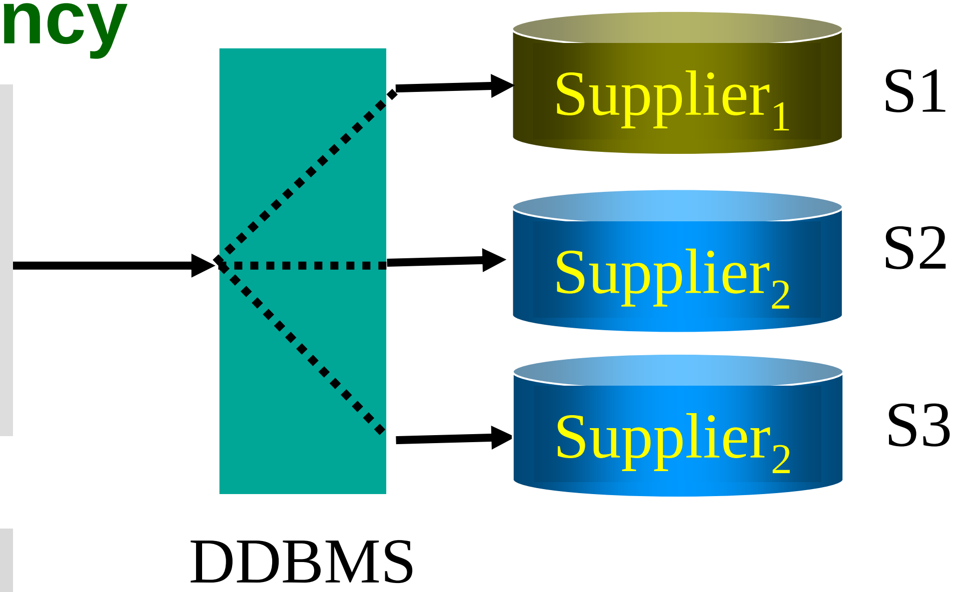
Application:

Read the supplier number from the user and return the name of the supplier with that number.

Level 1 of Distribution Transparency

Fragmentation Transparency

```
read (terminal, $SNum);
      Select     Name into $Name
      from       Supplier
      where      SNum = $SNum;
write (terminal, $Name).
```



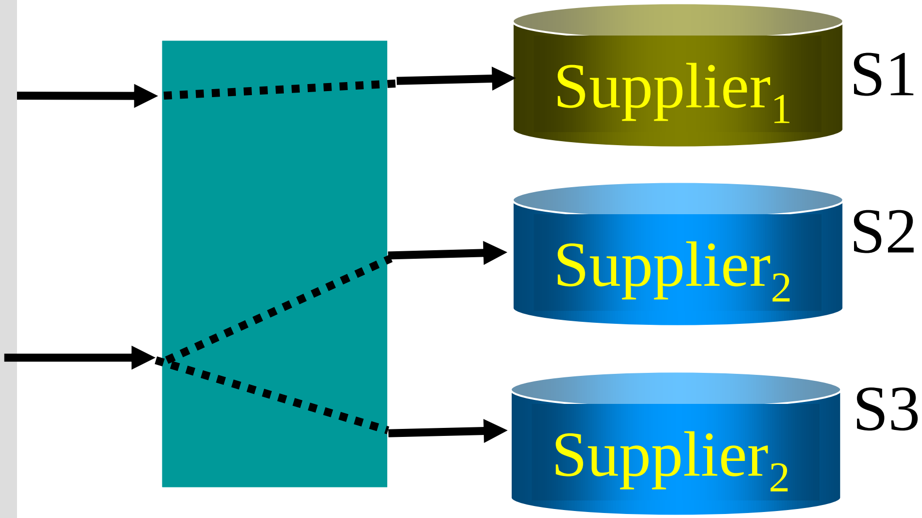
Read the supplier number from the user and return the name of the supplier with that number.

The DDBMS interprets the database operation by accessing the databases at different sites in a way which is completely determined by the system.

Level 2 of Distribution Transparency

Location Transparency (but fragmentation not)

read (terminal, $SNum);
    Select Name into $Name
    from Supplier_1
    where SNum = $SNum;
If not FOUND then
    Select Name into $Name
    from Supplier_2
    where SNum = $SNum;
write (terminal, $Name).

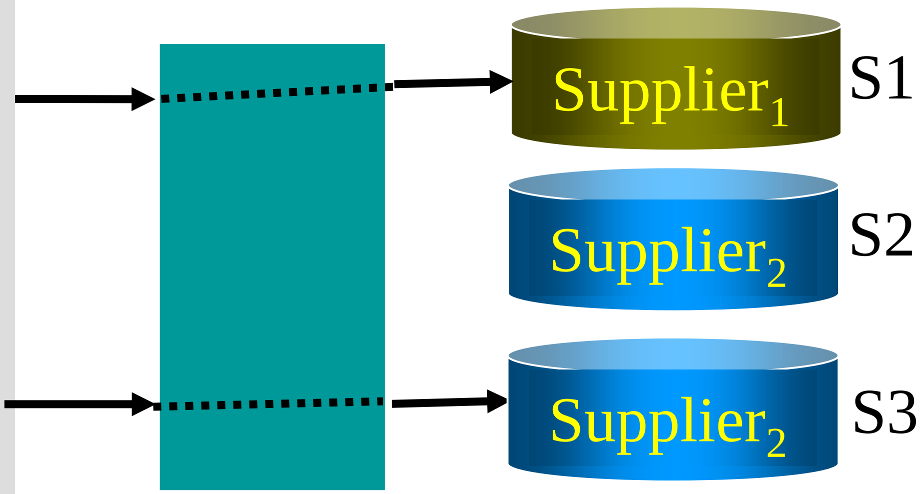


The application is independent from changes in allocation schema, but is not independent from changes to fragmentation schema.

Level 3 of Distribution Transparency

Local Mapping Transparency (but distribution not)

```
read (terminal, $SNum);
        Select      Name into $Name
        from        S1.Supplier1
        where       SNum = $SNum;
 If not FOUND then
        Select      Name into $Name
        from        S3.Supplier2
        where       SNum = $SNum;
write (terminal, $Name).
```

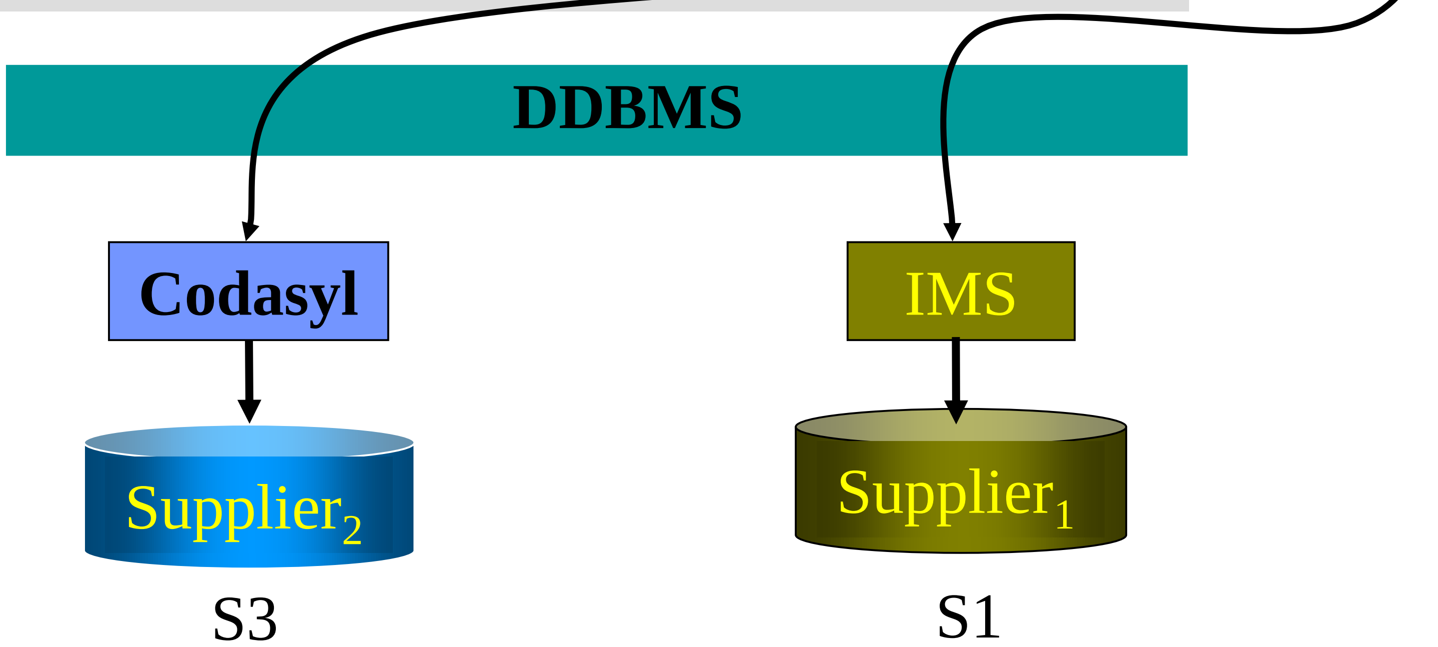


The applications have to specify both the fragment names and the sites where they are located. The mapping of database operations specified in applications to those in DBMSs at sites is transparent.

Level 4 of Distribution Transparency

- No transparency at all !!!

```
read (terminal, $SNum);
  $SupIMS($Snum,$Name,$Found) at S1;
 If not FOUND then
  $SupCODASYL($Snum,$Name,$Found) at S3;
write (terminal, $Name).
```



Distribution Transparency for Updates

Difficulties:

- broadcasting updates to all copies
- migration of tuples because of change of fragment defining attributes

$$EMP1 = \Pi_{Enum,Name,Sal,Tax}\sigma_{Dnum\leq 10}(EMP)$$
$$EMP2 = \Pi_{Enum,Mnum,Dnum}\sigma_{Dnum\leq 10}(EMP)$$
$$EMP3 = \Pi_{Enum,Name,Dnum}\sigma_{Dnum>10}(EMP)$$
$$EMP4 = \Pi_{Enum,Mnum,Sal,Tax}\sigma_{Dnum>10}(EMP)$$

EMP1

Enum	Name	Sal	Tax
100	Ann	100	10

EMP2

Enum	Mnum	Dnum
100	20	3

Update Dnum=15 for Employee with Enum=100

EMP3

Enum	Name	Dnum
100	Ann	15

EMP4

Enum	Mnum	Sal	Tax
100	20	100	10

An Update Application

```
UPDATE   EMP
SET      Dnum = 15
WHERE    Enum = 100;
```

With Level 1 Fragmentation Transparency

With Level 2 Location Transparency only

```
Select   Name, Tax, Sal into $Name, $Sal, $Tax
From     EMP 1
Where    Enum = 100;

Select   Mnum into $Mnum
From     EMP 2
Where    Enum = 100;

Insert into EMP 3 (Enum, Name, Dnum)
                  (100, $Name, 15);
Insert into EMP 4 (Enum, Sal, Tax, Mnum)
                  (100, $Sal, $Tax, $Mnum);

Delete EMP 1 where Enum = 100;
Delete EMP 2 where Enum = 100;
```

Levels of Distribution Transparency

- Fragmentation Transparency
  - Just like using global relations
- Location Transparency
  - Need to know fragmentation schema; but no need to know where fragments are located
  - Applications access fragments (no need to specify sites where fragments are located).
- Local Mapping Transparency
  - Need to know both fragmentation and allocation schema; no need to know what the underlying local DBMSs are.
  - Applications access fragments explicitly specifying where the fragments are located.
- No Transparency
  - Need to know local DBMS query languages, and write applications using functionality provided by the Local DBMS

On Distribution Transparency

- More distribution transparency requires appropriate DDBMS support, but makes end-application developers′ work easy.
- The less distribution transparency, the more the end-application developer needs to know about fragmentation and allocation schemes, and how to maintain database consistency.
- There are tough problems in query optimization and transaction management that need to be tackled (in terms of system support and implementation) before fragmentation transparency can be supported.

On Distribution Transparency (*cont.*)

- The level of transparency is inevitably a compromise between ease of use and the difficulty and overhead cost of providing high levels of transparency

- DDBMS provides fragmentation transparency and location transparency; OS provides network transparency; and DBMS provides data independence

Some Aspects of the Classical DDBMS Architecture

- Distributed database technology is an “add-on” technology, and most users already have populated centralized DBMSs, whereas top-down design assumes implementation of new DDBMS from scratch.
- In many application environments, such as semi-structured databases, continuous streaming multimedia data, the notion of fragment is difficult to define.

Outline

- Introduction
- Top-Down Design of DDBMS Architecture
  - Schema and Distribution Transparency
- ☞ Bottom-up Design of DDBMS Architecture
  - Architectural Alternatives for DDBMSs
  - Reference Architectures for a DDBMS
- Global Directory/Dictionary

Bottom-up Distributed Architectural Models

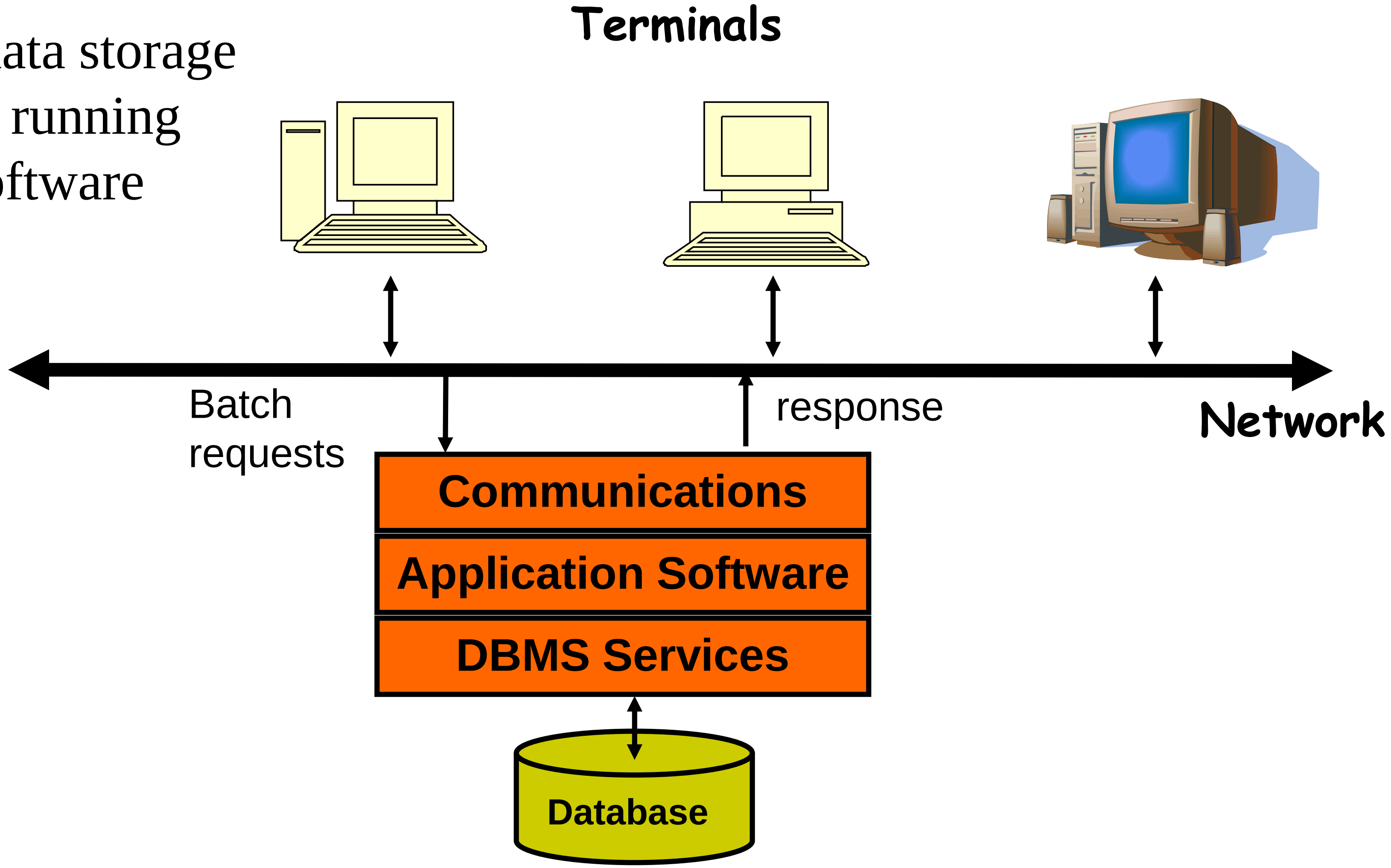
- Possible ways in which multiple data(bases) are put together for sharing, which are characterized according to three dimensions
  - Distribution
  - Heterogeneous
  - Autonomy

Dimension 1: Distribution (分布)

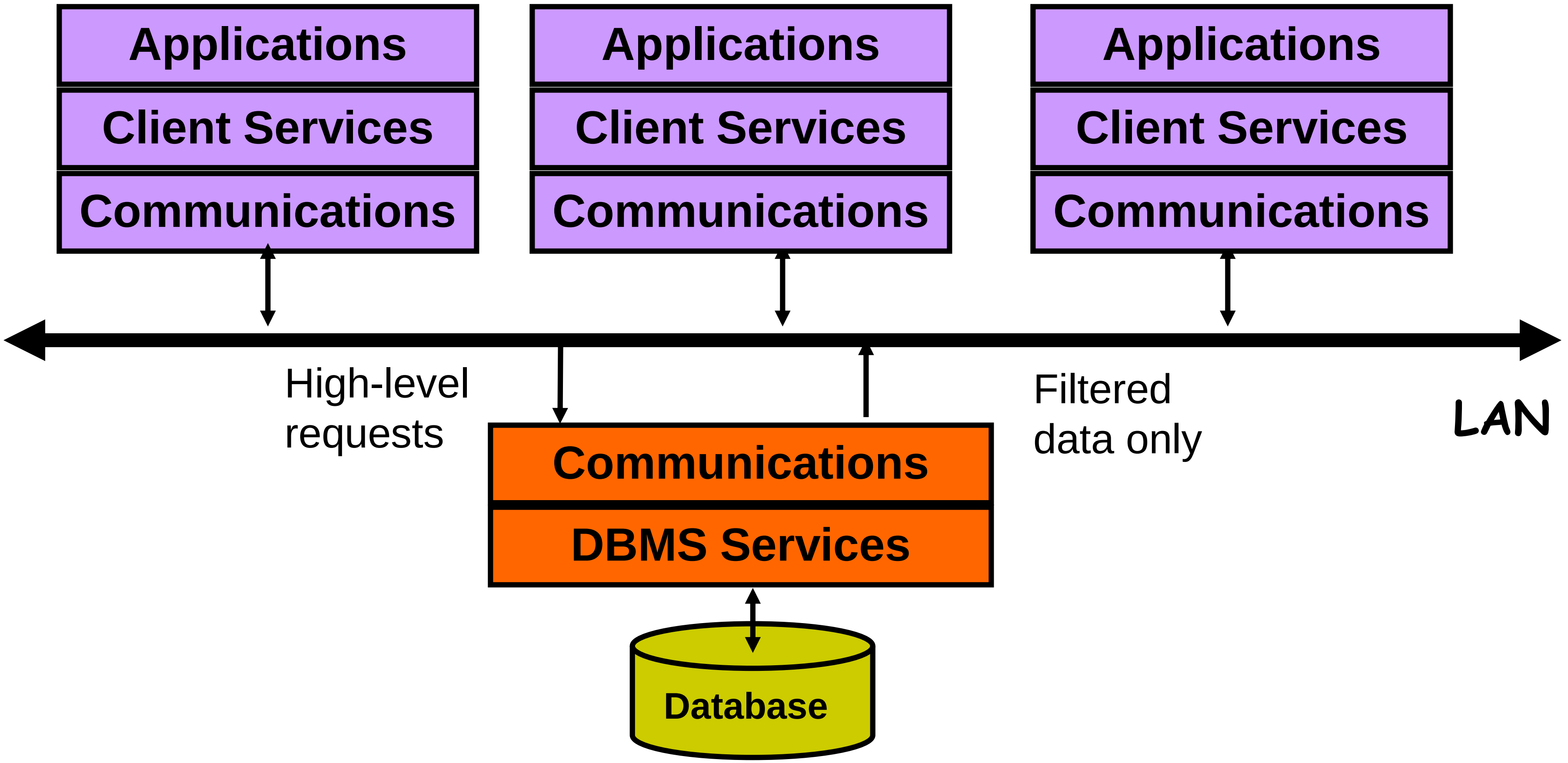
- Whether the components of the system are located on the same machine or not
  - 0 - no distribution - single site (central database)
  - 1 - client-server - distribution of DBMS functionalities
  - 2 - master-slaves - distribution of DBMS functionalities
  - 3 - peer to peer

0 – No Distribution (Time Sharing Access to a Central Database)

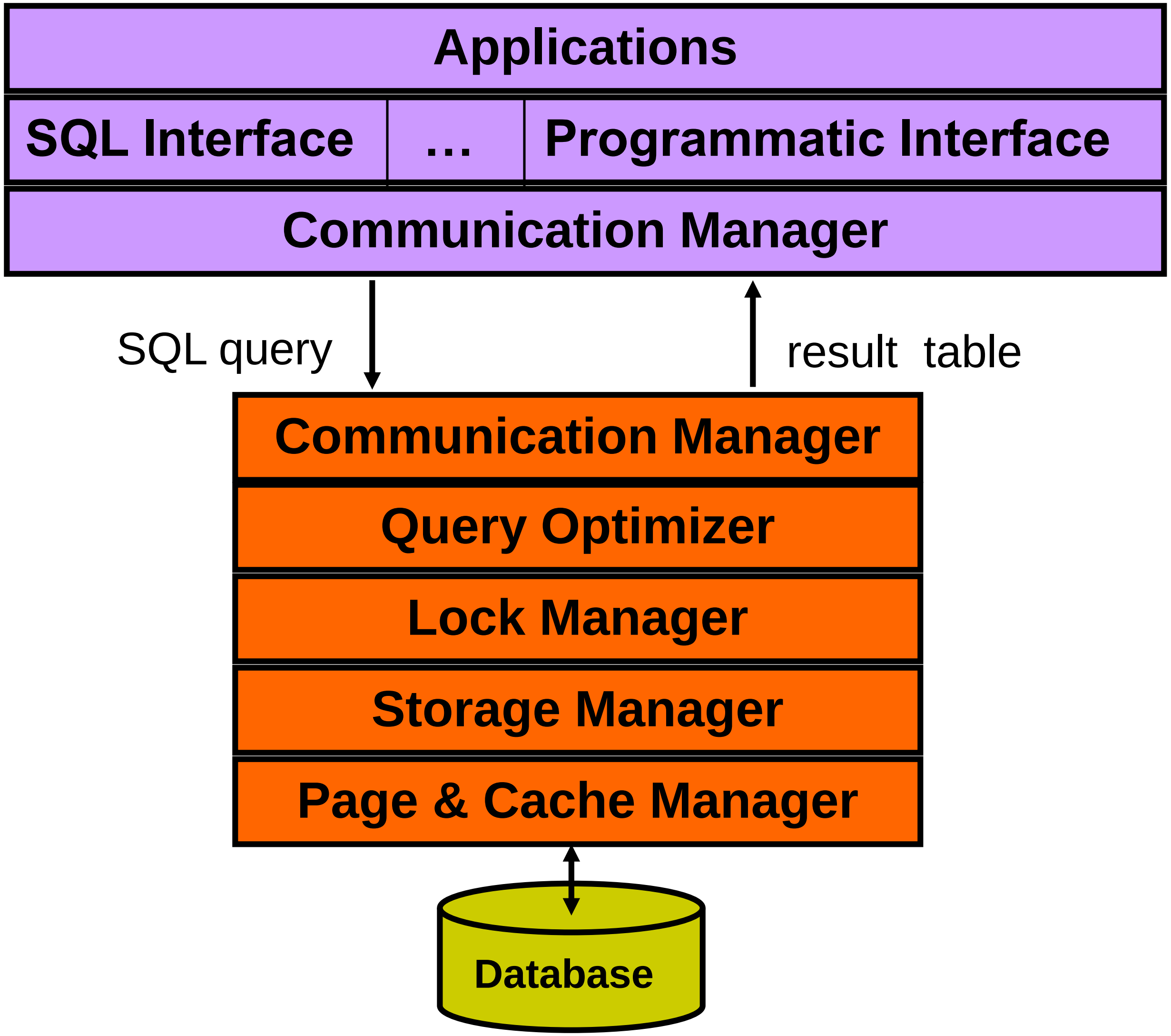
- No data storage
- Host running all software



0 – No Distribution (Multiple Clients / Single Server)



Task Distribution



Advantages of Client-Server Architectures

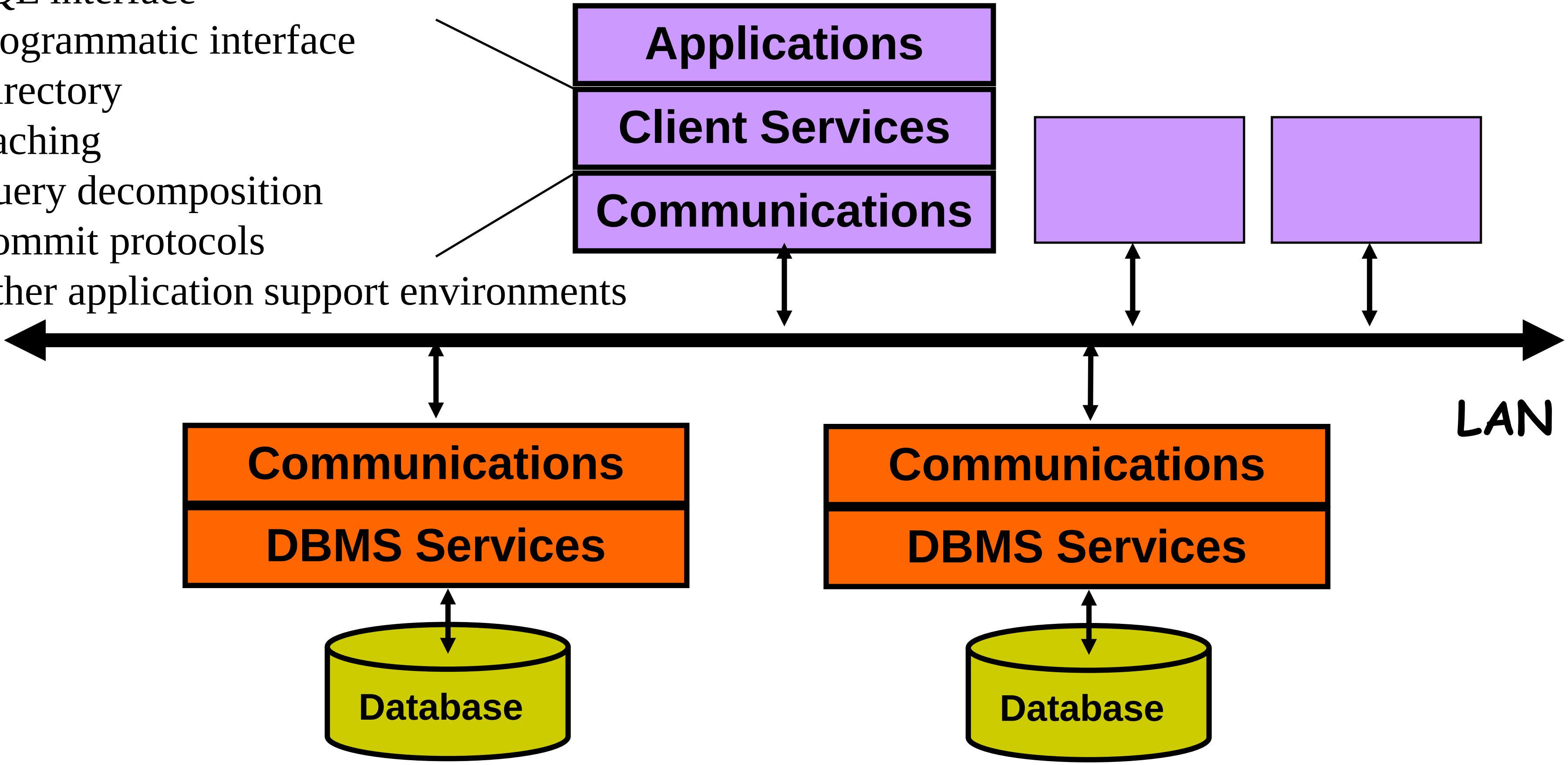
- More efficient division of labor
- Horizontal and vertical scaling of resources
- Better price/performance on client machines
- Ability to use familiar tools on client machines
- Client access to remote data (via standards)
- Full DBMS functionality provided to client workstations
- Overall better system price/performance

Problems with Multiple-Clients / Single Server Architectures

- Server forms bottleneck
- Server forms single point of failure
- Database scaling difficult

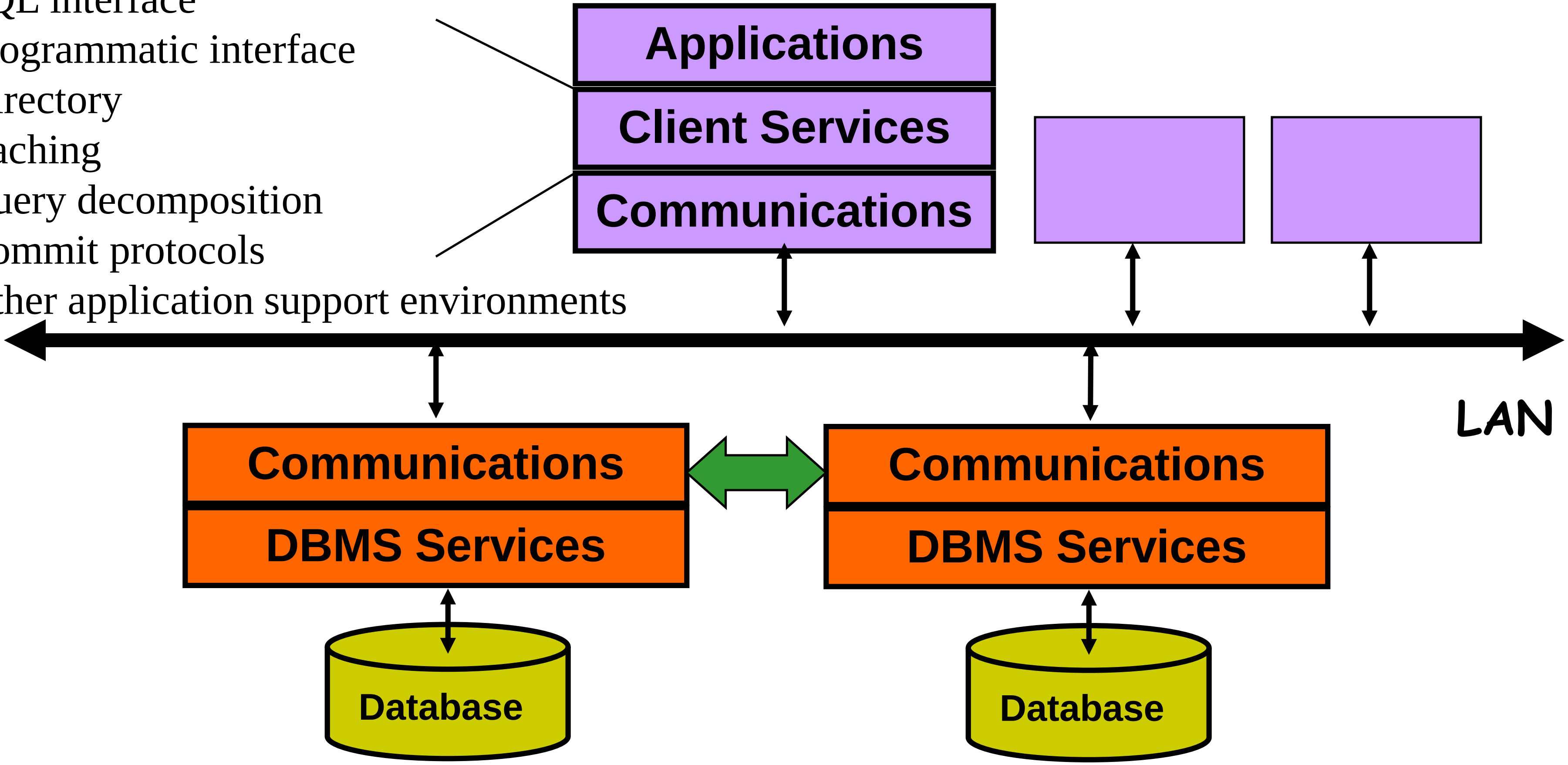
1 - Multiple Clients / Multiple Servers

- SQL interface
- Programmatic interface
- Directory
- Caching
- Query decomposition
- Commit protocols
- Other application support environments



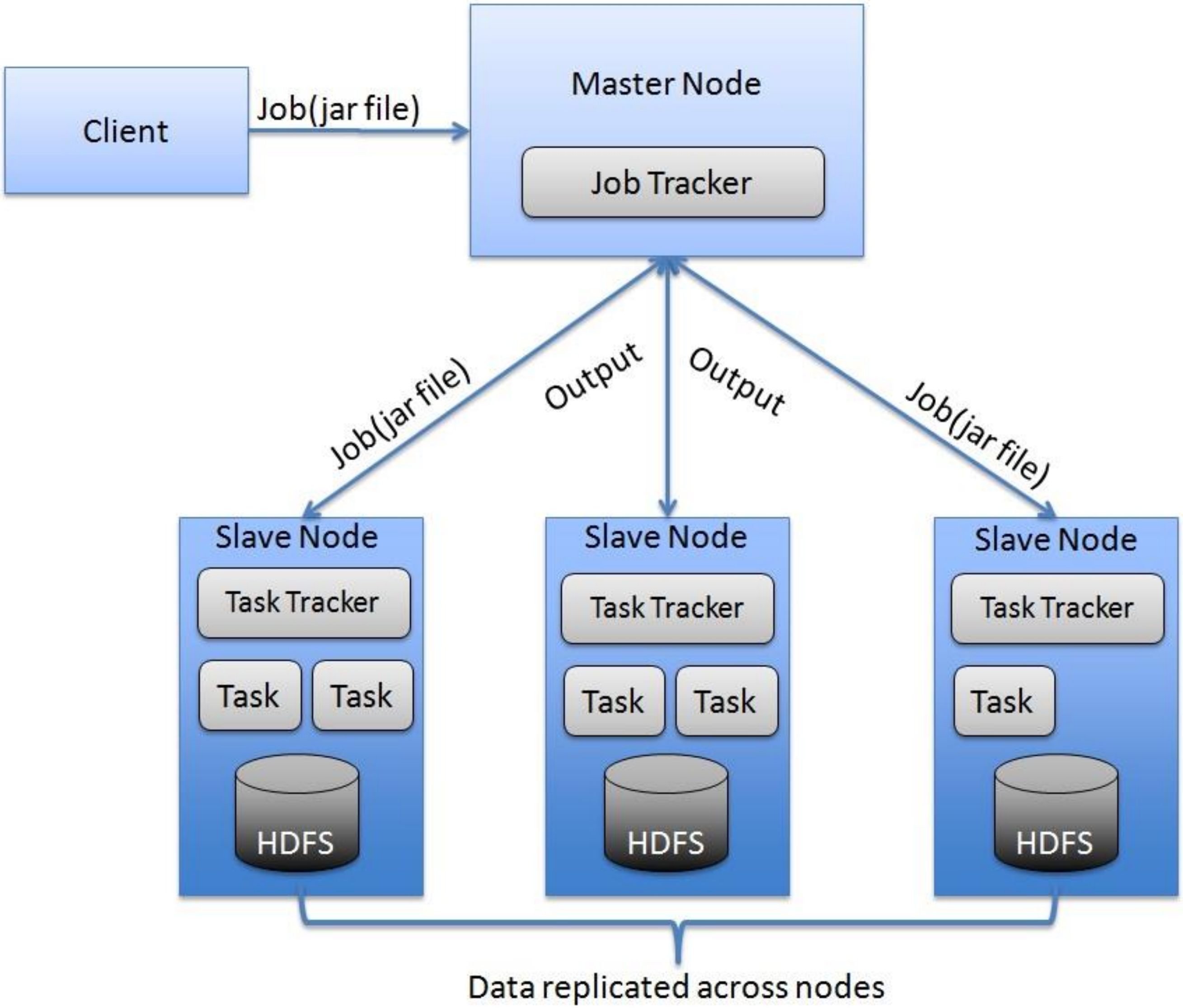
Server to Server

- SQL interface
- Programmatic interface
- Directory
- Caching
- Query decomposition
- Commit protocols
- Other application support environments



2 – Master-Slaves Architecture

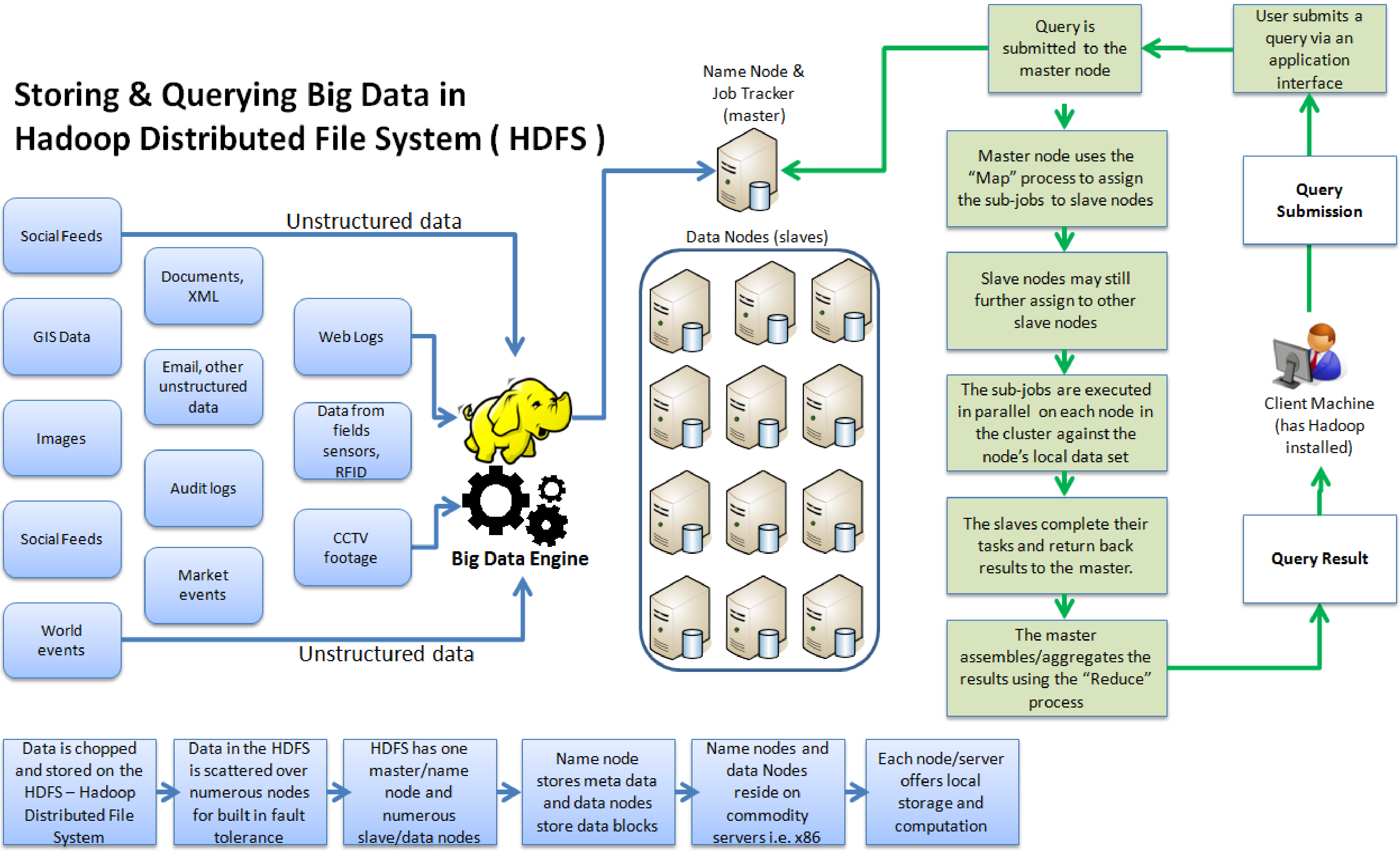
Apache Hadoop



What is Hadoop?

- Hadoop is a software framework for distributed processing of large datasets across a large cluster of computers
  - ***Large dataset*** → Terabytes or petabytes of data
  - ***Large cluster*** → hundreds or thousands of nodes
- Hadoop is open-source implementation for Google ***MapReduce***
- Hadoop is based on a simple programming model called ***MapReduce***
- Hadoop is based on a simple data model
  - any data will fit

Storing & Querying Big Data in Hadoop Distributed File System (HDFS)



Designed by Sri Prakash, November 2012

Design Principles of Hadoop

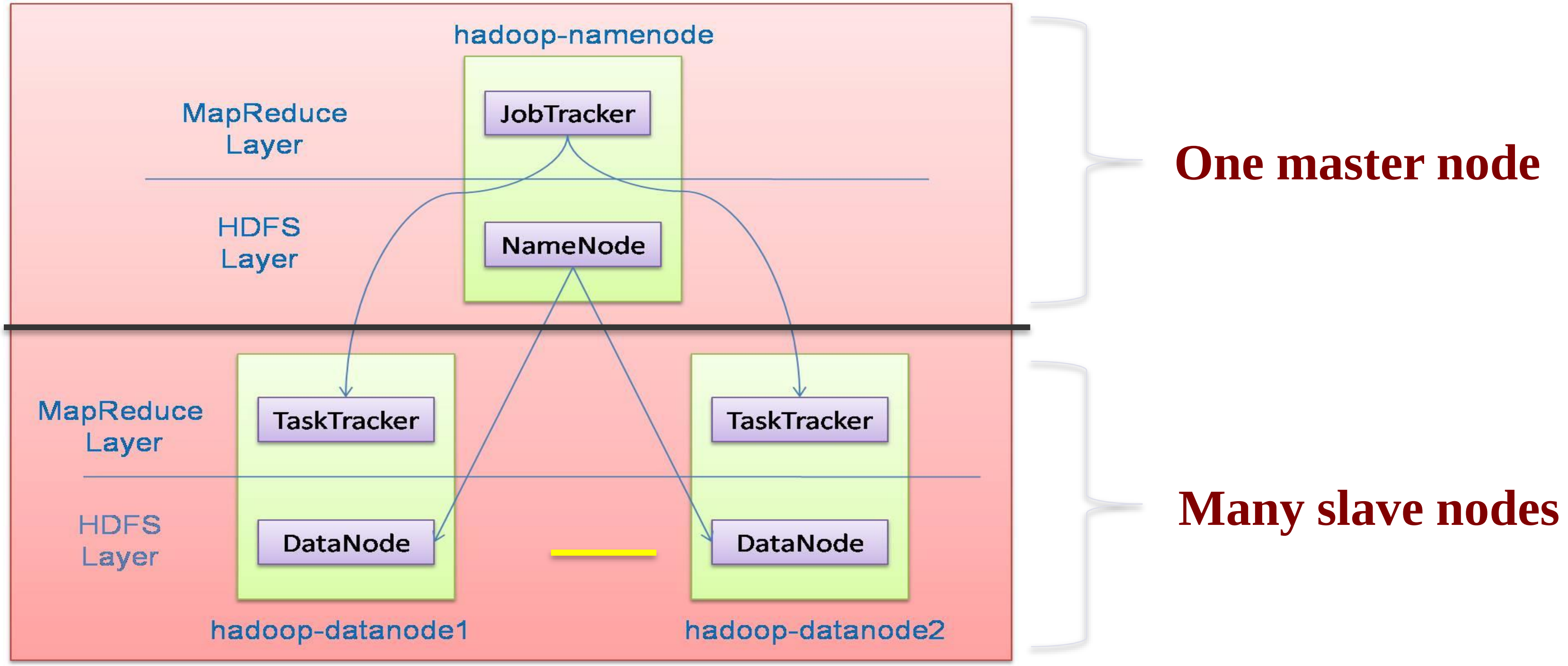
- Need to process big data
- Need to parallelize computation across thousands of nodes
- **Commodity hardware**
  - Large number of low-end cheap machines working in parallel to solve a computing problem
- This is in contrast to traditional parallel DBs
  - Small number of high-end expensive machines

Design Principles of Hadoop (*cont.*)

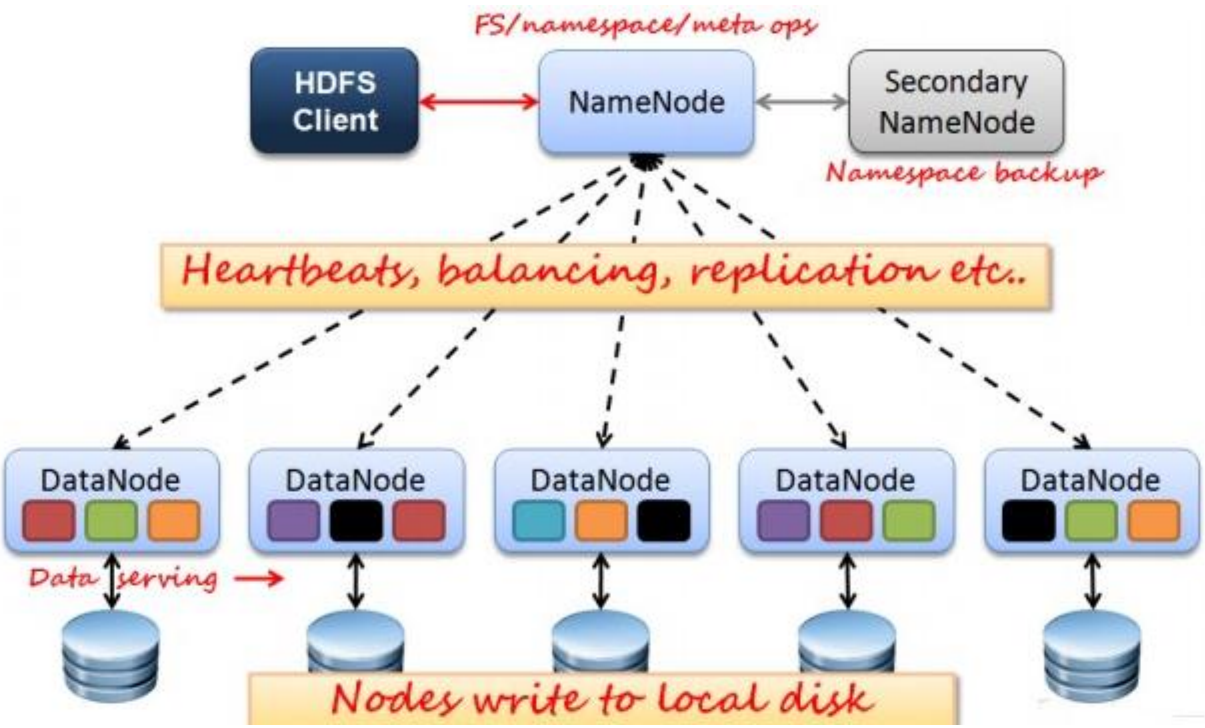
- **Automatic parallelization & distribution**
  - Hidden from the end-user
- **Fault tolerance and automatic recovery**
  - Nodes/tasks will fail and will recover automatically
- **Clean and simple programming abstraction**
  - Users only provide two functions “map” and “reduce”

Hadoop Architecture

- Master-slave & shared-nothing
- Two layers
  - Execution engine (MapReduce)
  - Distributed file system (HDFS)



Hadoop Distributed File System (HDFS)



A centralized namenode

- Maintains metadata info about files

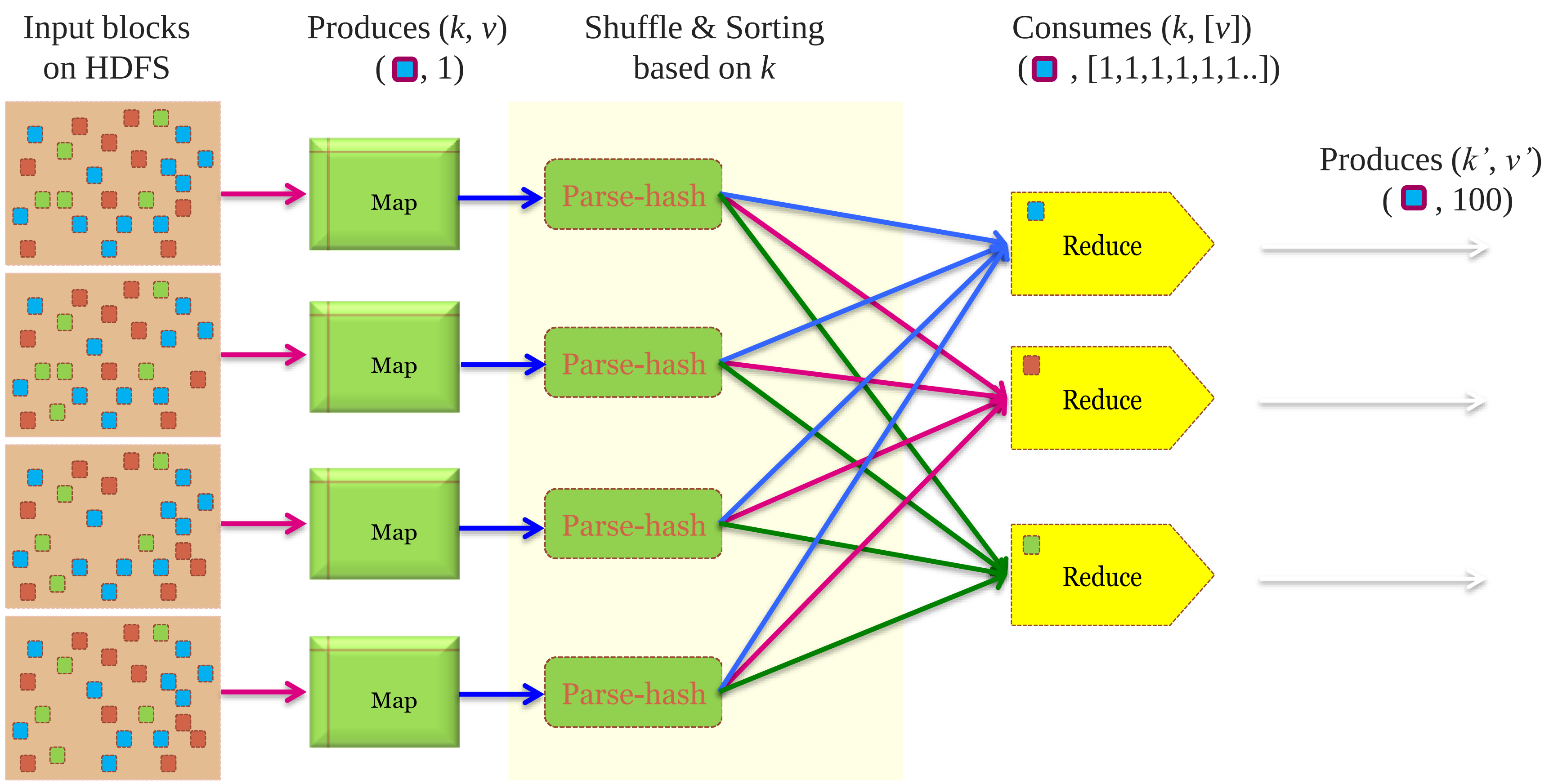
Many data nodes (1000s)

- Store the actual data
- Files are divided into blocks
- Each block is replicated *N* times (default = 3)

Main Properties of HDFS

- ***Large:*** A HDFS instance may consist of thousands of server machines, each storing part of the file system's data
- ***Replication:*** Each data block is replicated many times (default is 3)
- ***Failure:*** Failure is the norm rather than exception
- ***Fault Tolerance:*** Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS
  - Namenode consistently checks Datanodes

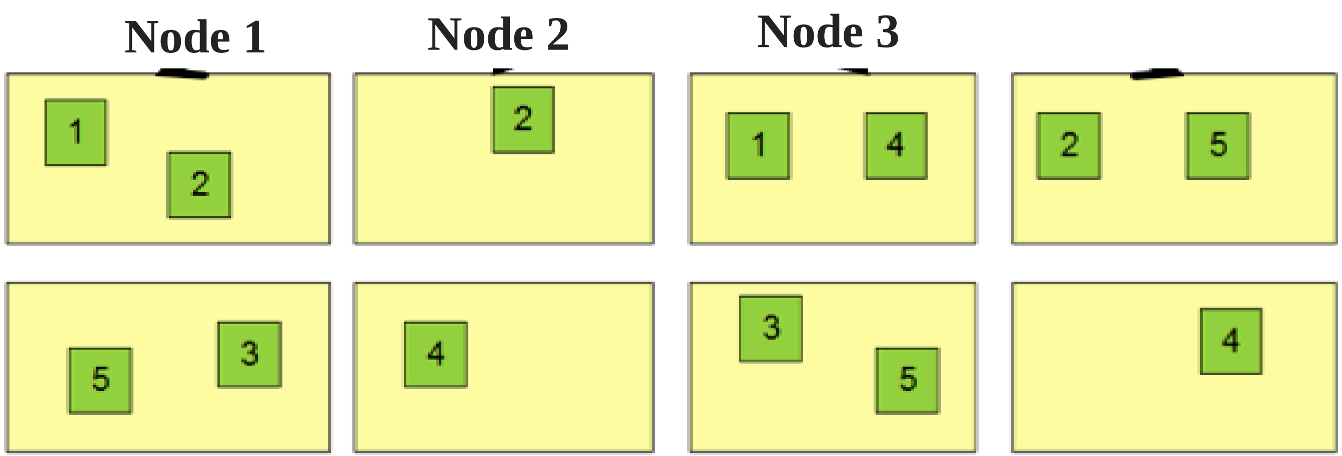
Map-Reduce Execution Engine (Example: Color Count)



Users only provide the "Map" and "Reduce" functions

Properties of MapReduce Engine

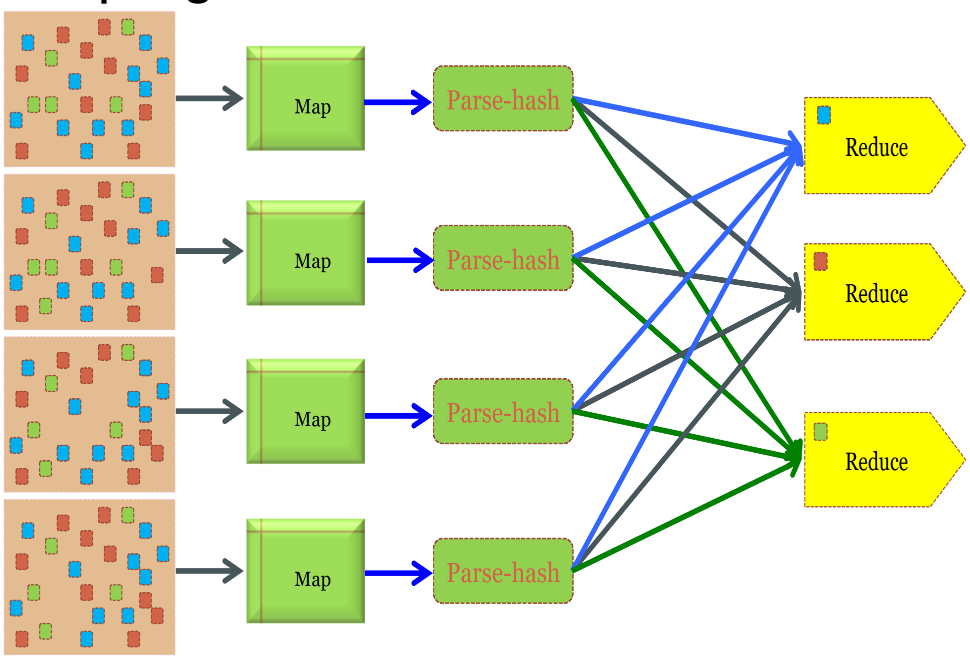
- Job Tracker is the master node (runs with the namenode)
  - Receives the user's job
  - Decides on how many tasks will run (number of mappers)
  - Decides on where to run each mapper (concept of locality)



- This file has 5 blocks → run 5 map tasks
- Where to run the task reading block "1"
  - *Try to run it on Node 1 or Node 3*

Properties of MapReduce Engine (*cont.*)

- Task Tracker is the slave node (runs on each data node)
  - Receives the task from Job Tracker
  - Runs the task until completion (either map or reduce task)
  - Always in communication with the Job Tracker reporting progress



In this example, 1 map-reduce job consists of 4 map tasks and 3 reduce tasks

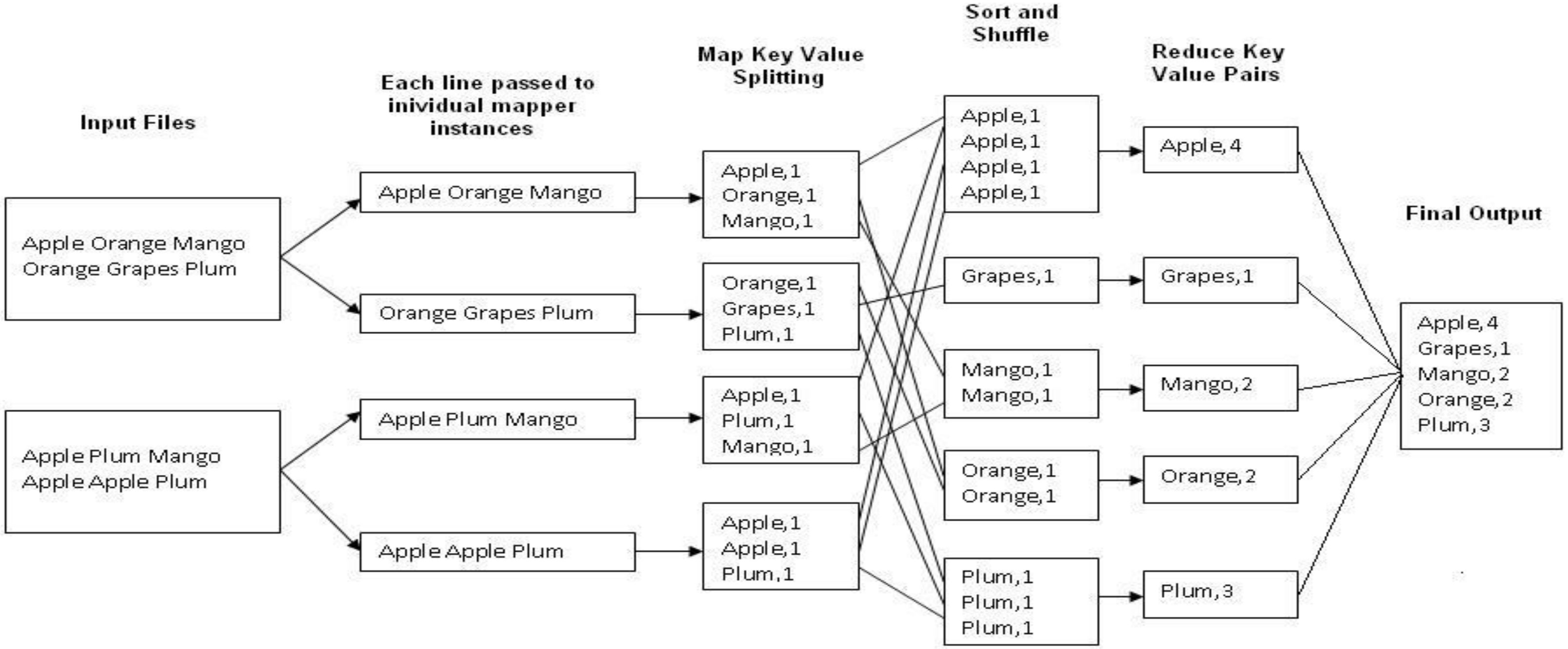
(Example: Color Count)

Key-Value Pairs

- Mappers and Reducers are users' code (provided functions)
- Just need to obey the Key-Value pairs interface
- **Mappers:**
  - Consume <key, value> pairs
  - Produce <key, value> pairs
- **Reducers:**
  - Consume <key, <list of values>>
  - Produce <key, value>
- **Shuffling and Sorting:**
  - Hidden phase between mappers and reducers
  - Groups all similar keys from all mappers, sorts and passes them to a certain reducer in the form of <key, <list of values>>

Example 1: Word Count

Job: Count the occurrences of each word in a data set

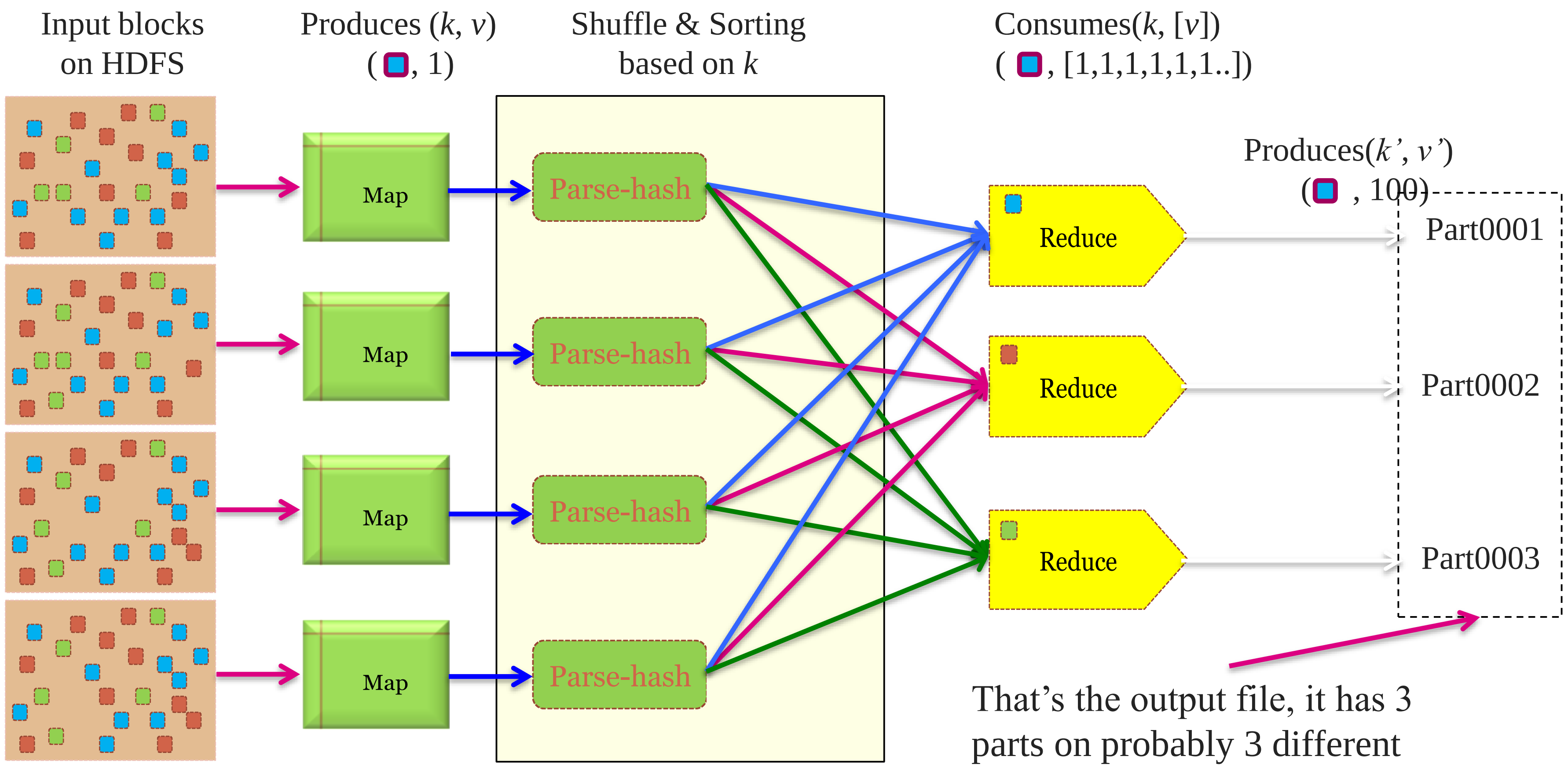


4 Map Tasks

5 Reduce Tasks

Example 2: Color Count

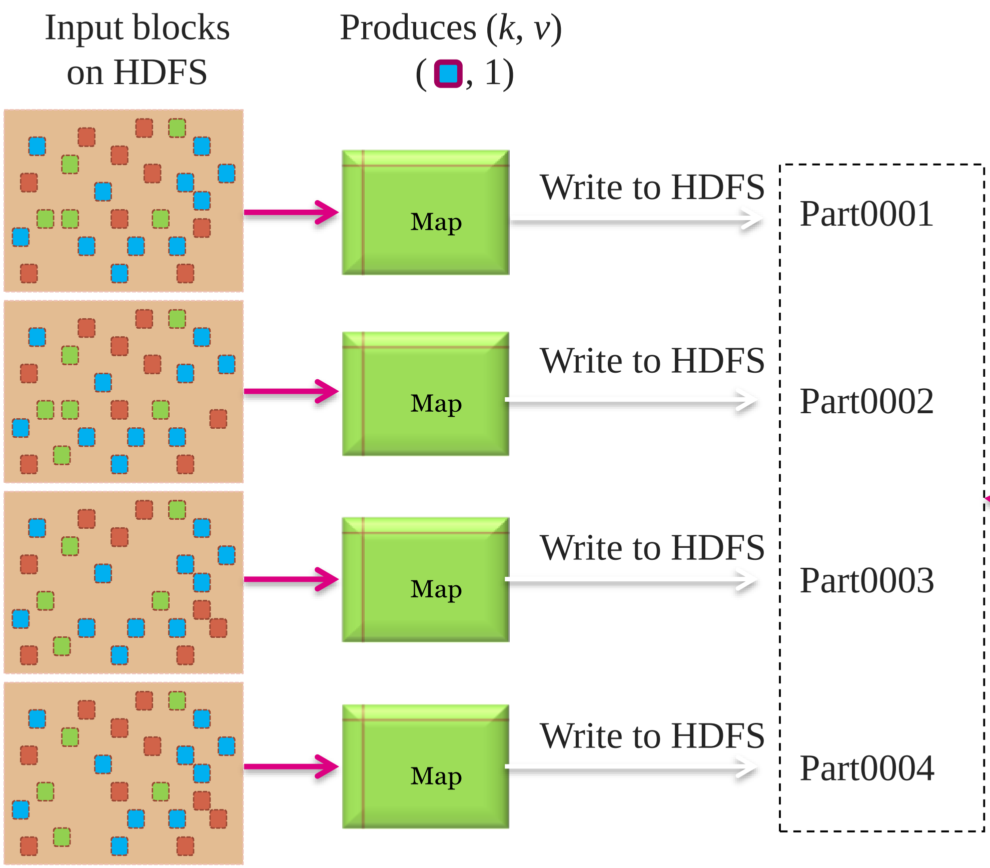
Job: Count the number of each color in a data set



That's the output file, it has 3 parts on probably 3 different machines

Example 3: Color Filter

Job: Select only the blue and the green colors



- Each map task will select only the blue and green colors
- No need for reduce phase

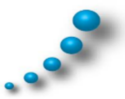
That's the output file, it has 4 parts on probably 4 different machines

Use of MapReduce/Hadoop

- Google: Inventors of MapReduce computing paradigm
- Yahoo: Developing Hadoop open-source of MapReduce
- IBM, Microsoft, Oracle, Facebook, Amazon, AOL, NetFlex, etc. adopt
- Many others + universities and research labs adopt
- Applications
  - Web applications, social networks, scientific applications, applications generating big data

Large-Scale Data Analytics

- MapReduce computing paradigm (e.g., Hadoop) vs. Traditional database systems


Scalability (petabytes of data, thousands of machines)


Flexibility in accepting all data formats (no schema)


Efficient and simple fault-tolerant mechanism


Commodity inexpensive hardware


Performance (tons of indexing, tuning, data organization tech.)

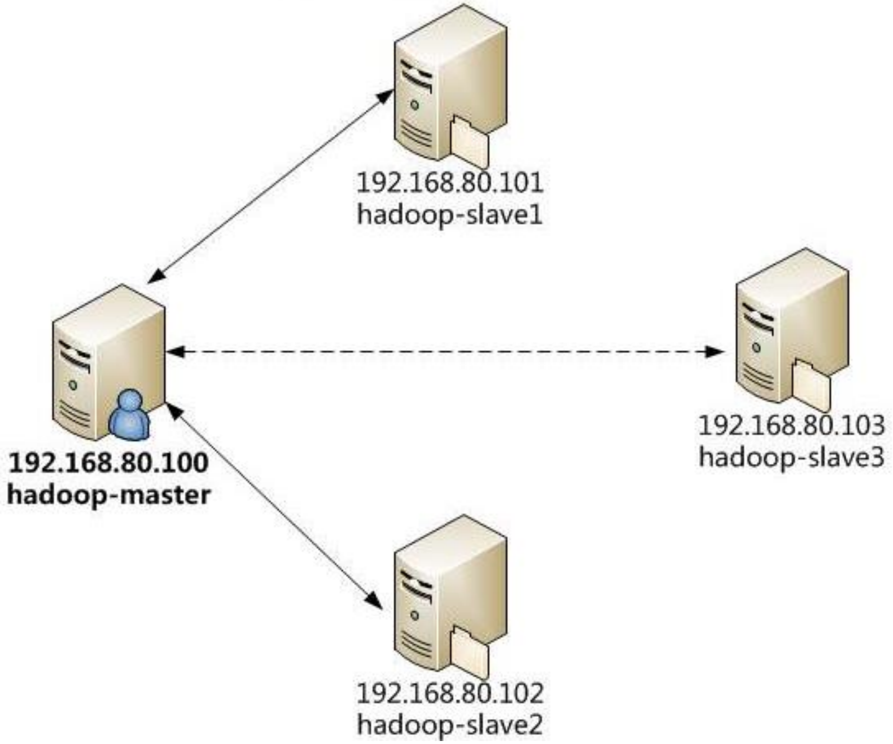

Features:
- Provenance tracking
- Annotation management
- ….

Hadoop vs. DDBS

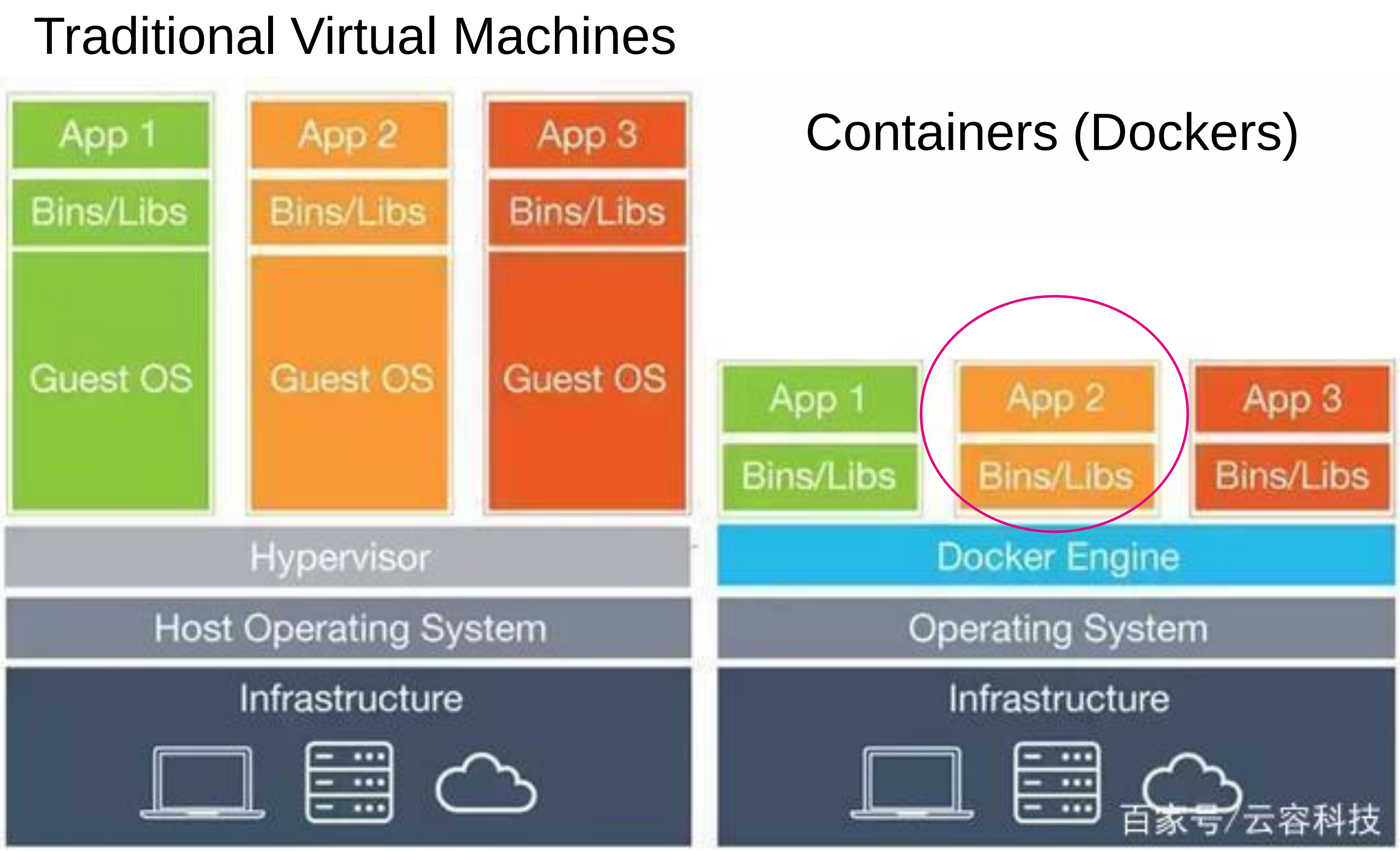
	Distributed Databases	Hadoop
Computing Model	- Notion of transactions - Transaction is the unit of work - ACID properties, Concurrency control	- Notion of jobs - Job is the unit of work - No concurrency control
Data Model	- Structured data with known schema - Read/Write mode	- Any data will fit in any format - (un)(semi)structured - Read Only mode
Cost Model	- Expensive servers	- Cheap commodity machines
Fault Tolerance	- Failures are rare - Recovery mechanisms	- Failures are common over thousands of machines - Simple yet efficient fault tolerance
Key Characteristics	- Efficiency, optimizations, fine-tuning	- Scalability, flexibility, fault tolerance

How to set up a Hadoop cluster on one machine?

- Two methods
  - Virtual machines (simulating multiple machines)
  - Containers

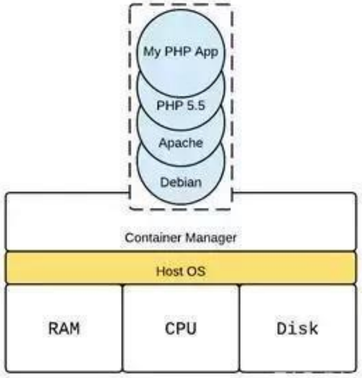


Differences between Virtual Machines and Containers



- Containers run on top of a container engine, which runs on top of an operating system.

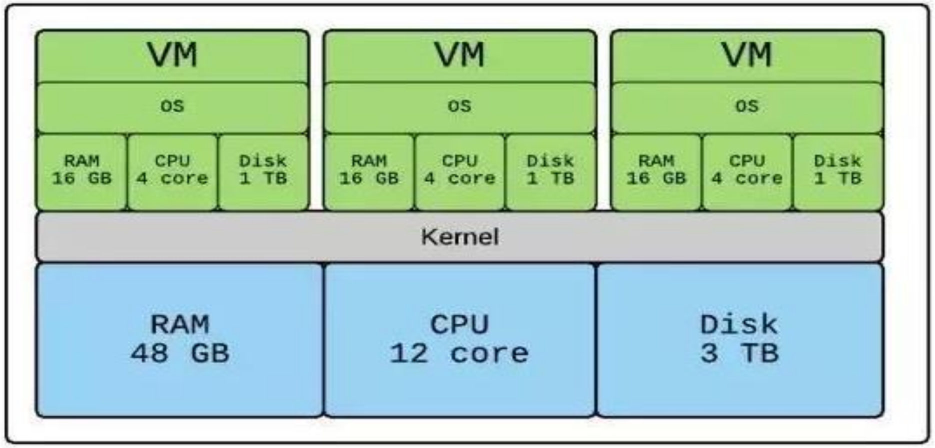
What is container technology



- Containers are sometimes called light-weight virtual machines, but they are nothing like virtual machines. A container is a “package” that only contains an application and its dependencies, nothing more.
- Containerization has become a major trend in software development as an alternative or companion to virtualization.
- It involves encapsulating or packaging up software code and all its dependencies so that it can run uniformly and consistently on any infrastructure.
- Starting a container is very quick since OS is already running.

Resource Allocation

- Each virtual machine has to be assigned fixed resources



3 virtual machines run on one computer (48 GB RAM, 12-core CPU, 3TB storage)， and each is assigned 16 GB RAM, 4-core, and 1 TB storage.

If one virtual machine needs less than 1GB RAM and 100 MB storage， extra resources are wasted.

- But containers can share CPU, memory, and storage

OS Environment

- The virtual machine contains a complete OS environment, and also provides OS control support.
- Therefore, the size of the virtual machine is relatively large.
- Before running the application, the virtual machine takes a few minutes to boot the OS before it can initialize and run the application.
- But containers are smaller in scale, and can usually be started in seconds.

Differences between Virtual Machines and Containers

	Virtual Machine	Container
Disk Storage	Very Large, GB	Small, MB - KB
Start Speed	Slow，in minutes	Fast，in seconds
Running Mode	Run on Hypervisor	Run on the host kernel
Concurrency	More than a dozen, up to dozens, on a host	Hundreds or even hundreds or thousands
Performance	Lower than host machine	Close to the local process of the host machine
Resource Utilization	Low	High

Convenience of usnig containers

- When the code is switched from a programmer's laptop to a test server, or from a physical server to a public cloud/private cloud; the runtime library version the code depends on changes (e.g., python2.7 is used for development, but python3 is used on the production machine) ; or the OS that the code runs on has changed (such as ubuntu for development, redhat for production)
- In addition to investing time in the development of the application itself, programmers also need to spend extra energy to deal with this kind of environment or platform inconsistency, which is sometimes a headache
- Containers enable application developers not to consider the underlying environments

Advantages of Containers

- Overcome the limitations of traditional virtual machines
  - troublesome to create an environment and deploy an application, and the portability of the application is also cumbersome
- Agile development
- Smooth deployment
- Version management
- Portable computing environment
- Standardization
- Safety
- High availability
- Convenient management

Techniques of Container

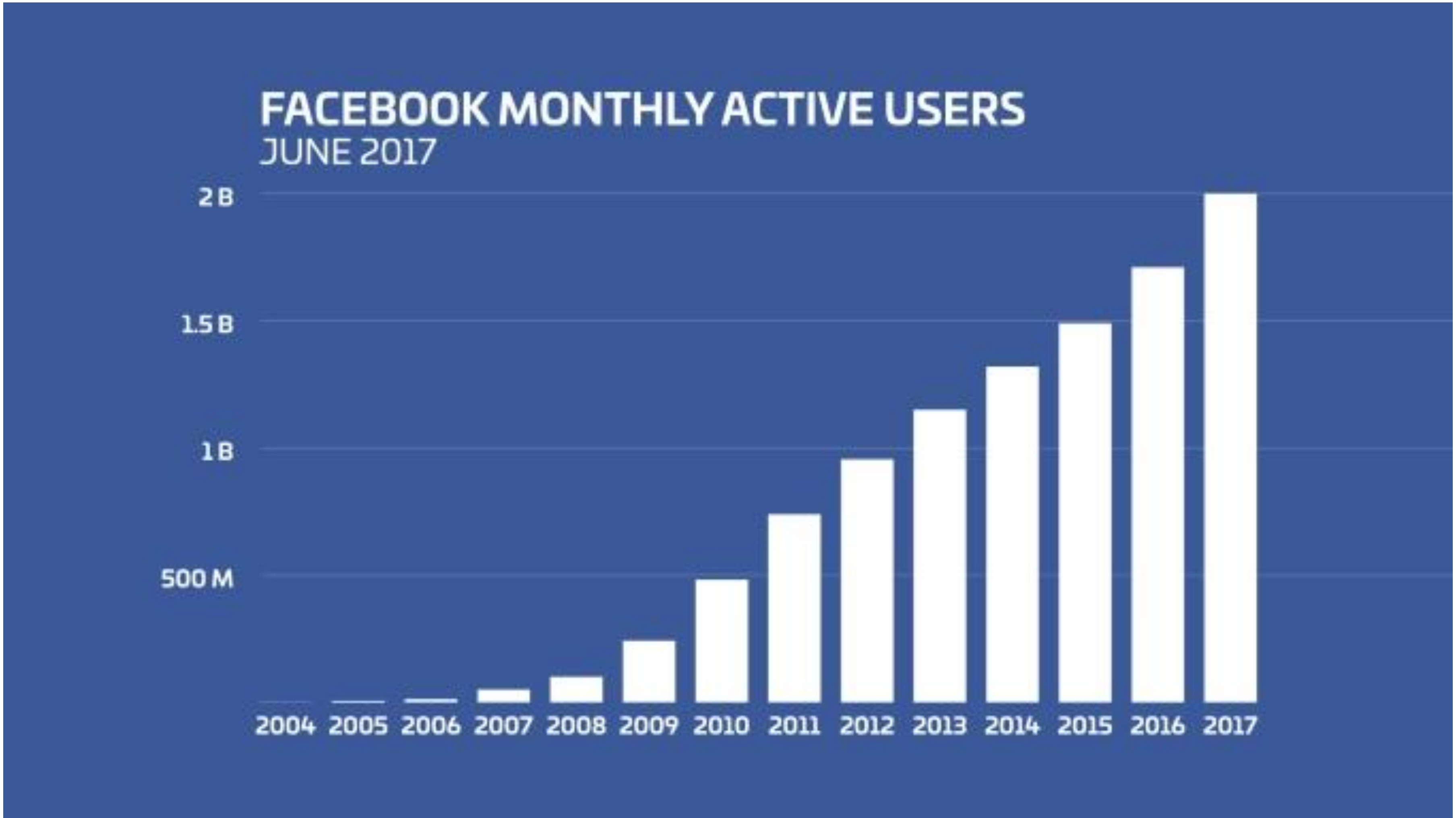
- Namespace technique
  - The Linux Namespaces mechanism is a kernel-level environment isolation method provided by Linux
  - System resources such as PID, IPC, and Network are no longer global, but belong to a specific Namespace
  - Resources in each Namespace are transparent to other Namespaces
- Cgroups technique
  - Linux Cgroups （Linux Control Group） constrain the resources (CPU, RAM, storage, Network bandwidth, etc.) that a process group can use
  - Linux Cgroups can prioritize and audit processes, and suspend and resume processes

Container technology and Docker

- Container technology has long existed, and the reason why it has not become mainstream is that it fails to provide a standardized running environment
- Docker was born in 2013，and has become synonymous with container technology
- The new generation of container technology represented by Docker aims to provide a standardized running environment from the very beginning. It achieves "build once, run anywhere". The same building version can be used for development, testing, pre-release, and any environment such as production, and achieve decoupling from the underlying OS.
- Container as Service

Another Master-Slave Fashion

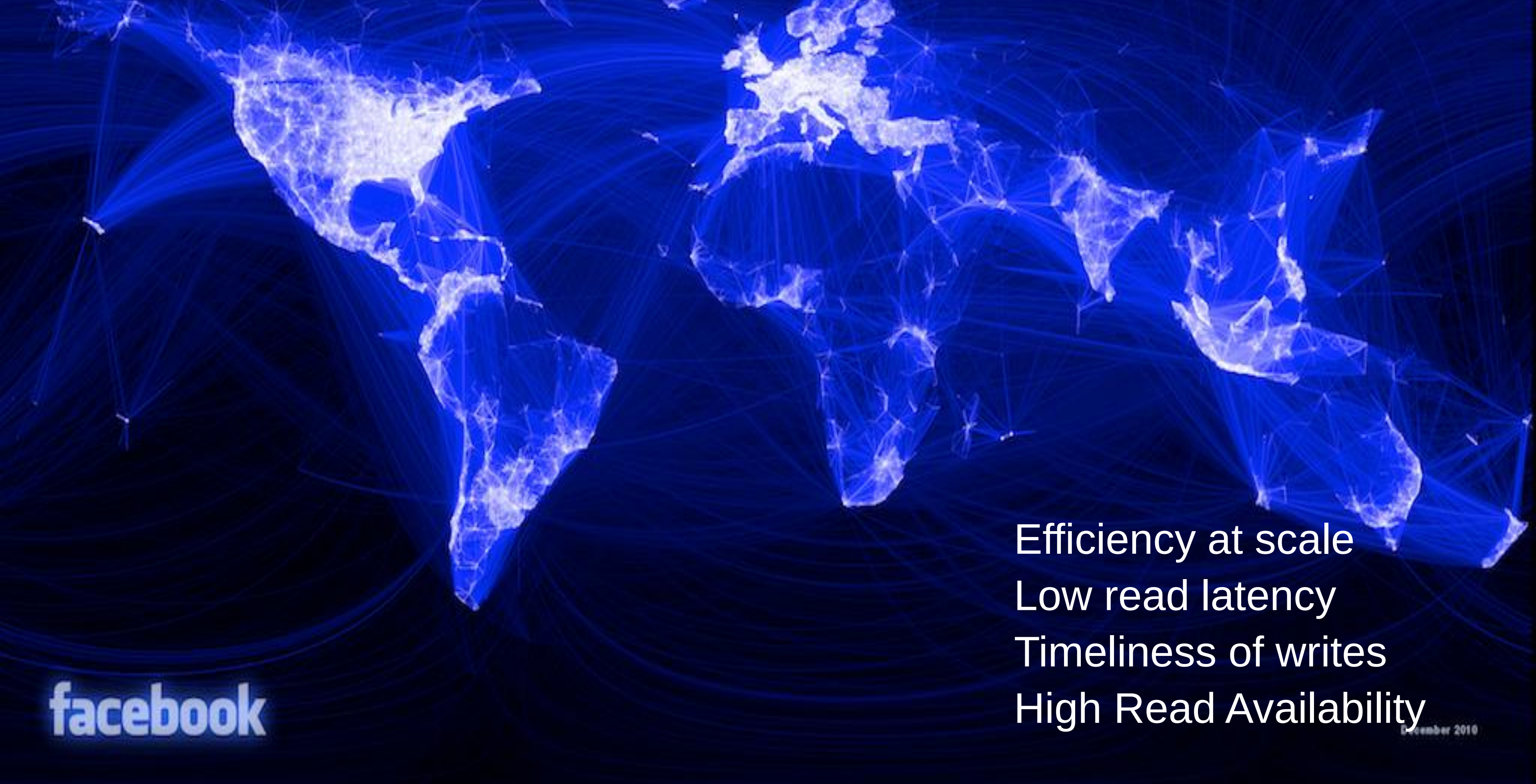
TAO - Facebook's Distributed Data Store for the Social Graph



TAO’s Goals/Challenges

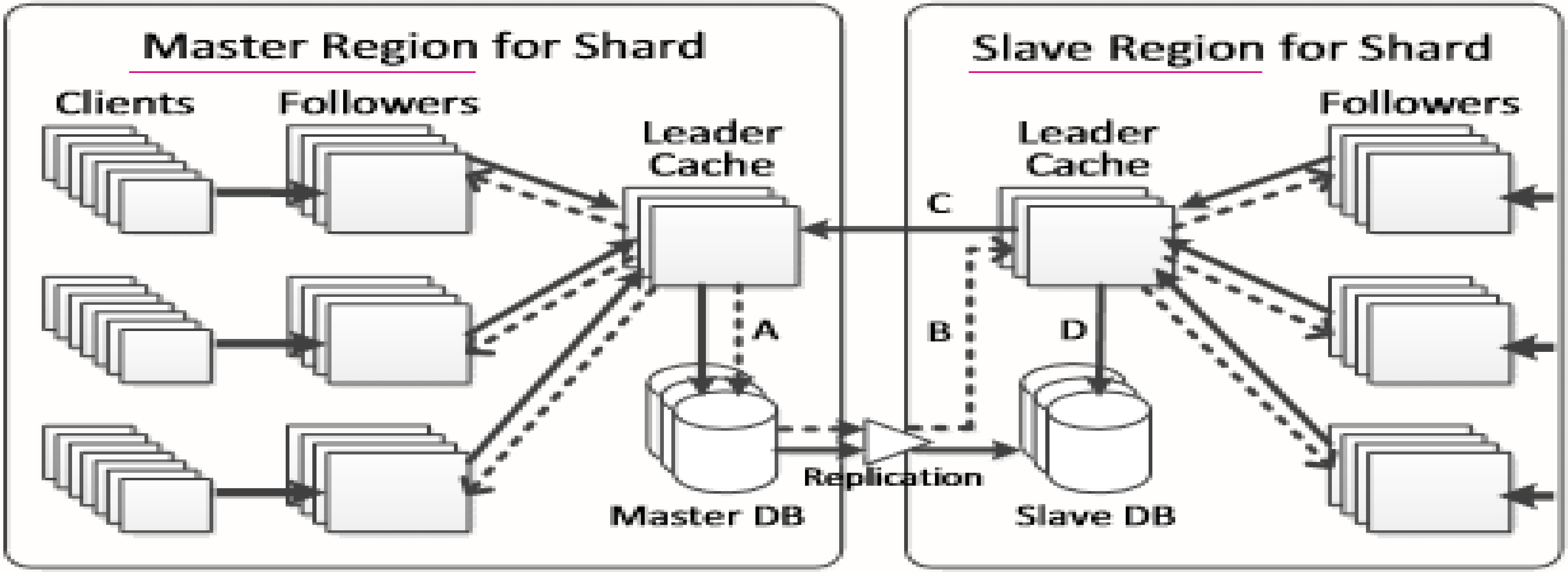
Billions of users, billions of data accesses to the social graph

How to handle this huge workload efficiently?



- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability

Master-Slave Architecture



Why "Regions" ?

Reason: network latency.

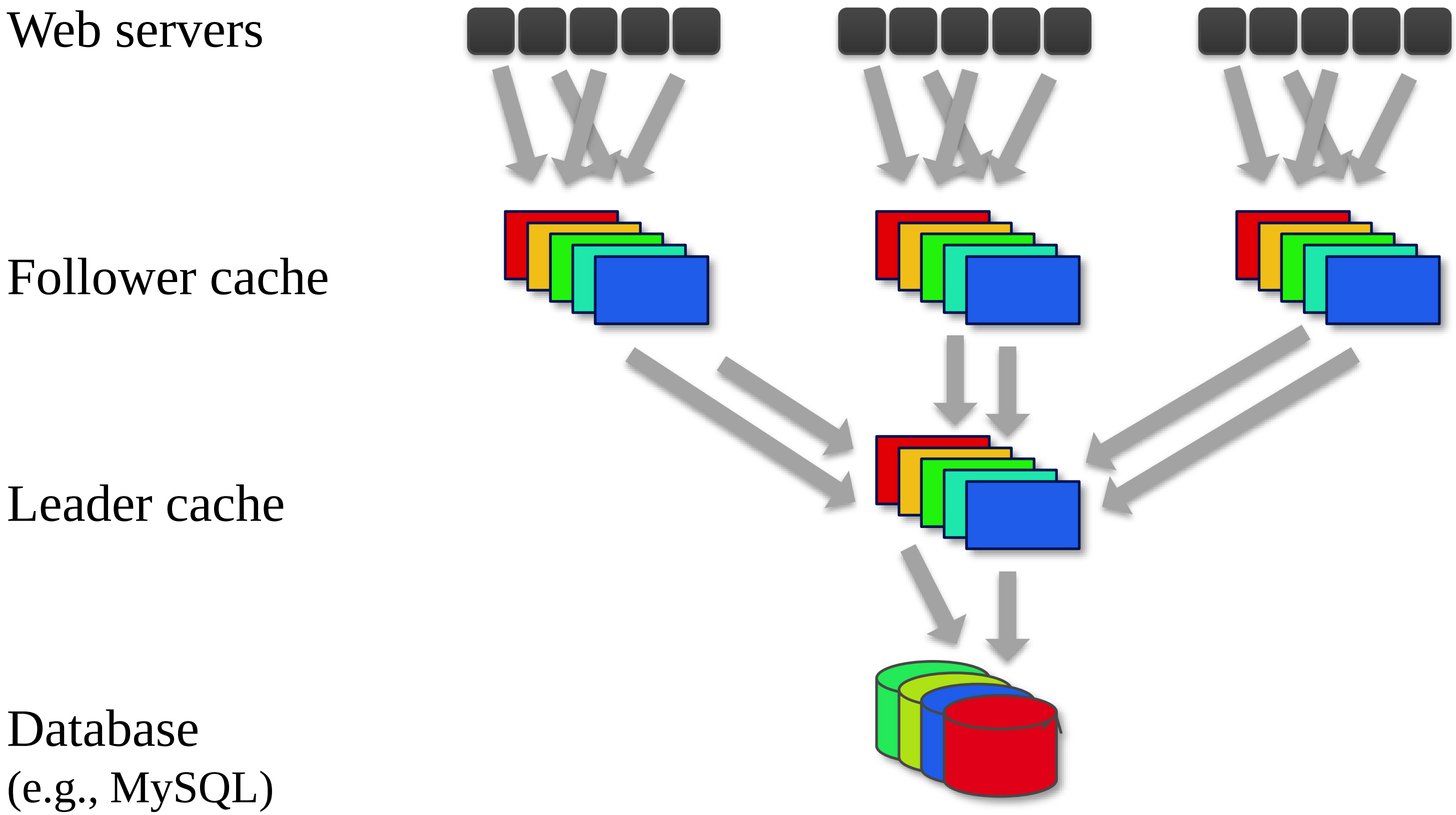
- Each region has one Leader tier and multiple Followers
- Master Region vs. Slave Region
- Master/Slave designation is made separately for each shard

Why Master/Slave ?

Reason: read misses are 25x more frequent than writes. In this way, Slave Regions can entirely self-service read misses without bothering Master Region (via Slave leader).

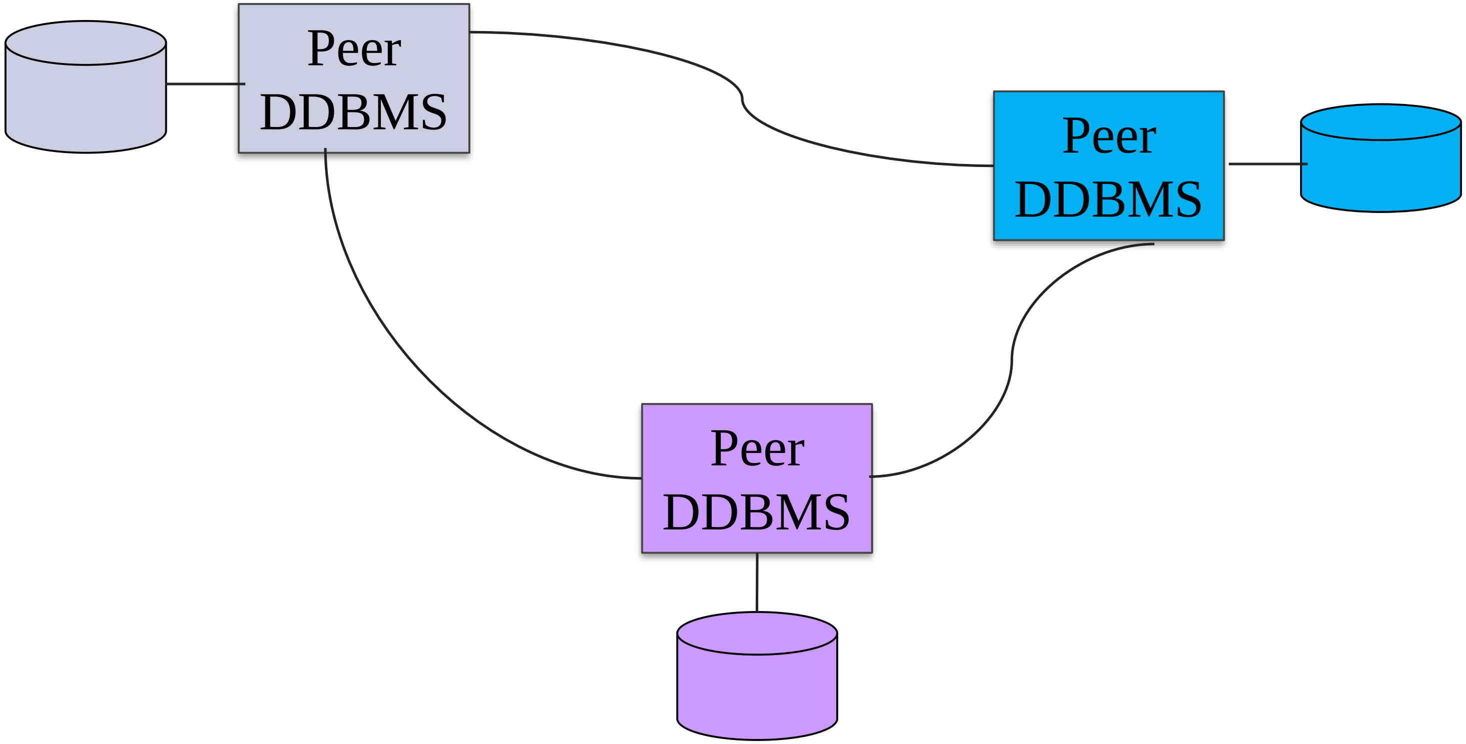
- Writes in a Slave Region go to Master Region
- Consistency is handled asynchronously by sending cache maintenance messages from the Master to Slaves

TAO's Follower and Leader Caches

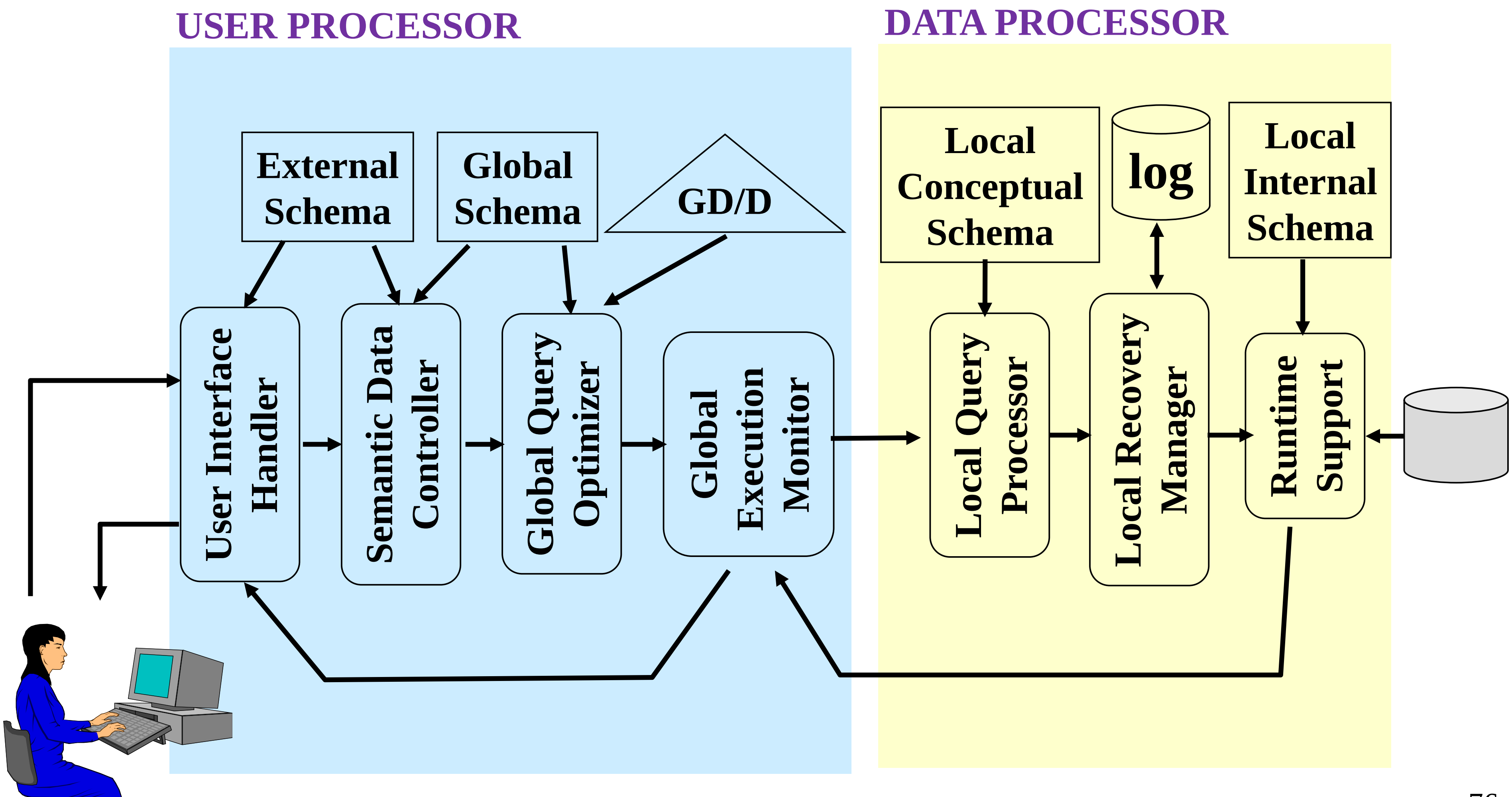


1. Client request goes to a single cache server in a local "follower" tier
2. Follower fulfills request. If read miss or a write, go to a Leader
3. If Leader read miss or a write, Leader will go to the DB

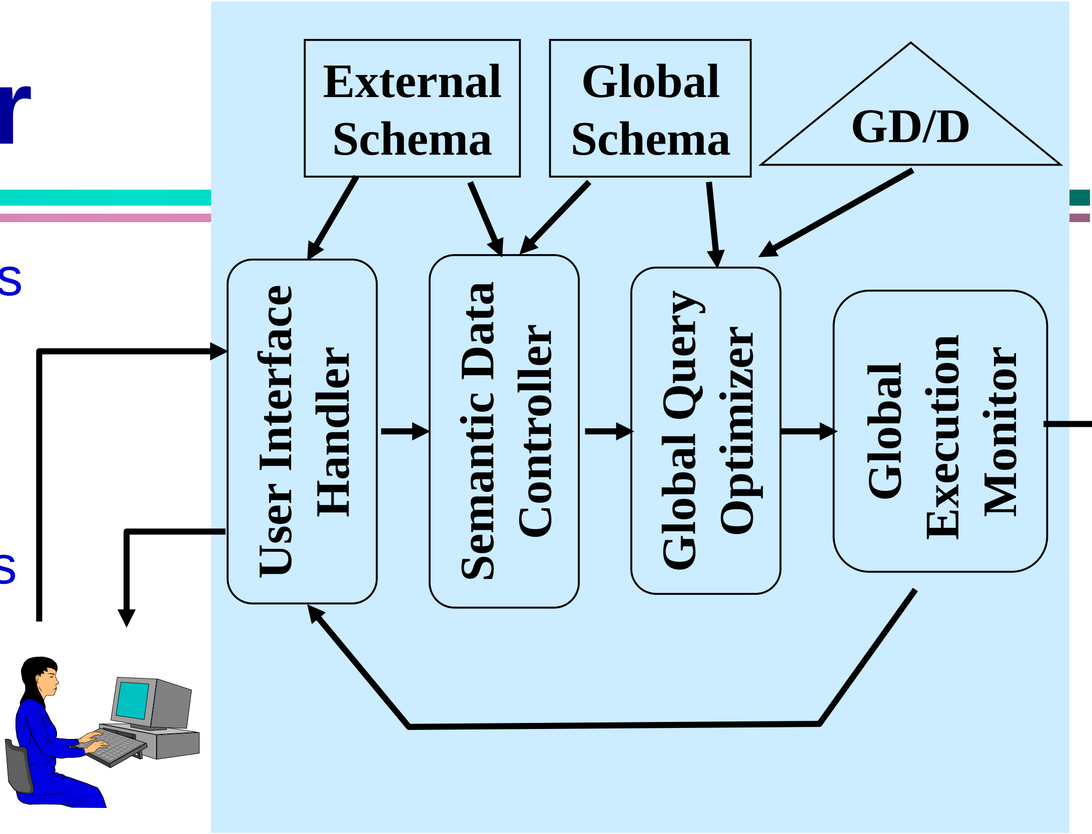
3 - Peer-to-Peer Distributed Architecture



A Peer DBMS


USER PROCESSOR
DATA PROCESSOR
External Schema
Global Schema
GD/D
Local Conceptual Schema
log
Local Internal Schema
User Interface Handler
Semantic Data Controller
Global Query Optimizer
Global Execution Monitor
Local Query Processor
Local Recovery Manager
Runtime Support

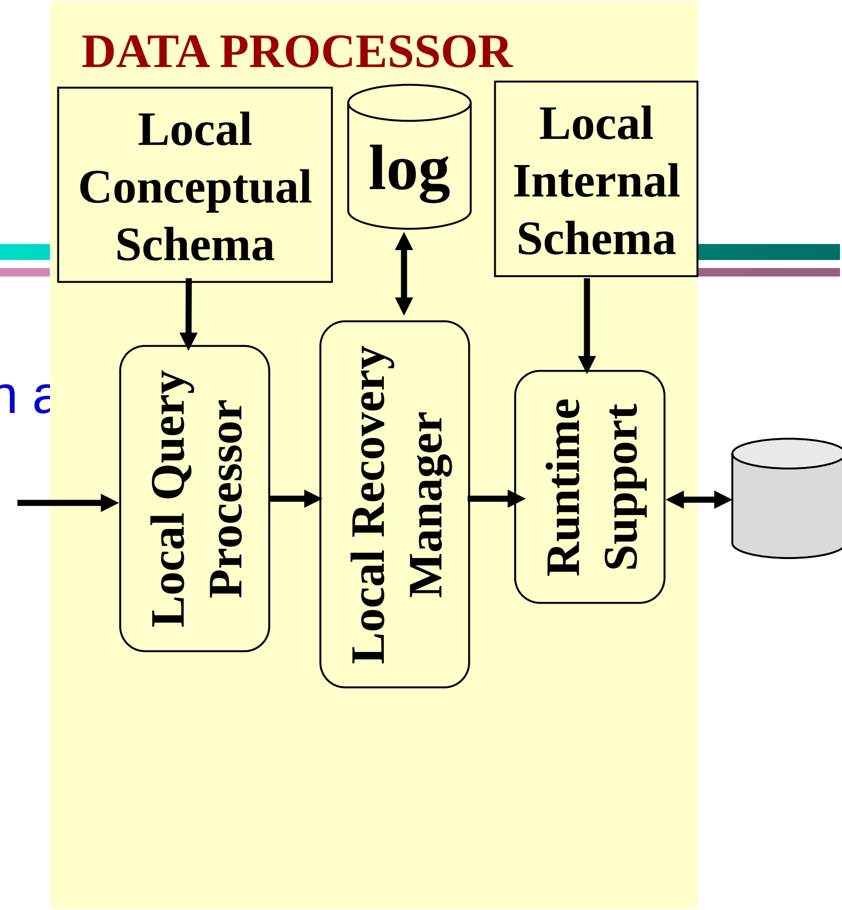
User Processor



- User interface handler interprets user commands and formats the result data as it is sent to the user.
- Semantic data controller checks the integrity constraints and authorization requirements.
- Global query optimizer and decomposer determines execution strategy, translates global queries to local queries, and generates strategy for distributed join operations.
- Global execution monitor (distributed transaction manager) coordinates the distributed execution of the user request.

Data Processor

- Local query processor selects the access path and performs local query optimization a
- Local recovery manager maintains local database consistency.
- Run-time support processor physically accesses the database. It is the interface to the OS and contains database buffer manager.

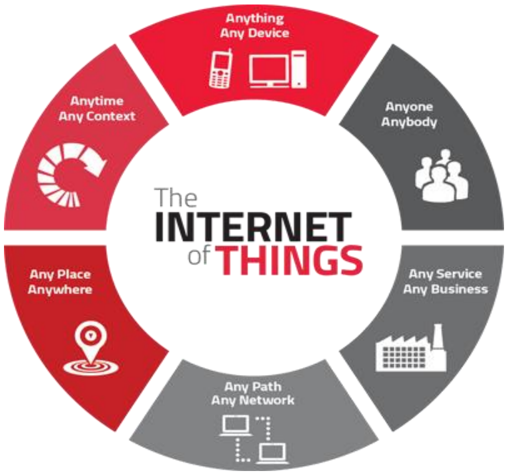


Apache Cassandra: Peer-to-Peer

- A peer-to-peer distributed storage system for managing very large amounts of data spread out across many commodity servers, while providing highly available service with no single point of failure.
- Originally developed at Facebook, open sourced in 2008, and became a top-level Apache project in 2010.
- Deployed as the backend storage system for multiple services in Twitter, Reddit, Netflix, Digg, Rackspace, Cisco and more companies that have large, active data sets.

Cassandra

Application Areas of Cassandra



Internet of things applications

Perfect for consuming lots of fast incoming data from devices, sensors, and similar mechanisms that exist in many different locations.

Messaging Apps

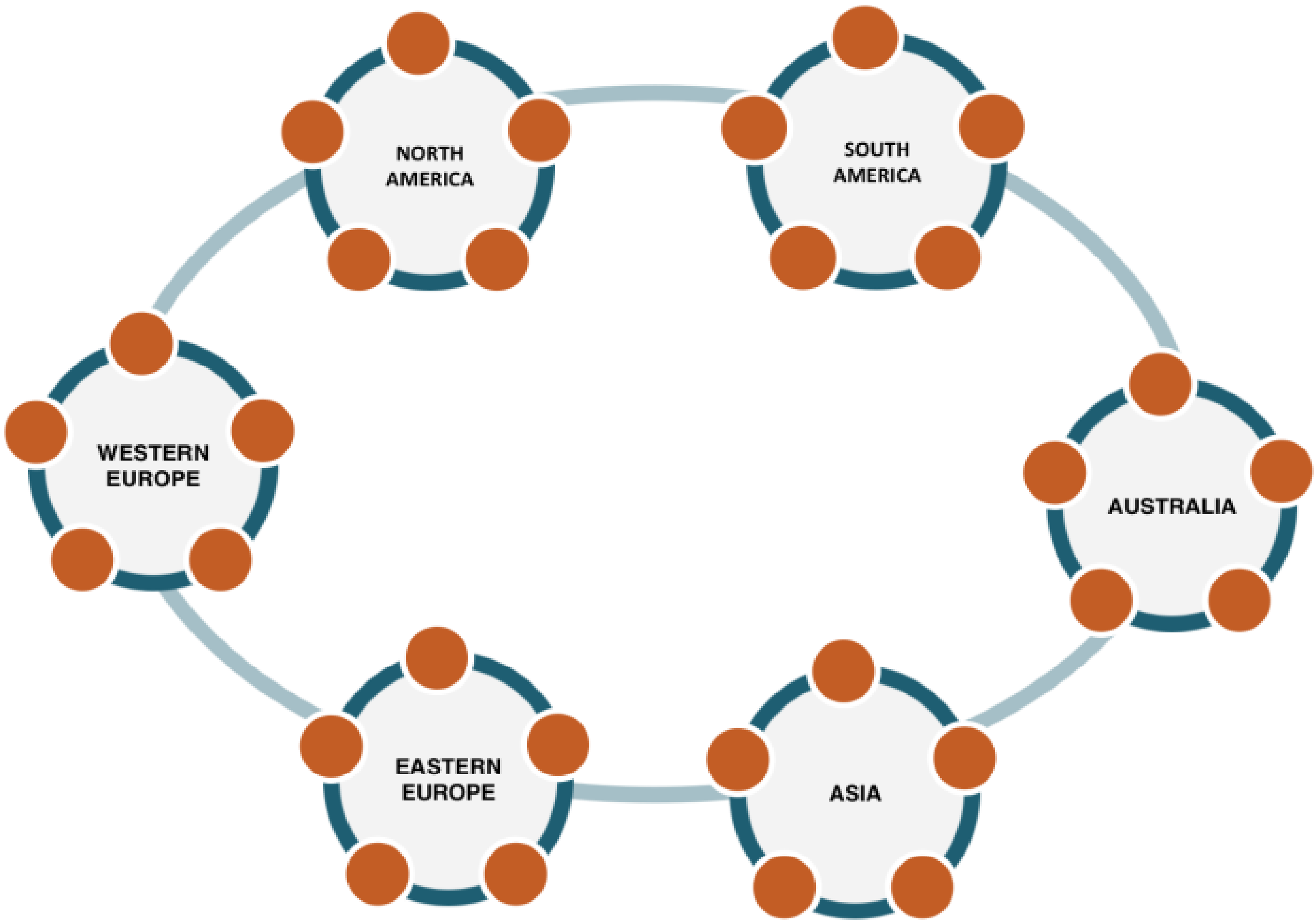
Serves as the database backbone for numerous mobile phones and messaging providers' applications.

Social media analytics and recommendation engines

Many online companies, websites, and social media providers use Cassandra to ingest, analyze, and provide analysis and recommendations to their customers.

Architecture of Cassandra

Masterless "ring" architecture. All nodes play an identical role.



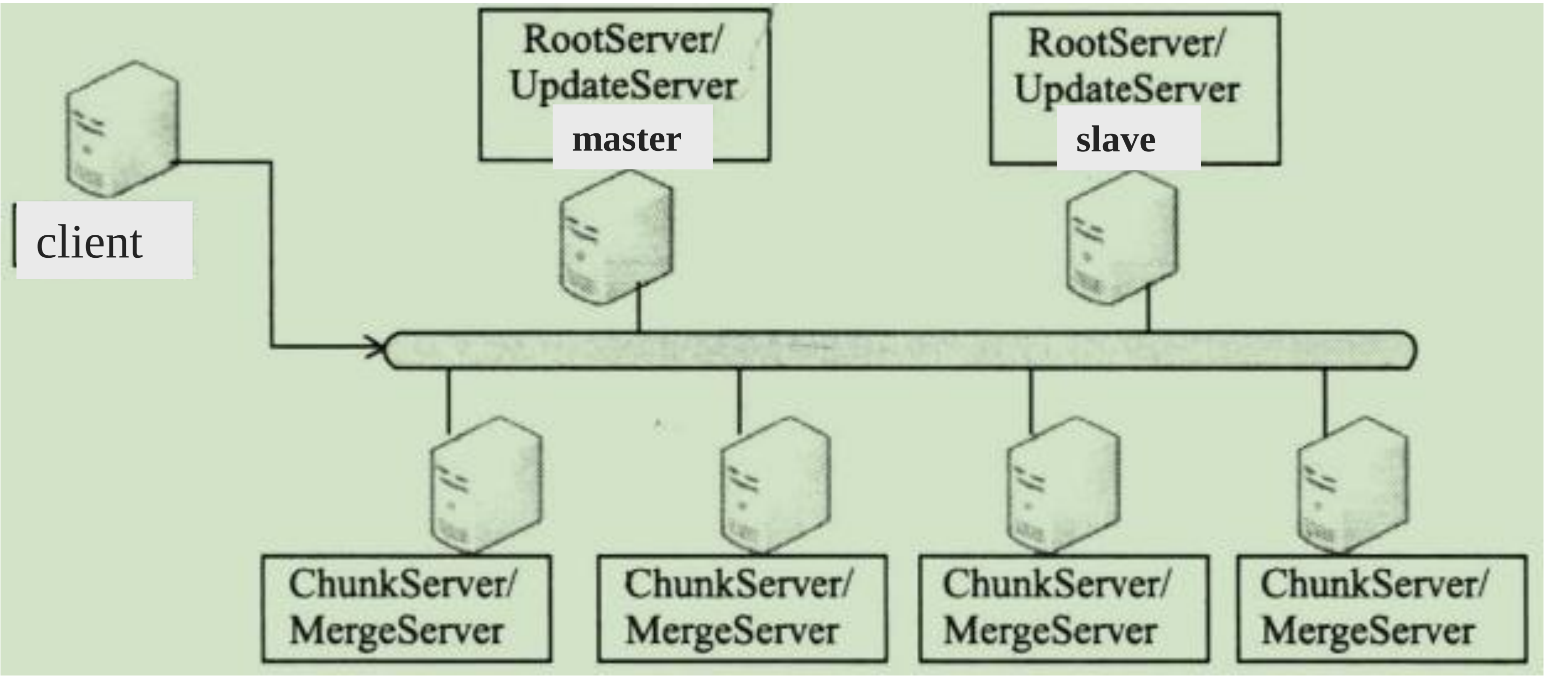
Gossip

- Peer to peer protocol
- Gossip every one second, up to 3 others nodes
- Share state information of the nodes

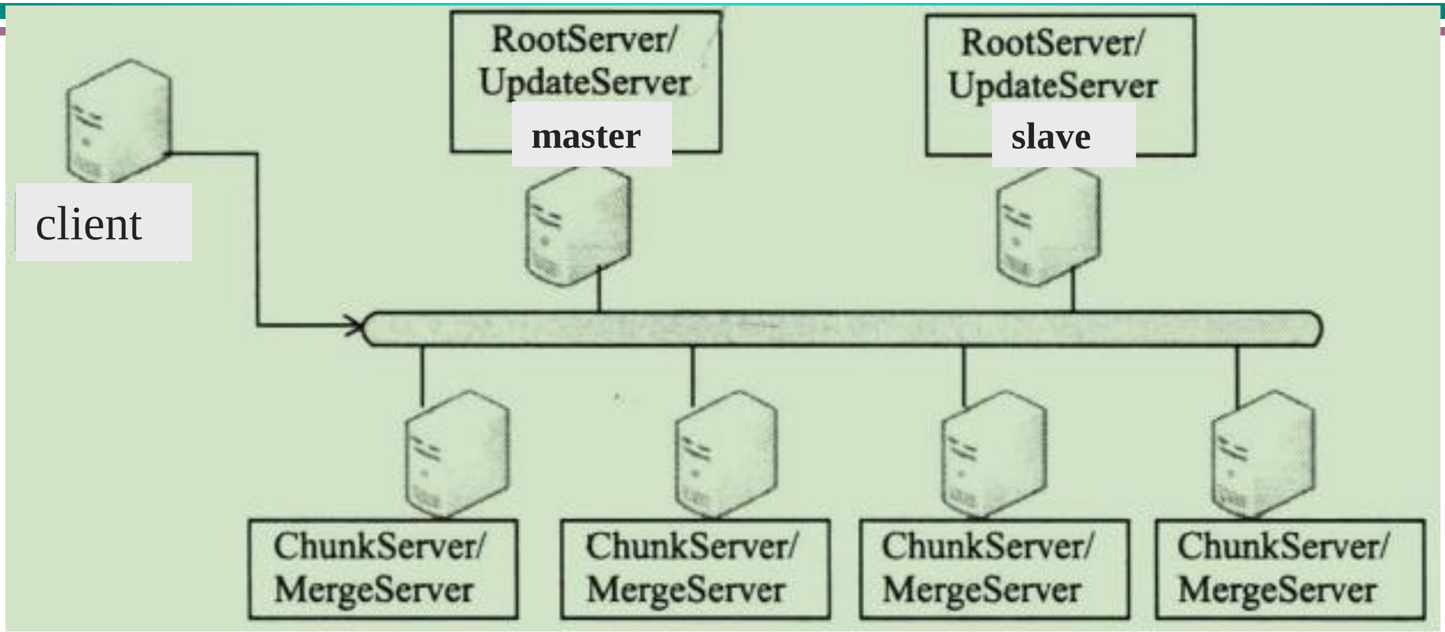
- Achieve three major functions
  - Failure detection
  - Dynamic load balance
  - Elastic expansion without central control

Oceanbase Architecture

- **RootServer:** responsible for data partition, load balancing, and cluster server management. A master and a slave, which have strong data synchronization.
- **UpdateServer:** storing incremental update information. A master and a slave, which can have different data synchronization modes.
- Processes of UpdateServer and RootServer run on the same (server) machine.

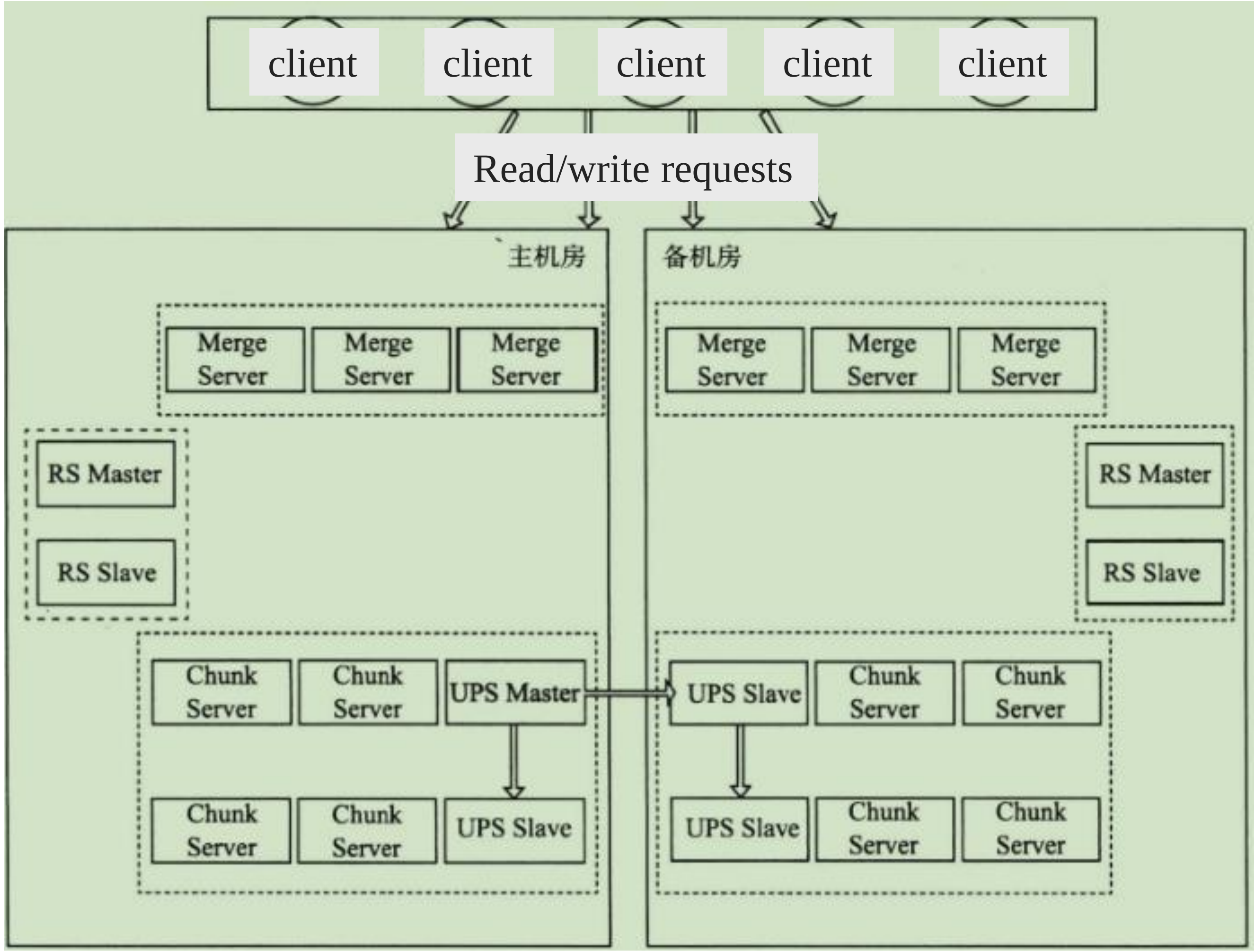


Oceanbase Architecture



- **ChunkServer：** storing two or three copies of base data;
- **MergeServer：** receiving clients SQL requests, and after syntactic and semantic analysis and query optimization, sending the requests to the corresponding ChunkServer or UpdateServer.
  If data is distributed on multiple ChunkServers, MergeServer needs to combine the results from multiple ChunkServers;
- Clients communicate with MergeServer through MySQL protocol

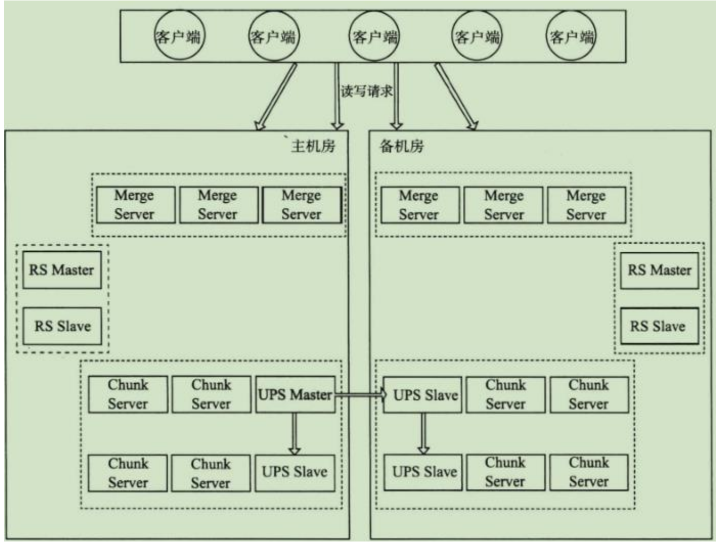
Oceanbase Architecture



Master cluster computer room **Slave cluster computer room**

Oceanbase Architecture

- OceanBase supports multiple computer rooms.
- Each computer room deploys a complete oceanbase cluster (rootserver, mergeserver, chunkserver, and updateserver)
- Each cluster is responsible for data partition, load balancing, and cluster server management by its own rootserver;
- The client is configured with RootServer address lists of multiple clusters. Users can set the traffic distribution ratio of each cluster, and the client sends read/write requests to different clusters according to this ratio;



- Data synchronization between clusters is achieved through the incremental update operation log of the master UpdateServer of the master cluster to the slave cluster.

Dimension 2: Heterogeneity (异质)

- Various levels (hardware, communication, OS)
- DBMS important ones (like data model, query language, transaction management algorithms, etc.)
  - 0 - homogeneous
  - 1 - heterogeneous

Dimension 3: Autonomy (自治)

- Refer to the distribution of control, not of data, indicating the degree to which individual DBMSs can operate independently.
- Requirements of an autonomous system
  - The local operations of the individual DBMSs are not affected by their participation in the DDBS.
  - The individual DBMS query processing and optimization should not be affected by the execution of global queries that access multiple databases.
  - System consistency or operation should not be compromised when individual DBMSs join or leave the distributed database confederation.

Three Versions of Autonomy

- Design autonomy
  - Ability of a component DBMS to decide on issues related to its own design
  - Freedom for individual DBMSs to use data models and transaction management techniques they prefer
- Communication autonomy
  - Ability of a component DBMS to decide whether and how to communicate with other DBMSs
  - Freedom for individual DBMSs to decide what information (data & control) is to be exported
- Execution autonomy
  - Ability of a component DBMS to execute local operations in any manner it wants to.
  - Freedom for individual DBMSs to execute transactions submitted in any way that it wants to

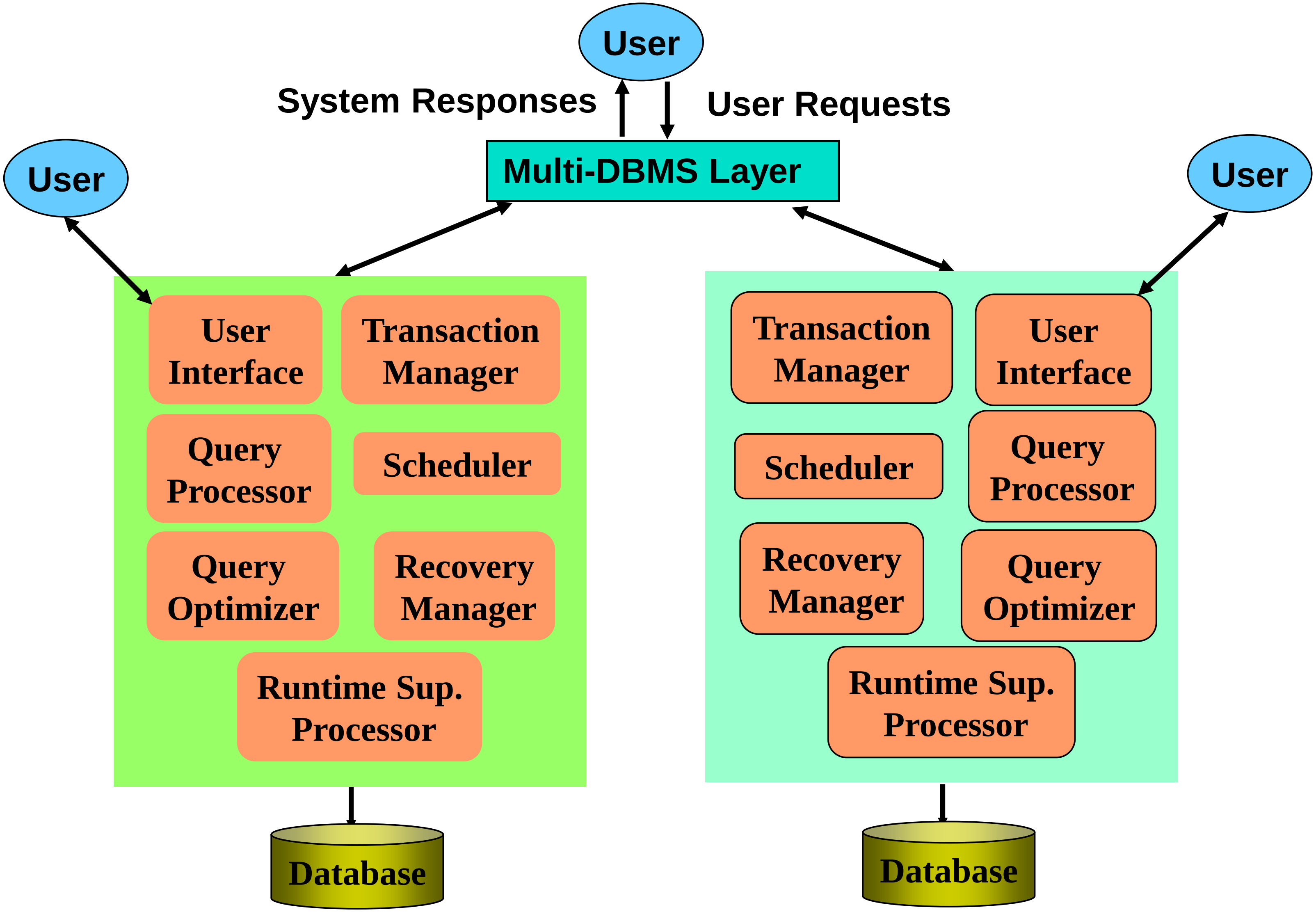
Dimension 3: Autonomy (*cont*.)

- 0 – Tightly coupled - integrated
- 1 – Semi-autonomous - federated
- 2 – Total Isolation – multi-database systems

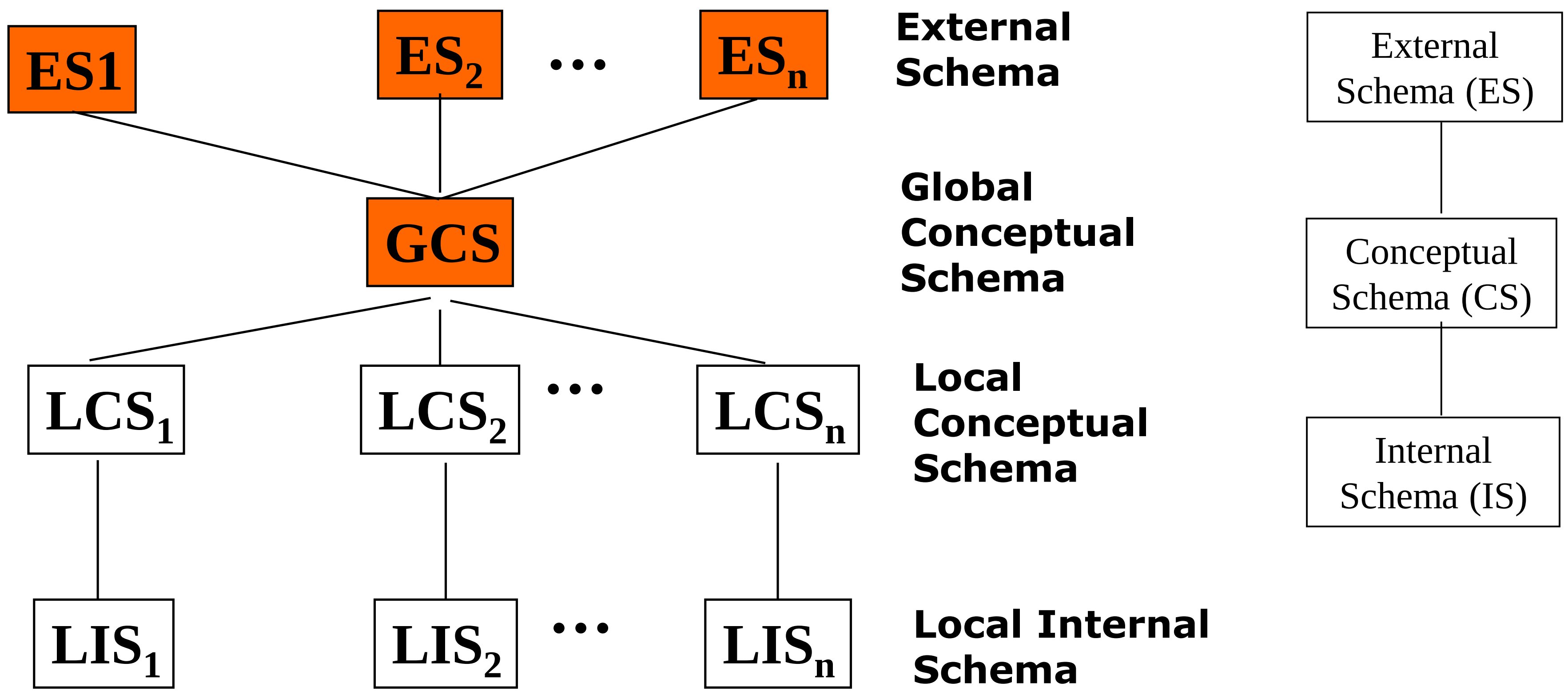
Taxonomy of Distributed Databases

- Composite DBMSs - tight integration
  - single image of entire database is available to any user
  - homogeneous or heterogeneous
- Federated DBMSs – semi-autonomous
  - DBMSs that can operate independently, but have decided to make some parts of their local data shareable
  - Need to be modified to enable them to exchange information
- Multi-DBMSs - total isolation
  - individual systems are stand alone DBMSs, which know neither the existence of other databases or how to communicate with them
  - no global control over the execution of individual DBMSs.
  - homogeneous or heterogeneous

Components of a Multi-DBMS

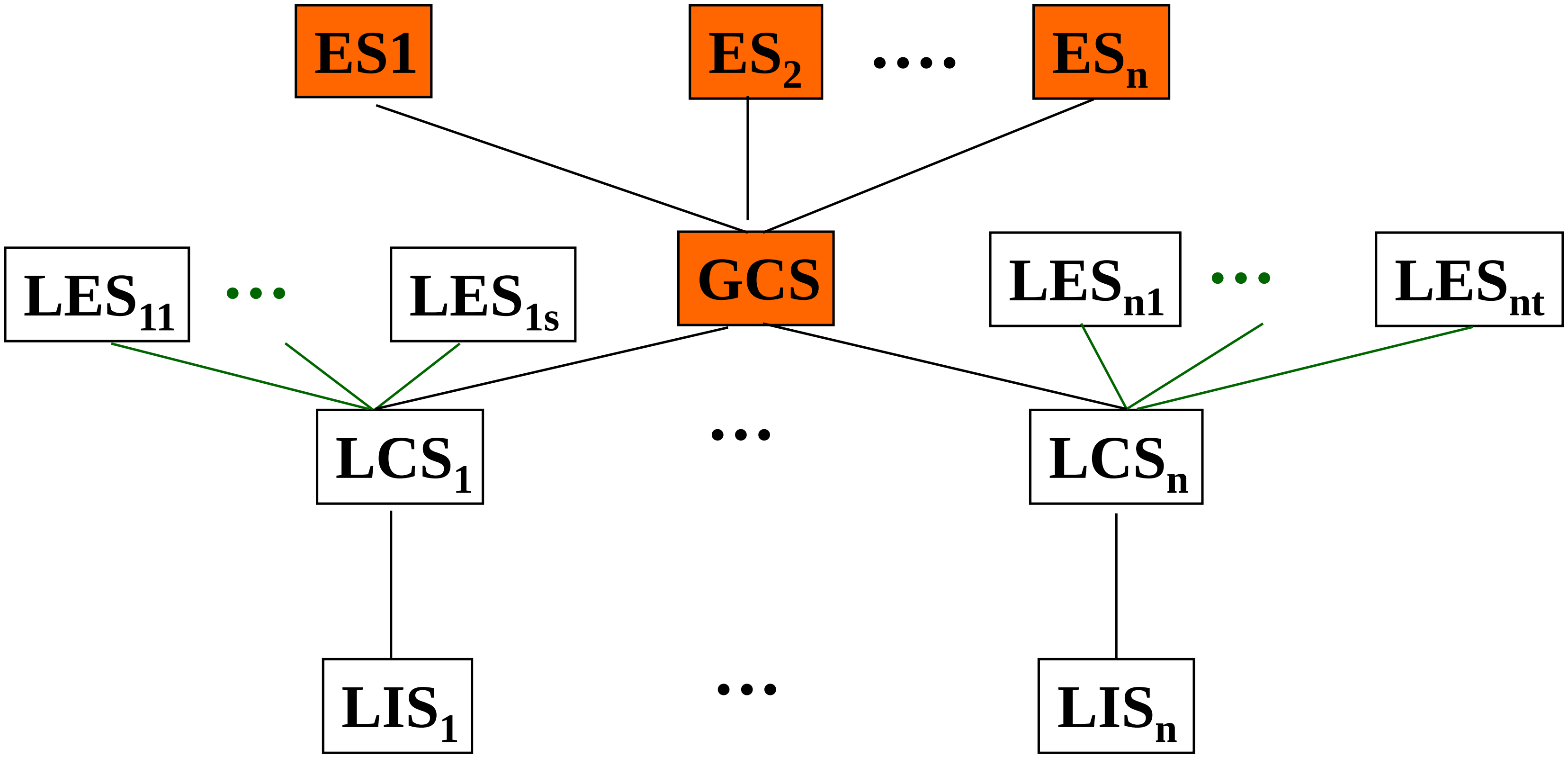


Distributed Database Reference Architecture



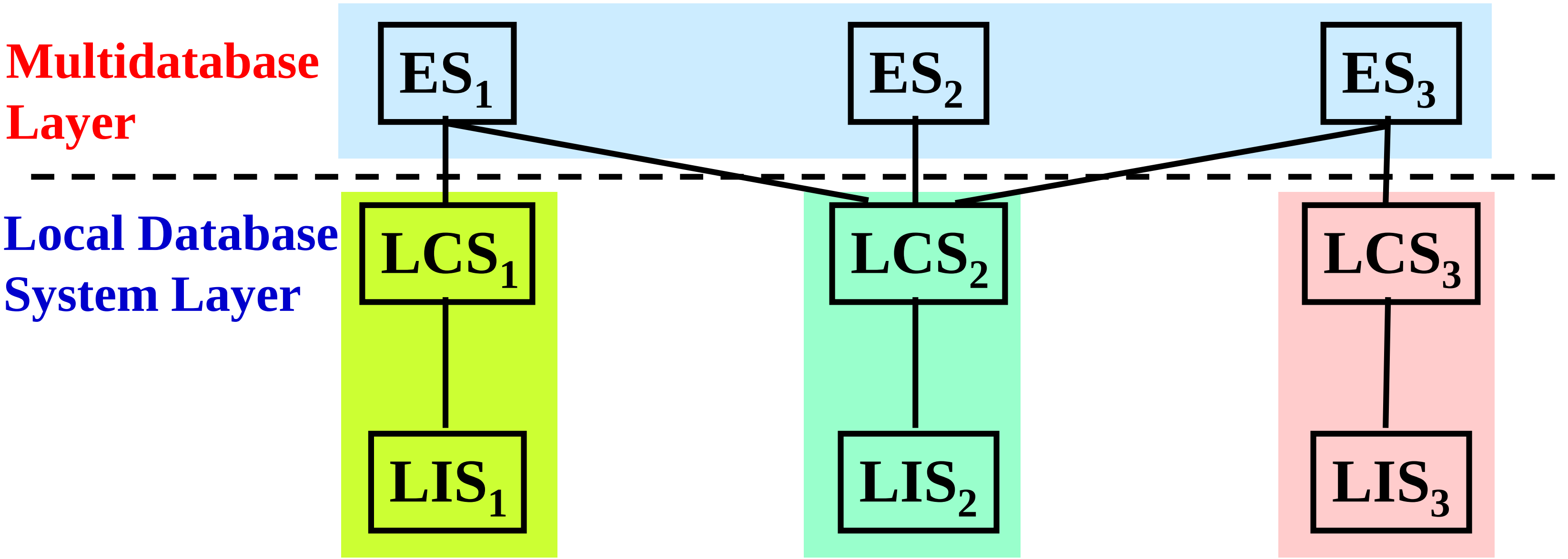
Logically integrated, and provide the levels of transparency

Multi-DBMS Architecture with a Global Conceptual Schema



- The GCS is generated by integrating LES's or LCS's
- The users of a local DBMS can maintain their autonomy
- Design of GCS is bottom-up

Multi-DBMS without a Global Conceptual Schema



- Local database system layer consists of several DBMSs which present to multidatabase layer part of their databases
- The shared database has either local conceptual schema or external schema
- External views on one or more LCSs.
- Access to multiple databases through application programs

Multi-DBMS without a Global Conceptual Schema (*cont*.)

- Multi-DBMS components architecture
  - Existence of fully fledged local DBMSs
  - Multi-DBMS is a layer on top of individual DBMSs that support access to different databases
  - The complexity of the layer depends on existence of GCS and heterogeneity
- Federated Database Systems
  - Do not use global conceptual schema
  - Each local DBMS defines export schema
  - Global database is a union of export schemas
  - Each application accesses global database through import schema (external view)

Outline

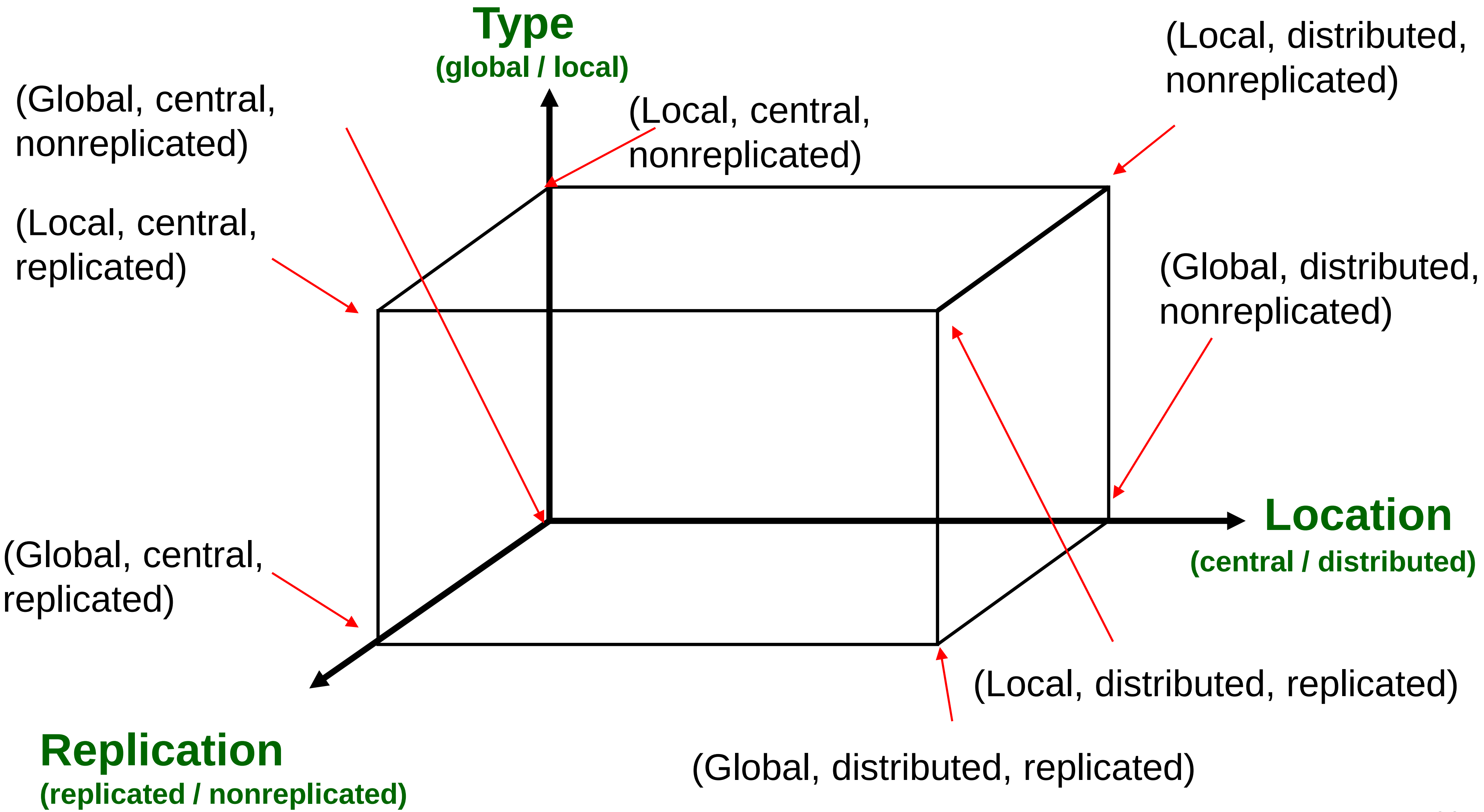
- Introduction
- Top-Down Design of DDBMS Architecture
  - Schema and Distribution Transparency
- Bottom-up Design of DDBMS Architecture
  - Architectural Alternatives for DDBMSs
  - Reference Architectures for a DDBMS
- ☞ Global Directory/Dictionary

Global Directory/Dictionary

- Directory is itself a database that contains meta-data about the actual data stored in the database. It includes the support for fragmentation transparency in the classical DDBMS architecture.
- Directory can be local or distributed.
- Directory can be replicated and/or partitioned.

- Directory issues are very important for large multi-database applications, such as digital libraries.

Alternative Directory Management Strategies



Question & Answer