Distributed architectures for databases

Introduction

The need for distributed architectures for data management comes from:

- performance requirements
- presence (historical) of distributed data that can not be centralized
    - sectoral data management
    - privacy issues

A first definition: homogeneous and heterogeneous database

- In a homogeneous distributed database
  - All nodes have the same software (DBMS)
  - All nodes are willing to cooperate in order to process requests from users
  - Each node gives part of its autonomy (in transaction management) to other nodes.
  - To the user it appears as a single database

A first definition: homogeneous and heterogeneous database

- In a heterogeneous distributed database
  - Different nodes may use different software (DBMS) and schemes
    The difference in the scheme is the main problem for
    - the execution of the query
    - transaction management
  - The nodes may not be aware of the existence of the other nodes and could provide for cooperation only a limited range of features than those in their potentiality

Introduction

The problems that arise when multiple systems interact with each other are mainly:

- *portability*, or the ability to “porting” programs from one environment to another (*at compile time*)
- *interoperability*, or the ability to make heterogeneous systems to interact with each other (*at runtime*)

In the case of databases, both properties depend on the presence of standards such as SQL, JDBC (if respected)

- Problem: to guarantee the properties *ACID* in distributed environments

Paradigms for data distribution

Client-server architecture: separation between client and server in the database

Distributed Databases: the same application uses different database servers

Parallel DataBases: different storage media and processors used in parallel to improve performance

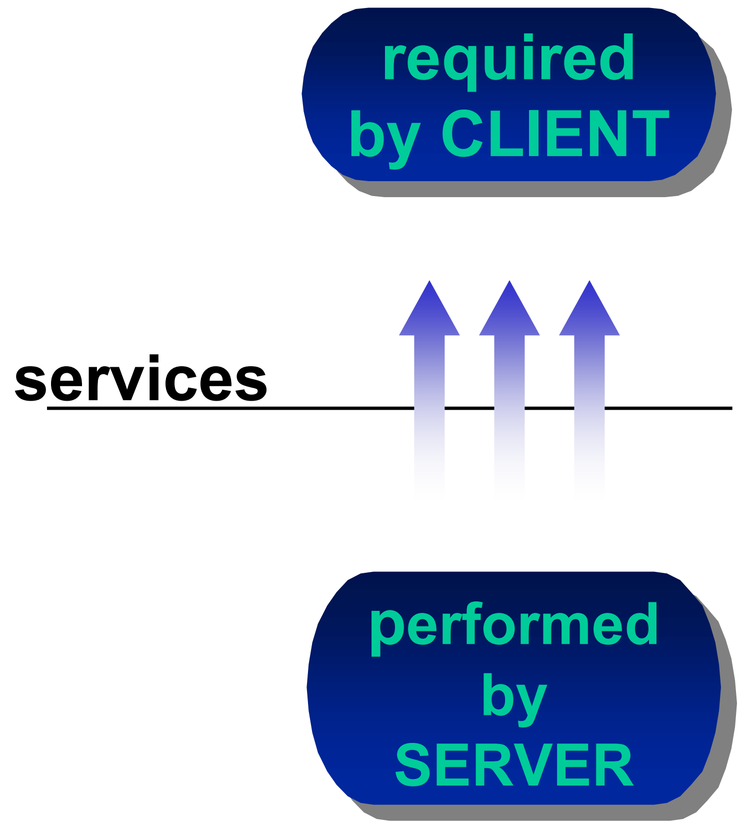
Replicated Databases: Data that logically represent the same information is physically stored on different servers

Client-server Paradigm

Technique to structure software systems

- An "interface of services " is made "public"

- Two types of systems:
  CLIENT(S)
  requires the services
  DB SERVER(S)
  provides the services



Client-server in information systems

Ideal functional separation

- CLIENT: Presentation of information
- SERVER: Data Management

In the context of databases, the roles of client and server are quite distinct and SQL provides considerable help in this direction

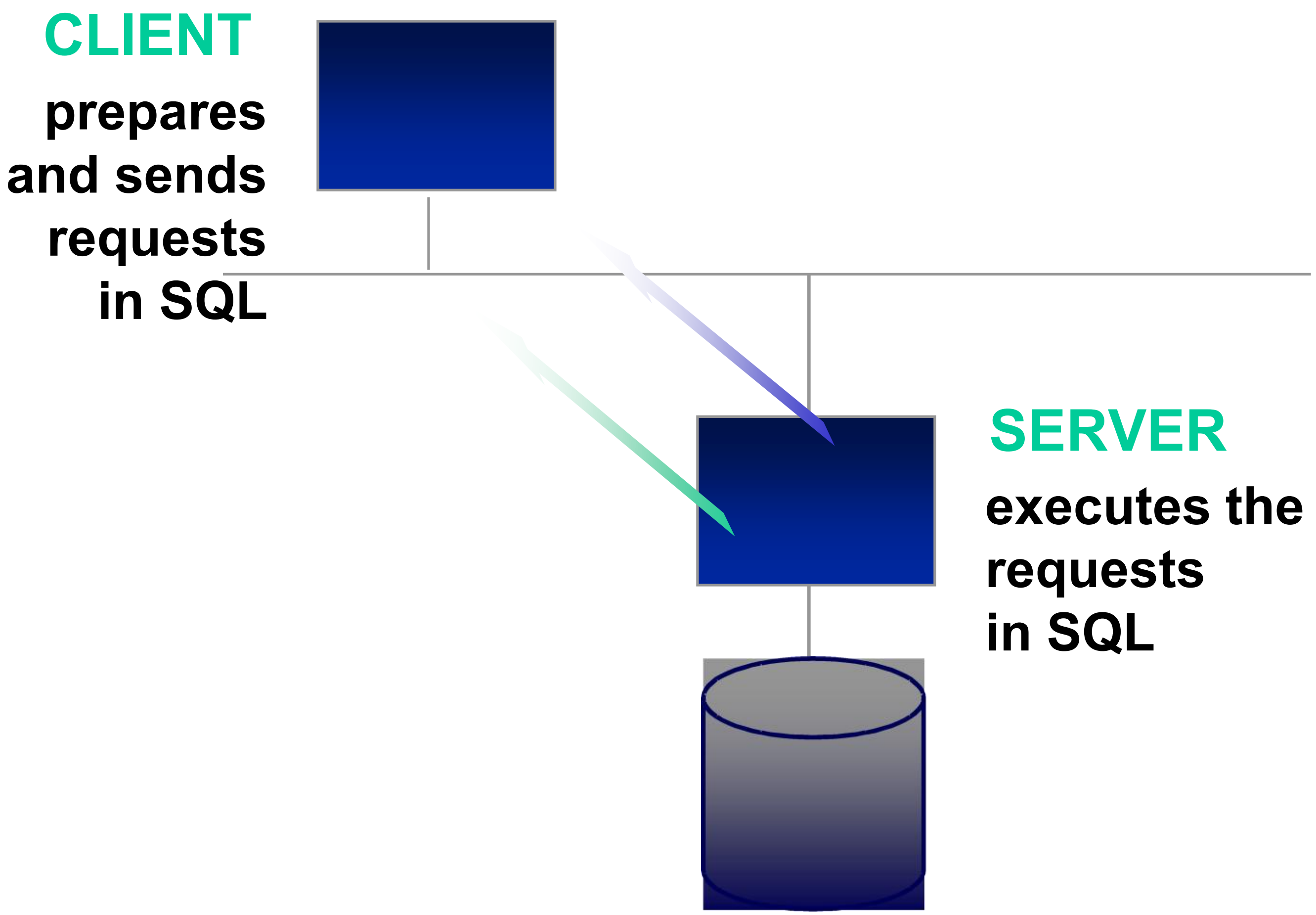
- CLIENT: formulates queries, processes results
- SERVER: runs queries
- NETWORK: transfers the activation commands (Eg: SQL commands) and the results

Client-server architecture

- The client process has an active role
  (Autonomously generates service requests)
- The server process is a passive process (performs a computation only following a request by a client)

- the client process requires a limited set of services sequentially to one or more server processes
- a server process responds to many requests from many client processes

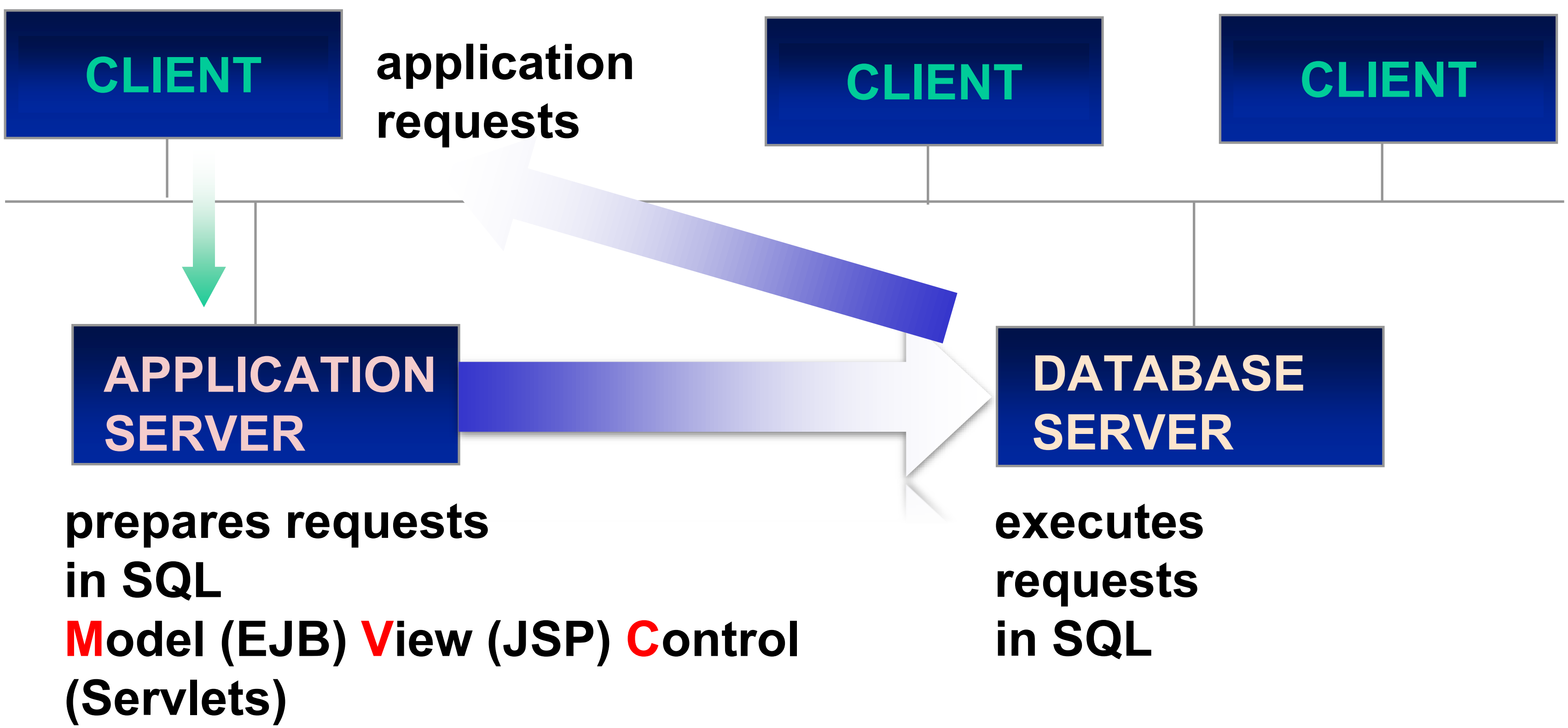
Classical Client-server architecture



Classical Client-server architecture

Typically the server manages directly or indirectly queues of requests (if indirectly, they are managed by the *dispatcher*) and runs an arbitrary number of server threads for transaction management

Client-server architecture with application server



Client-server architecture

- Data management and the allocation of client/server processes on different computers is now widespread
  - The client and server functionality are well identified
  - They lead to a convenient separation of design and management of activities
- The characteristics of the machines used for client and server can be very different
  - computers dedicated to clients must be suitable to user interaction
  - computers dedicated to the server must have a high capacity memory (for buffer management) and a high-capacity drive (for storing the data base)

Distributed Databases

Distributed Databases

- A distributed database is a system in which at least one client interacts with more than one DB server to run an application

- The distributed databases require to re-examine
  - as the user can specify queries
  - how server technology can be extended to a distributed environment

Distributed Databases: properties

- Independence from the data distribution: Users do not know where the data is stored (it extends the principles of logical and physical independence of data)
- Distributed transaction atomicity: Users should be able to write transactions that access to multiple sites just as local transactions

These two properties are generally not supported, for efficiency problems ⇒users need to know where the data is stored

Distributed Databases: properties

- Based on the natural application needs
  businesses are structurally distributed, a distributed data management allows to distribute the control and data processing where these are generated and most commonly used
- Offer more flexibility, modularity and fault tolerance
  - They can be configured with progressive addition and modification of components
  - Even if they are more susceptible to failure due to the greater structural complexity, respond to failure with a decrease in performance rather than a total failure

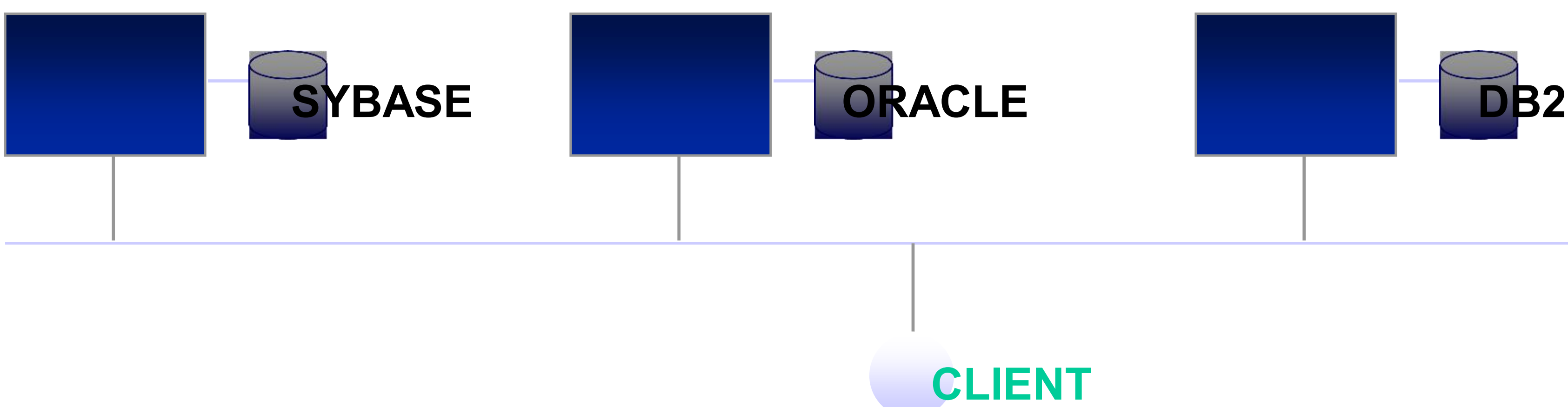
Types of distributed databases

NETWORK:

- **LAN (Local Area Network)**
- **WAN (Wide Area Network)**

DBMS:

- homogeneous system
- heterogeneous system



Typical Applications

	LAN	WAN
HOMOGENEOUS	Management and financial applications	Booking systems, financial applications
HETEROGENEOUS	Interdepartmental management applications	Integrated booking systems, Interbank systems

Distributed Databases

- In a distributed database each server has the ability to manage applications independently
- A distributed database should minimize the interactions and the need to transmit data across the network
- Data distribution should be planned (i.e. designed) in such a way that applications can operate independently on a single server

Fragmentation and data allocation

- Given a relationship *R*, its fragmentation is to determine the fragments R_i of *R* applying to *R* algebraic operations

  - **horizontal fragmentation**: The fragments R_i are groups of tuples that have the same schema of R (obtainable as selection on R)
    - horizontal fragments are generally separated

  - **vertical fragmentation**: Each fragment R_i has as schema a subschema of the schema R (obtainable as a projection on R)

All vertical fragments must include the primary key of R

Fragmentation and data allocation

- A fragmentation is correct if it is
  - *Complete*: each data item of R must be present in at least one of its fragments Ri.
  - *Restorable*: the content of R must be restorable from its fragments.

- Fragmentation is a data organization technique that allows a distribution and efficient processing of data

Fragmentation of data

Consider the relationship

EMPLOYEE (Empnum, Name, Deptnum, Salary , Taxes)

- Horizontal fragmentation:
  - EMP1 = $\sigma_{Empnum <= 3}$ EMPLOYEE
  - EMP2 = $\sigma_{Empnum > 3}$ EMPLOYEE

- Reconstruction requires union:

  EMPLOYEE = EMP1 ∪ EMP2

Fragmentation of data

EMPLOYEE (Empnum, Name, Deptnum, Salary , Taxes)

- Vertical fragmentation:
  - EMP1 = $\pi_{\text{EmpNum, Name}}$ EMPLOYEES
  - EMP2 = $\pi_{\text{EmpNum, DeptNum, Salary, Tax}}$ EMPLOYEES

- Reconstruction requires a natural join on the key value:

EMPLOYEES = EMP1 $\triangleright \triangleleft$ EMP2

Fragmentation of data

EmpNum	Name	DeptNum	Salary	Tax
1	Robert	Production	3.7	1.2
2	Greg	Administration	3.5	1.1
3	Anne	Production	5.3	2.1
4	Charles	Marketing	3.5	1.1
5	Alfred	Administration	3.7	1.2
6	Paolo	Planning	8.3	3.5
7	George	Marketing	4.2	1.4

Fragmentation of data

EmpNum	Name	DeptNum	Salary	Tax
1	Robert	Production	3.7	1.2
2	Greg	Administration	3.5	1.1
3	Anne	Production	5.3	2.1

First horizontal fragment

EmpNum	Name	DeptNum	Salary	Tax
4	Charles	Marketing	3.5	1.1
5	Alfred	Administration	3.7	1.2
6	Paolo	Planning	8.3	3.5
7	George	Marketing	4.2	1.4

Second horizontal fragment

Fragmentation of data

EmpNum	Name
1	Robert
2	Greg
3	Anne
4	Charles
5	Alfred
6	Paolo
7	George

First vertical fragment

EmpNum	DipNum	Salary	Tax
1	Production	3.7	1.2
2	Administration	3.5	1.1
3	Production	5.3	2.1
4	Marketing	3.5	1.1
5	Administration	3.7	1.2
6	Planning	8.3	3.5
7	Marketing	4.2	1.4

Second vertical fragment

Fragmentation and data allocation

- Each fragment R_i corresponds to a different physical file and is allocated on a different server
- The final relation R is a virtual entity (similar to a view) while the fragments are actually stored

The allocation scheme describes the mapping relationships from fragments to the server where they are stored. Such mapping can be:

- **non-redundant,** when each fragment or relation is stored on a single server
- **redundant,** when at least one fragment or relation is stored on more than one server

Fragmentation and data allocation

The redundancy has advantages from the point of view of performance and availability, but poses problems for the maintenance of the consistency between copies of the same fragment

Fragmentation and allocation of the fragments may be more or less visible at the application level (levels of transparency)

Levels of transparency

Three significant levels of transparency are provided: *fragmentation*, *allocation* and *language transparency*

- Consider the relation EMPLOYEE, with the horizontal fragments EMP1 and EMP2 and with the following allocation scheme:
  - EMP1@company.london.uk
  - EMP2@company.manchester1.uk
  - EMP2@company.manchester2.uk

Fragmentation transparency

At this level, the programmer needs not to worry about the fact that the database is distributed or fragmented

- example query:

```
procedure Query1(:num, OUT: name);
   SELECT Name into :name
   FROM EMPLOYEE
   WHERE EmpNum =:num;
end procedure;
```

Allocation transparency

At this level, the programmer must know the structure of the fragments, but not their location

In the case of replication, the programmer does not indicate which copy is accessed (replication transparency)

- example query:

```
procedure Query2(:num, :name);
        SELECT Name into :name FROM EMP1 WHERE EmpNum = :num;
if :empty then
        SELECT Name into :name FROM EMP2 WHERE EmpNum = :num;
end procedure;
```

Language transparency

At this level, the programmer must specify in the query both the structure of the fragments that their position

- Queries expressed at higher levels of transparency are transformed at this level by the query optimizer, that knows the fragmentation and allocation of data
- example query:

```
procedure Query3(:num, :name);
SELECT Name into :name FROM EMP1@company.london.uk  WHERE EmpNum = :num;
if :empty then
SELECT Name into :name FROM EMP2@company.manchester1.uk WHERE EmpNum =
:num;
end procedure;
```

Optimization

In the levels fragmentation and allocation choices are made by the optimizer, but it can also be guided by the programmer.

The previous query can be made more efficient in two ways:

- using the parallelism: instead of sending the two requests one after the other, these can be processed in parallel, thus saving on the overall response time
- using information about the logical properties of the fragments (but this will not develop flexible programs)

```
procedure Query4(:num, :name, :city);
case :city of
"London ":        SELECT name into :name
                  FROM EMP1 WHERE EmpNum = :num;
"Manchester":     SELECT name into :name
                  FROM EMP2 WHERE EmpNum = :num;
end procedure;
```

Classification of transactions

- **remote requests**: Read-only transactions that consist of an arbitrary number of SQL queries, addressed to a single remote DBMS (which can only be queried)
- **remote transactions**: Transactions consist of an arbitrary number of SQL commands (select, insert, delete, update), aimed at a single remote DBMS (each transaction writes on a DBMS)
- **distributed transactions**: Transactions consist of an arbitrary number of SQL commands (select, insert, delete, update), addressed to an arbitrary number of remote DBMS (each transaction may write to multiple DBMSs, but each command refers to a *single* DBMS)
- **distributed requests**: Arbitrary transaction, in which each SQL command can be addressed to any remote DBMS (requiring a distributed optimizer)

Classification of transactions

<u>Example</u> distributed transaction (Transfer between two current accounts) to the level of allocation transparency

BANK ACCOUNT(<u>Number,</u>Name, Balance)

fragmented in such a way that all accounts less than 10,000 are on the first fragment CC1 and all the upper ones are on the second fragment CC2

```
begin transaction
UPDATE CC1        SET Balance = Balance - 100,000
                  WHERE Number = 3154;
UPDATE CC2         SET Balance =  Balance + 100,000
                  WHERE Number = 14878;
commit work;
end transaction;
```

Do local ACID proprerties guarantee global ACID properties??? NO

Technology of distributed databases

The main problem is the guarantee of the *ACID* properties of transaction

- Durability does not pose new problems given that each system guarantees the persistence by itself
- Consistency poses problems in the case of integrity constraints that refer to distributed data. However, this is not allowed by current DBMSs
- There are problems for atomicity and isolation

Technology of distributed databases

In order to ensure atomicity and isolation, the DBMS software must be revised so to address the following issues:

- query optimization
- concurrency control
- recovery management

Optimization of distributed queries

Optimization is needed when a DBMS receives a distributed request

- The questioned DBMS is responsible for global optimization
  - it decides how to divide the query in subqueries, each addressed to a specific DBMS
  - it builds a "distributed" execution plan, which consists in the coordinated execution of various programs in various DBMSs and in the exchange of data between them
- The cost of running a distributed query includes the cost of transmission of data through the network

Optimization of distributed queries

A common approach is the cost-based one: it is possible to consider all the plans and choose the cheapest (as for the centralized case)

- differences wrt centralized:
  - cost of communication must be considered
  - the autonomy of the local sites must be respected
  - new distributed join methods (semijoin, bloomjoin)

The site to which the query is addressed constructs a *global* plan, which includes the suggested *local* plans for each site. If a site can improve the local plan that is suggested, it is free to do so.

Concurrency Control

- Local serializability does not guarantee serializability
- Example (the first subscript denotes the transaction, the second subscript denotes the site)

  S1: $r_{11}(X)\ w_{11}(X)\ r_{21}(X)\ w_{21}(X)$
  S2: $r_{22}(Y)\ w_{22}(Y)\ r_{12}(Y)\ w_{12}(Y)$

Note: x and y can be databases, tables, partitions, etc.

- global serializability of a set of transactions on distributed nodes of a distributed database requires the existence of a unique serial schedule S equivalent to all local schedules S_i generated at each node

Concurrency Control

- if each scheduler of a distributed database uses the 2PL and complete the commit action when all the resulting sub-schedule transactions have acquired all the resources, they are globally serializable
- if every distributed transaction acquires a single timestamp and uses the timestamp in all requests to all schedulers that use a timestamp-based concurrency control protocol, the resulting schedule is globally serializable, based on the order imposed by timestamp
  - Lamport method for assigning timestamp in distributed environment

Specific mechanisms are provided for the resolution of deadlocks in a distributed environment (besides the use of *timeout*)

Atomicity: faults in distributed systems

- **Node failures:** they can occur at any node of the system
- **Messages Losses** leave the execution of a protocol in a state of uncertainty
  - each protocol message (msg) is followed by a confirmation message (ack)
  - the loss of one of two messages may leave the sender uncertain about the the receipt of the message
- **Network partitioning** is a failure of the network communication link that divides it into two subnets the network, without the possibility of communication
  - a transaction can be active simultaneously in more than one subnet

Two - Phases Commit Protocol (2PC)

- Commit protocols allow a transaction to reach the correct decision on commit / abort for all the nodes participating in a transaction
- the two-phase commit protocol can be thought of as a wedding, as the decision of the two parts is received and registered by a third party, that is, the coordinator
  - servers (representing the participants at the wedding) are called resource managers (RM)
  - the coordinator (celebrant) is a process, said transaction manager (TM)

Two - Phases Commit Protocol (2PC)

- The Protocol consists of a quick exchange of messages between TM and RM and in record entries in their log
- TM can use
  - broadcast mechanisms, to transmit the same message to many nodes, then collecting the answers that come from the various nodes
  - serial communication with one of the RM at a time
- record of TM (in its log)
  - the record **prepare** contains the identity of all the RM processes (Id of node and process)
  - the records **global commit** or **global abort**, which describe the final decision (when the TM writes that record in his log it reaches the final decision)
  - the record **complete**, written at the end of the two-phase commit protocol

Two - Phases Commit Protocol (2PC)

- records of RM (in its log)
  - the record **ready** indicates the irrevocable availability to participate in the protocol
    - can only be written when the RM is restorable, that is, when it owns all the locks on the resources that need to be written
    - the record must also contain the id of the TM
  - We have also the classic records of the case centralized log (I, D, U, Checkpoint, Dump).
- A RM can, anytime, abort a sub-transaction, dismissing its effects, without participating in the two-phase commit protocol

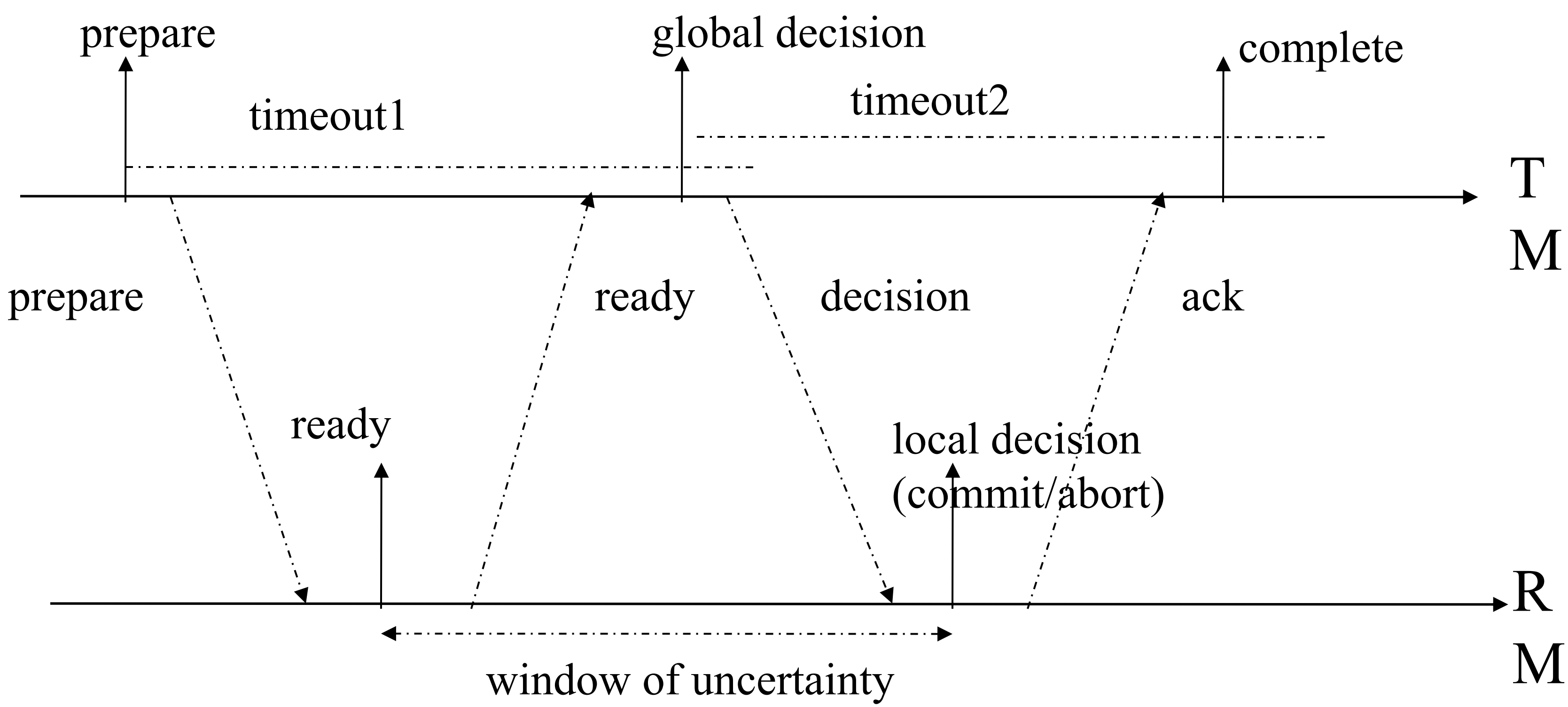
2PC - Phase 1

- The TM writes the record **prepare** in its log, sends a message **prepare** to all the RMs and sets a timeout that indicates the maximum time allotted for completion of the first phase
- The restorable RMs write on their log a record **ready** and send a message **ready** to the TM, indicating a willingness to participate in the commit
- Unrecoverable RMs send a message **not-ready** and terminate the Protocol
- TM collects the response messages by the RMs:
  – if it receives positive feedback from all RMs, it writes a record **global commit** on its log
  – if it receives one or more negative feedbacks, or if the time-out expires before the TM has received all the answers, it writes a record **global abort** on its log

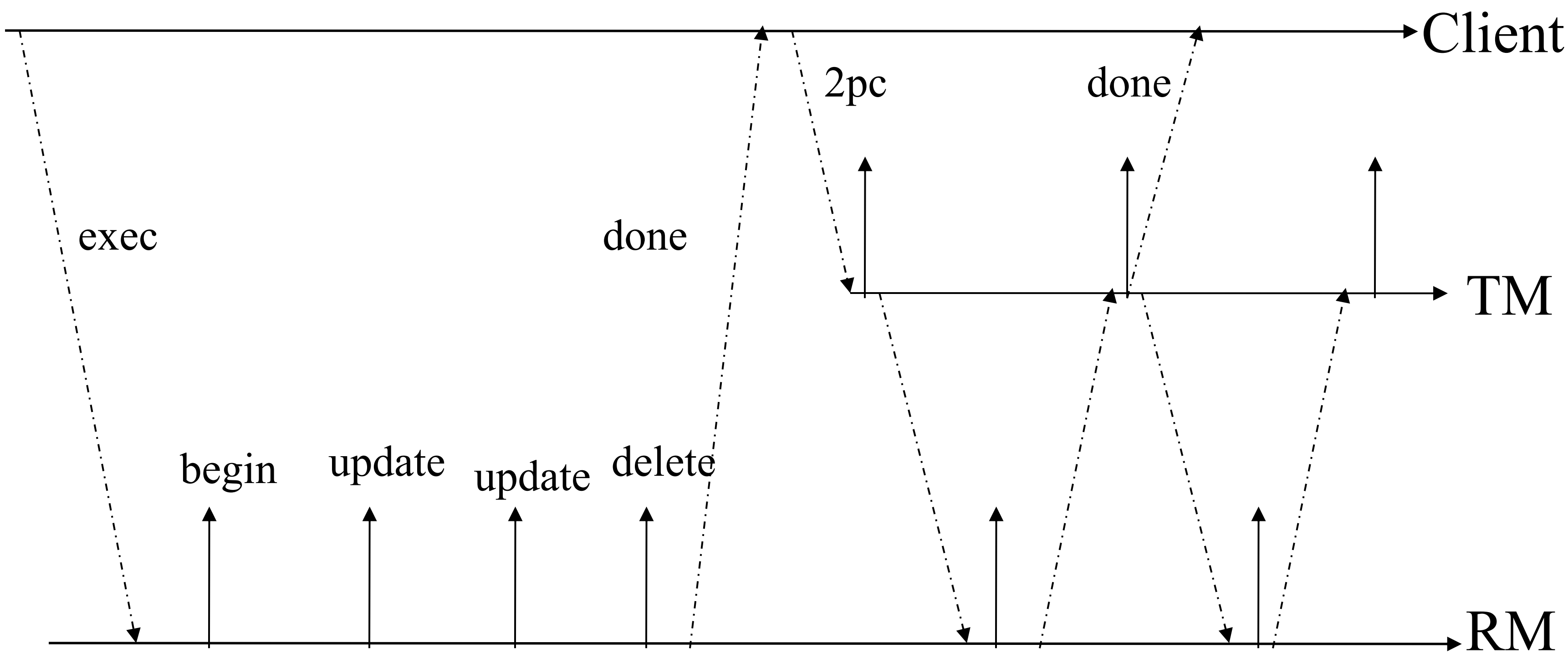
2PC - Phase 2

- TM transmits its global decision (**commit** or **abort**) to the RMs and sets a second timeout
- RMs that are ready to receive the decision message, write the records **commit** or **abort** in their log and send a confirmation (**ack**) to the TM
- then implement the commit or abort, writing the pages in the database such as in the centralized case
- TM collects all **acks** by the RMs involved in the second phase
- if the timeout expires it sets another timeout and repeats the transmission to all the RMs for which it has not received an **ack**
- when the TM receives all **acks,** it writes the record **complete** in its log

Two - Phases Commit Protocol (2PC)



2PC: the role of the client



Two - Phases Commit Protocol (2PC)

- A RM in the state **ready** loses its autonomy and waits the decision of the TM
- A failure of the TM leaves the RM in a state of uncertainty
- Resources acquired using lock are blocked
- The window between the writing on the RM log of the record **ready** and of the record **commit** or **abort** is called *window of uncertainty*
- The protocol must maintain that interval to minimum
- TM or RMs protocols perform recovery after failures
- The final state restored depends on the global decision of the TM

Recovery of the participants

- It depends on the last record written to the log
  - if it is an action or a record **abort**, actions are undone (*undo*)
  - if it is a record **commit**, the actions are re-executed (*redo*)
- In both cases the failure occurred before the start of the commit protocol
- If the last record written in the log is a record **ready**, the fault has occurred during the 2PC, and the participant is in doubt about the outcome of the transaction

Recovery of the participants

- During the recovery all the identifiers of in-doubt transactions are collected in a set (ready set)
- for each of them the final outcome of the transaction must be requested to the TM
- this can happen either as a result of a direct request (remote recovery) by the RM or as a repetition of the second phase of the protocol

Recovery of the TM

- When the last record in the log is a record **prepare**, the failure of the TM may have left some RM blocked
- There are two possible restore options:
  - write **global abort** on the log, and proceed with the second phase of the protocol
  - repeat the first phase, trying to reach a global commit
- when the last record in the log has a final decision, some RM may have been properly informed of the decision and others may have been left blocked, the TM must then repeat the second phase

Loss of network messages and partitioning

- The loss of the message **prepare** is not distinguishable by the TM from the loss of a message **ready**
- In both cases the timeout of the first phase expires and the final decision abort is taken
- Similarly, the loss of a decision message is indistinguishable from the loss of a message **ack**
- In both cases, the timeout expires and the second phase is repeated
- A network partitioning does not cause more problems, because the transaction will be completed successfully only if the TM and all the RMs belong to the same partition

Data Bases and parallelism

Data Bases and parallelism

- Received attention in the nineties with the spread of multiprocessor architectures, after the failure of architectures dedicated to databases (*database machines*) of the eighties
- The parallelism is possible with multiprocessor architectures with shared memory or without shared memory, although with different technical solutions
- With a **parallel database** it is possible to improve the performances through the parallel implementation of various operations
- Even if the data can be stored in a distributed manner, the distribution is guided only by considerations related to the performance.

Parallelism

- The reason for the success of the parallelism in the databases is that the data management operations are quite repetitive in nature, and may be performed efficiently in parallel
  - A full scan of a large data base may be performed by *n* scans, each on a portion of the database
  - If the database is stored in *k* disks, managed by *k* different processors, the response time will be about *1 / k* the time required for a sequential scan

Inter-query Parallelism

- Parallelism is called *inter-query* when several queries are executed in parallel
- The load imposed on the DBMS is typically characterized by simple and frequent transactions (up to thousands of transactions per second)
- The parallelism is introduced by incrementing the number of servers and allocate the optimum number of requests to each server
- In many cases, queries are redirected to a server by a process called *dispatcher*
- Useful for OLTP systems

Intra-query parallelism

- Parallelism is called *intra-query* when different parts of the same query are executed in parallel
- The load of the DBMS is typically characterized by a few very complex queries, which are decomposed into various partial subqueries, executed in parallel
- In general, the queries are executed one after the other, using all the multi-processor system for each query
- Useful for OLAP systems

Parallelism and data fragmentation

- The parallelism is generally associated with fragmentation of data: the fragments are distributed among many processors and allocated in different storage devices

consider the tables:

ACCOUNT(AccNum, Name, Balance)

TRANSACTION(AccNum, Date, SerialNumber , TransactionType, Amount)

Fragmented according to predefined intervals of the account number

Inter-query parallelism and data fragmentation

- a typical example of OLTP query with inter-query parallelism is:

```
procedure Query5(:acc-num, :total);
select Balance into :total
from Account
where AccNum = :acc-num;
end procedure
```

A query of this type is directed towards the specific fragment depending on the selection predicate

Intra-query parallelism and data fragmentation

- a typical example of OLAP query with intra-query parallelism is:

```
procedure Query6();
    select AccNum, sum(Amount)
    from Account join Transaction on Account.AccNum =
        Transaction.AccNum
    where Date >= 1.1.1998 and Date < 1.1.1999
    group by AccNum
    having sum(Amount) > 100000
end procedure;
```

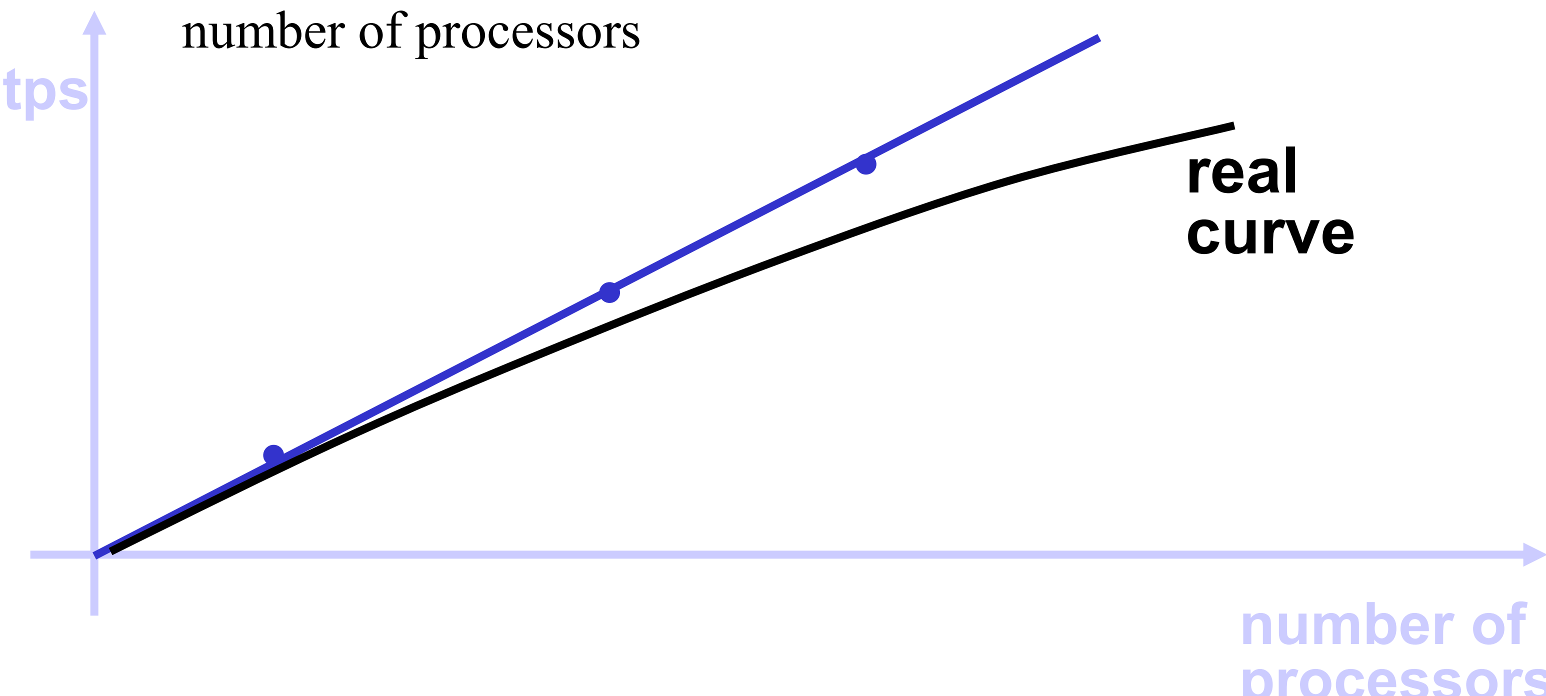
A query of this type is carried out on the various fragments in parallel

Distributed join

- Consider the join of pairs of fragments that correspond to the same range of account numbers
- The joins between the various fragments which match may be carried out in parallel
- Distributed joins are essential for **intra-query parallelism**
- The parallel execution on *n* join fragments of size *1 / n* is obviously preferable to the execution of a single join that involves the whole tables (such as the hash join)
- In general, when the initial fragmentation does not allow the execution of the distributed join present in the query, the data is redistributed dynamically to allow distributed join

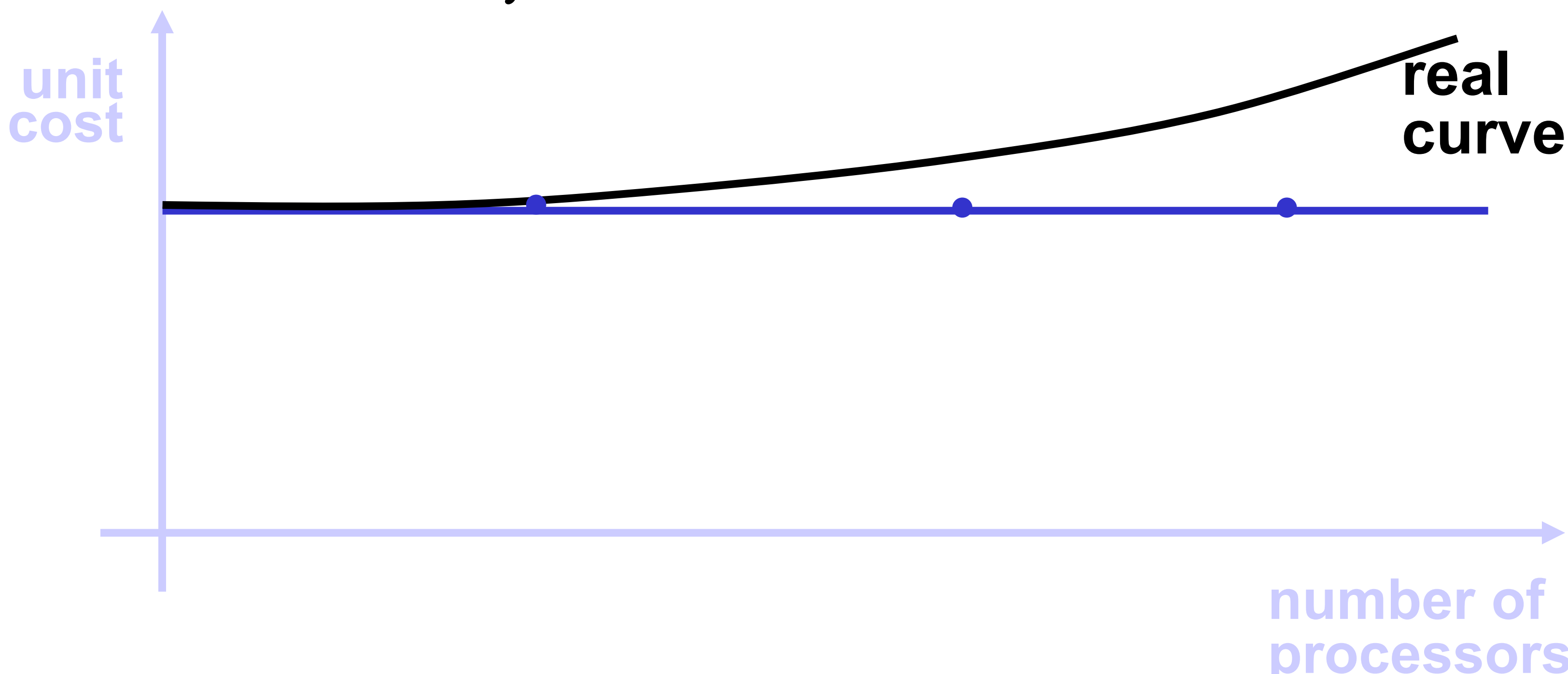
Speed-up and scale-up

- The curve of **speed-up** characterizes only the **inter-query** parallelism and measures the improvement of services, measured in tps (transactions per second), compared to an increase in the number of processors
  - in the ideal situation, the services grow approximately linearly in the number of processors



Speed-up and scale-up

- the curve of **scale-up** characterizes both **inter-query** and **intra-query** parallelism and measures the average cost of a single transaction with respect to the increase in the number of processors
  - in the ideal situation, the average cost remains approximately constant in the number of processors
  - it is said that the system "scales" well



Replicated Databases

Data Replication

- Data replication is an essential service for the creation of many distributed applications
- **Synchronous replication:** all copies of a table (or fragment) changed need to be updated before committing the transaction that modified it
  - the replication of the data is made transparent to the user
- **Asynchronous replication:** copies of a modified relation are only periodically updated
  - several copies of the data may be related to different states of the database, therefore contain different values
  - users should be aware of replication and data distribution

Systems based on data replication

- Replication is guaranteed by tools called data replicators, whose main function is to maintain the consistency between copies
- These tools operate in a transparent manner with respect to the applications running on the DBMS server
- In general, there is a master copy, and various secondary copies: updates are propagated asynchronously (without 2PC)
- The propagation is incremental when it is based on variations in the data
- The use of replication makes the system less sensitive to failures, because if the primary copy is not available, it is always possible to use one of the secondary copies

Synchronous Replication

- **Voting:** the transaction must write the majority of copies to modify an object and must read enough copies to be sure to see at least one of the latest copy
  - es. 10 copies, 7 updated, 4 read
  - each copy has a version number
  - generally difficult to apply because the readings are frequent
- **Read-one write-all:** writes are slower and the readings are faster, than the voting
  - is the most common approach for synchronous replication
- Depending on the technique chosen, the lock to acquire on the data are different

Synchronous Replication

- The cost of the synchronous replication is high
- Before a transaction may make the commit, it is necessary to obtain the lock on all the copies modified
  - The master sends lock requests to remote sites, and while waiting for the response, holds locks on other data
  - If a node or a link fails, the transaction may not be able to commit until it is restored
  - Even in the absence of faults, commit must follow an expensive protocol with many messages
- This is why asynchronous replication is preferred

Asynchronous Replication

- It allows the transaction performing the modification to carry out the commit before all copies have been modified (and in the readings, still look at a single copy)
- Users need to be aware from which copy they are reading, considering that some copies may be out of sync for short periods of time
- Two approaches: with master and peer-to-peer
- The difference lies in how many copies are "updatable" (master copies)

Replication peer-to-peer

- **Symmetric replication:** the changes can be performed on any one of the copies
- Changes to a copy must be propagated to other copies somehow
- It is possible to have conflicts, because two copies of the same information are handled concurrently without concurrency control
- There are techniques capable of detecting inconsistencies, the correction of which depends on the application
- This technique is typically used especially when it is not possible to have conflicts:
  - every master copy is relative to a single fragment
  - the update grants are reserved to one master at a time

Replication with master site

- Exactly one copy of a table (or fragment) is designated to be the master copy
- Replicas in other sites can not be updated directly
- The primary copy is accessible to all
- The other sites store secondary copies of the table (or fragment)

The main problem is how to propagate the changes on the master copy to other copies. This is done in two steps:

- first the changes made by transactions that have committed are captured (capture) and then
- these changes are applied (apply)

Replication with master site - step capture

- **Log-based capture:** log (kept for recovery management) is used to generate a table of changes (Change Data Table CDT) if this is done when the log is written to disk, it is necessary to remove in some way the changes made by transactions that have aborted
- **Procedural capture:** a procedure that is automatically invoked (trigger) that performs the capture, taking a snapshot of the primary copy

Log-based capture is better (cheaper and faster) but relies on the proprietary log details

Replication with master site - step apply

- The apply processes on the secondary site periodically receive a snapshot or changes contained in the CDT from the master site, and update the copy
- Replicas can be seen as views of the changed table. In this respect, the replication consists in incrementally updating the “materialized view”

- Capture based on log + continuous “apply”: the purpose is to minimize the delays in the propagation of changes
- Procedural capture + application-driven apply: the most flexible way to manage changes

Amazon Dynamo DB (p-to-p asynchronous)

We analyze the design and the implementation of Dynamo.

Amazon runs a world-wide e-commerce platform

It serves 300-310 millions customers

At peak times it uses >10000 servers located in many data centers around the world.

They have requirements of performance, reliability and efficiency that needs a fully scalable platform.

77

Motivation of Dynamo

- There are many Amazon services that only need primary-key access to a data store
  - To provide best-seller lists
  - Shopping carts
  - Customer preferences
  - Session management
  - Sales rank and product catalogs
- Using relational databases would lead to inefficiencies and limit scale availability

78

Scalability is application-dependent

Lesson 1: the reliability and scalability of a system is dependent on how its application state is managed.

Amazon uses a highly decentralized, loosely coupled service oriented architecture composed of hundred of services.

They need that the storage is always available.

Shopping carts always

Customers should be able to view and add items to their shopping carts even if:

- Disk are failing, or
- A data center are being destroyed by a tornados or a kraken.

80

Failures Happens

When you deal with an infrastructure composed by million of component servers and network components crashes are normal events

81

High Availability by contract

Service Level Agreement (SLA) is the guarantee that an application can deliver its functionality in a bounded time.

An example of SLA is to guarantee that the API provide a response within 300ms for 99.9% of its requests for a peak of 500 concurrent users (CCU).

Normally SLA is described using average, median and expected variance.

82

Dynamo DB

It uses a synthesis of well known techniques to achieve scalability and availability.

1. Data is partitioned and replicated using consistent hashing [Karger et al. 1997].
2. Consistency is facilitated by version clock and object versioning [Lamport 1978]
3. Consistency among replicas is maintained by a decentralized replica synchronization protocol (E-CRDT).
4. Gossip protocol is used for membership and failure detection.

Key and Value encoding

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes.

It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

84

Dynamo Architectural Choice 1/2

We focus on the core of distributed systems techniques used

Problem	**Technique**	**Advantage**
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available

85

Dynamo Architectural Choice 2/2

We focus on the core of distributed systems techniques used

Problem	**Technique**	**Advantage**
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

86

Partitioning: Consistent Hashing

Dynamo musts scale incrementally.

This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system.

Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts.

The output range of a hash function is treated as a fixed circular space or ring

87

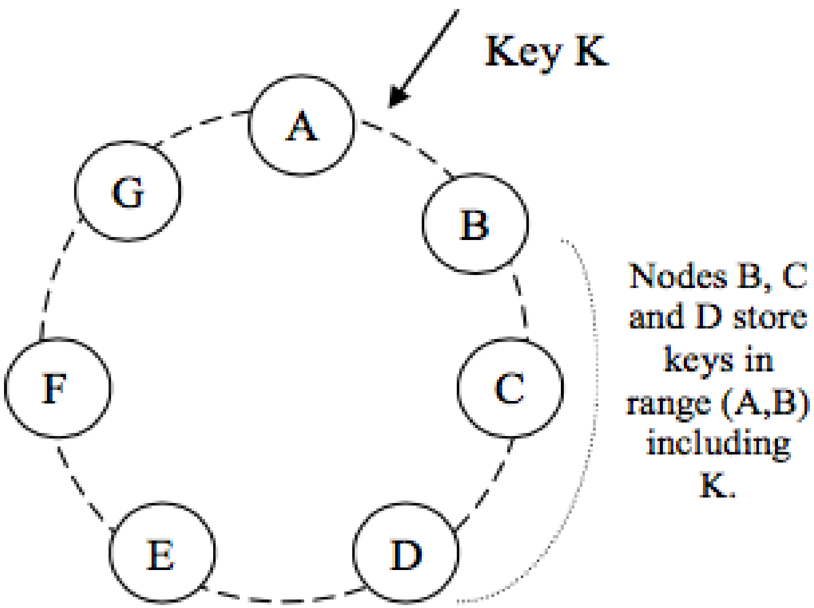
Partitioning: Consistent Hashing

Each node in the system is assigned a random value within this space which represents its "position" on the ring.

Each data item is assigned to a node by:

1. hashing the data item's key to yield its position on the ring,
2. and then walking the ring clockwise to find the first node with a position larger than the item's position.

Partitioning: Consistent Hashing



Each node becomes responsible for the region in the ring between it and its predecessor node on the ring

The main advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

89

Consistent Hashing: Idea

Consistent hashing is a technique that lets you smoothly handle these problems:

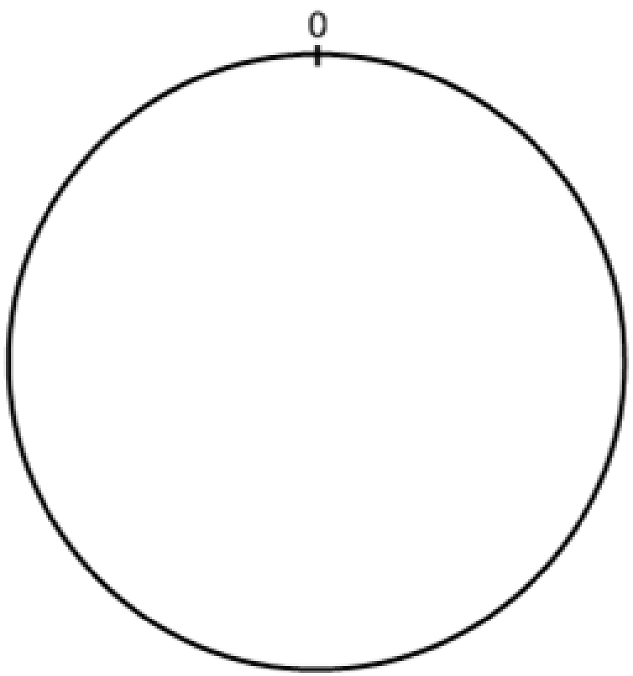
1. Given a resource key and a list of servers, how do you find a primary, second, tertiary (and on down the line) server for the resource?
2. If you have different size servers, how do you assign each of them an amount of work that corresponds to their capacity?

Consistent Hashing: Idea

Consistent hashing is a technique that lets you smoothly handle these problems:

3. How do you smoothly add capacity to the system without downtime?
4. Specifically, this means solving two problems:
    - How do you avoid dumping 1/N of the total load on a new server as soon as you turn it on?
    - How do you avoid rehashing more existing keys than necessary?

Consistent Hashing: How To



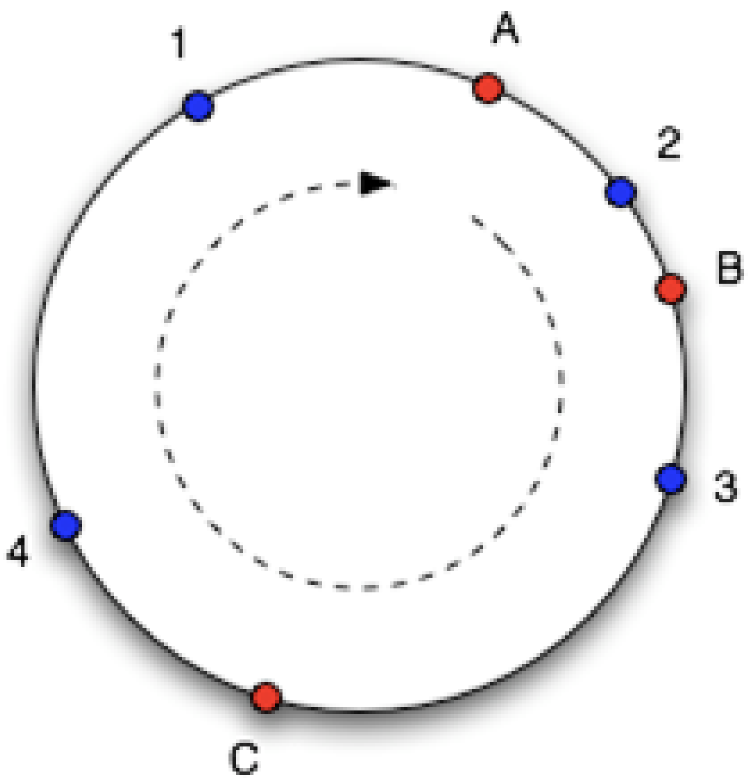
Imagine a 128-bit space.

visualize it as a ring, or a clock face

Now imagine hashing resources into points on the circle

92

Consistent Hashing: How To



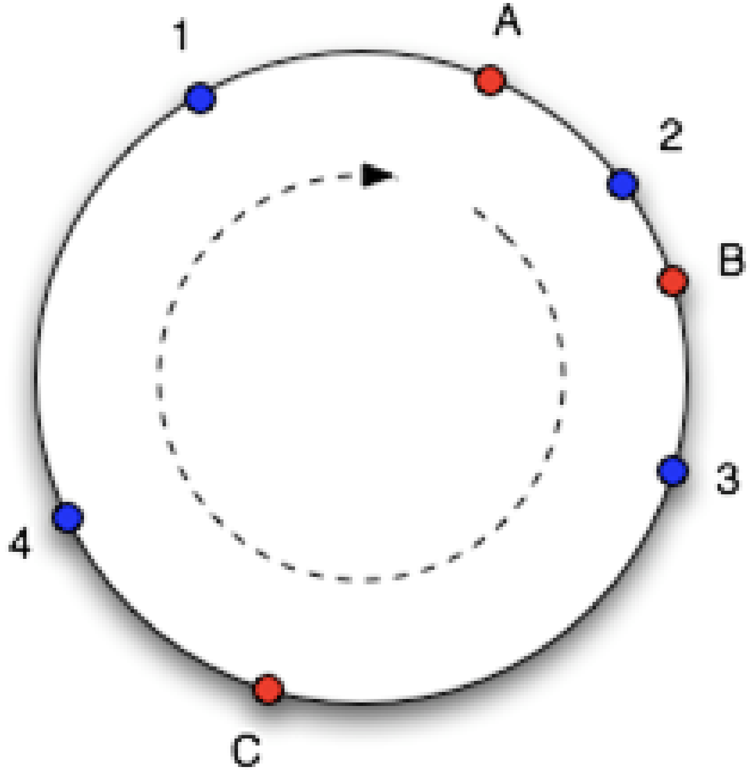
They could be URLs, GUIDs, integer IDs, or any arbitrary sequence of bytes.

Just run them through a good hash function (eg, MD5) and shave off everything but 16 bytes.

We have four key-values: 1, 2, 3, 4.

93

Consistent Hashing: How To



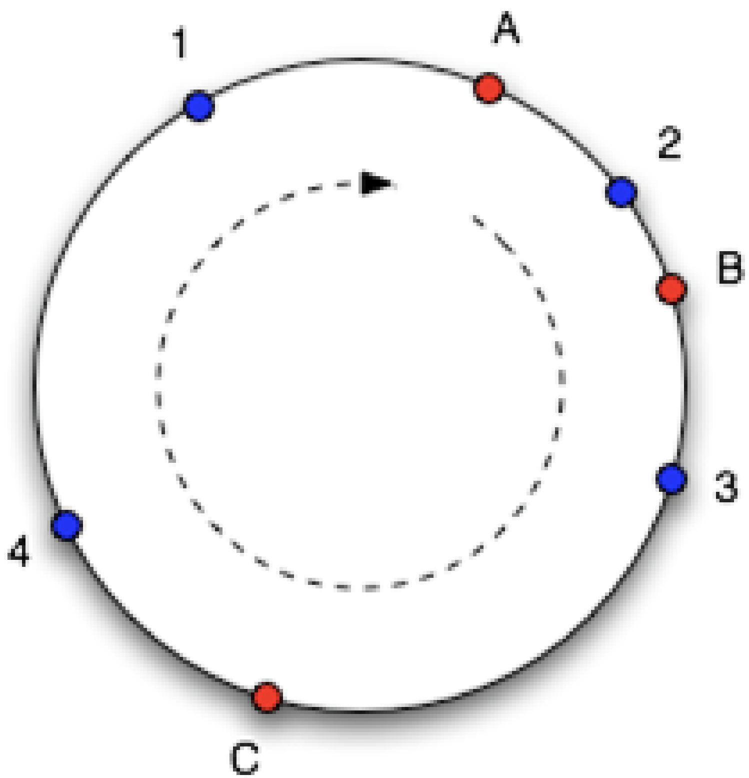
Finally, imagine our servers.

- A,
- B, and
- C

We put our servers in the same ring.

We solved the problem of which server should use Resource 2

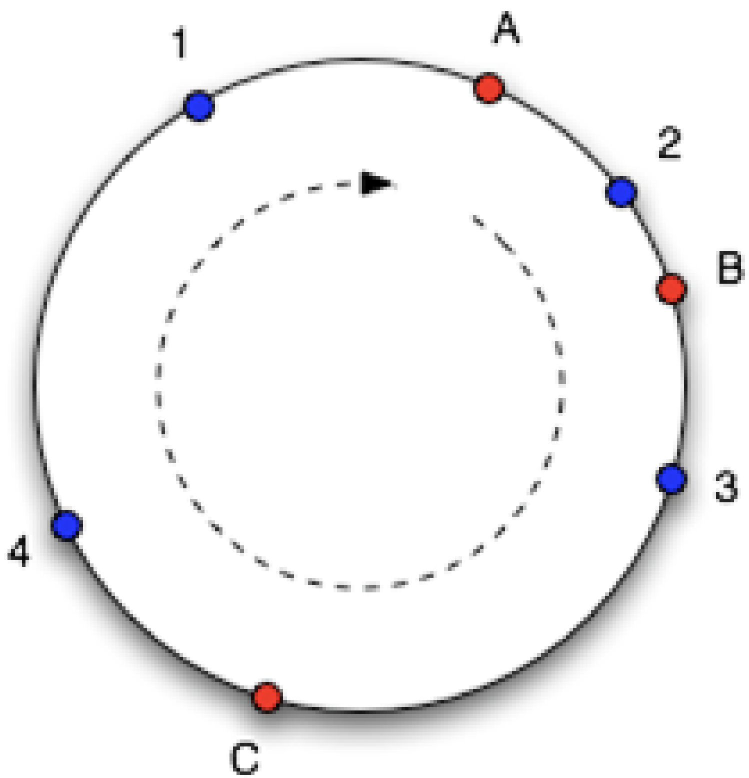
Consistent Hashing: How To



- We start where resource 2 is and, head clockwise on the ring until we hit a server.
- If that server is down, we go to the next one, and so on and so forth

95

Consistent Hashing: How To

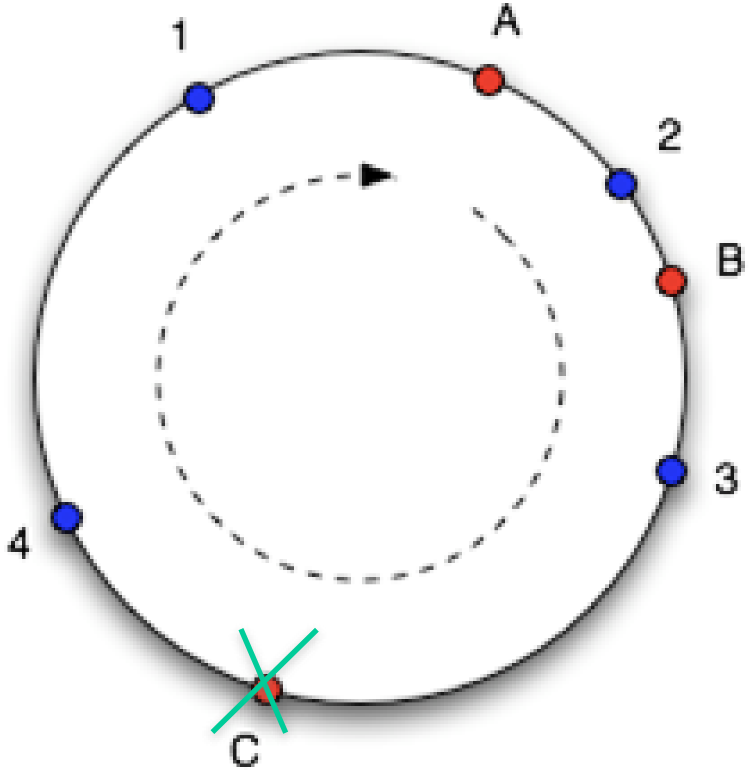


Key-value 4 and 1 belong to the server A

Key-value 2 to the server B

Key-value 3 to the server C

Consistent Hashing: Del Server

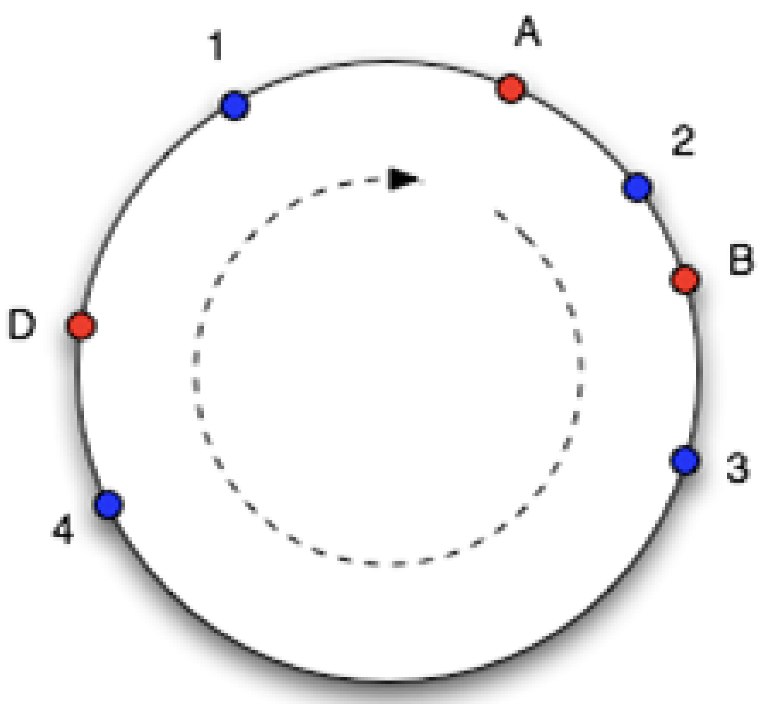


If the server C is removed

Key-value 3 now belongs to the server A

All the other key-values mapping are unchanged

Consistent Hashing: Add Server



If server D is added in the position marked

What are the object that will belongs to D?

Data Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts.

Each data item is replicated at N hosts, where N is a parameter configured “per-instance”.

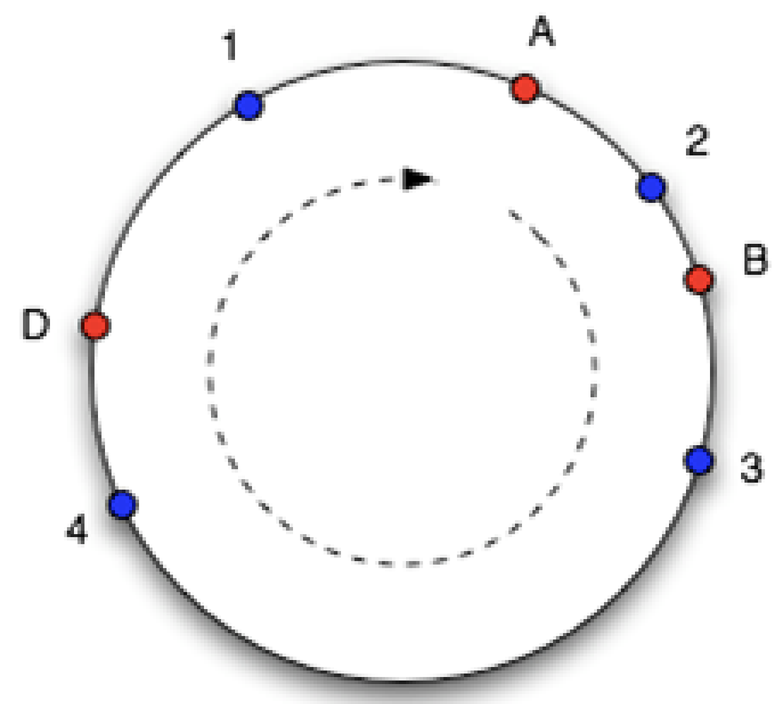
Each key ***k*** is assigned to a coordinator node (described above).

The coordinator is in charge of the replication of the data items that fall within its range (ring).

Data Replication

The coordinator locally stores each key within its range,

And in addition, it replicates these keys at the N-1 clockwise successor nodes in the ring.



Data Replication

The list of nodes that is responsible for storing a particular key is called the *preference list*

The system is designed so that every node in the system can determine which nodes should be in this list for any particular key.

To account for node failures, preference list contains more than N nodes.

To avoid that with “virtual nodes” a key *k* is owned by less than N physical nodes, the preference list skips some nodes.

101

High Availability for writes

With eventual consistency writes are propagated asynchronously.

A put() may return to its caller before the update has been applied at all the replicas.

In this scenario a subsequent get() operation may return an object that does not have the latest updates.

102

High Availability for writes: Example

We can see this event with shopping carts.

The "*Add to Cart*" operation can never be forgotten or rejected.

When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

Question!

103

High Availability for writes

Dynamo treats the result of each modification as a new and immutable version of the data.

It allows for multiple versions of an object to be present in the system at the same time.

Most of the times, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation).

104

Heterogeneity

Heterogeneity

- Heterogeneity might be related to any level:
  - the software that creates and manipulates data in the various sites differs
  - the format and structure of data on the various sites is different
- the heterogeneity, moreover, can manifest at all levels of the database system:
  - applications written in different languages
  - different query languages
  - different data models
  - different DBMS
  - different file systems
  - ...

Cooperation between existing systems

- The great development of networks in recent years has led to opportunities for integration of existing information systems
- The reasons may be different:
  - integration of components developed separately
  - cooperation or merge of companies or entities
- Integration of databases is quite difficult: the integration objectives and too ambitious standardization are doomed to fail
- The ideal model of a fully integrated database, which can be queried in a transparent and efficient manner is impossible to develop and manage

Cooperation between existing systems

- **Cooperation:** ability of applications of a system to make use of application services provided by other systems, also managed by different entities
- Different types of cooperation: cooperation centered on processes and cooperation centered on the data
  - **Cooperation centered on processes**: Systems will provide services to each other without making remote data explicitly visible
  - **Cooperation centered on the data:** The data of a system are visible to other systems
- In general, the systems are distributed, heterogeneous, autonomous (managed by different subjects)

Multi-database systems

- Each of the participants databases continues to be used by its members
- Systems can also be accessed by a module, called **mediator**, which only shows the base portion of data that must be exported
- This portion is made available to a global manager, which realizes the integration
- In general, the data can not be modified by means of the mediator, because each system is autonomous
- Properties:
  - integrated view to users
  - high level of transparency
  - high level of currency, because the data is accessed directly

Multi-database systems

- **Portability** is the ability to “port” programs from one environment to another
  – It is typically a property related to the compilation time
  – It is facilitated by a standard level of language (eg. SQL)
- **Interoperability** is the ability to interact shown by heterogeneous systems
  – It is typically a property relating to the execution time
  – It is facilitated by standards in terms of data access protocols (eg. ODBC)

No global schema

- The levels of abstraction in DBMSs typically refer to a single database scenario:
  - One single (and ***global***) database
- The real world, in fact, is very often represented by **multiple** databases
  - autonomous
  - semantically heterogeneous
  - manipulable through a more sophisticated language (**multi-base**)

Homogeneity

Federated Architecture

(Hambiger & Mcleod, Years 1980)

- Each database should be **independent**
- In general, there will be no global schema
  - the global integration is against the autonomy
- Databases used together form a federation of autonomous databases
- For each database in the federation three schemes must be specified:
  - ES: export schema
  - IS: import schema
  - PS: private schema: for all private data, including those of ES and IS
- There has to be some dictionary of the federation (FD)

Federated Architecture

(Hambiger & Mcleod, Years 1980)

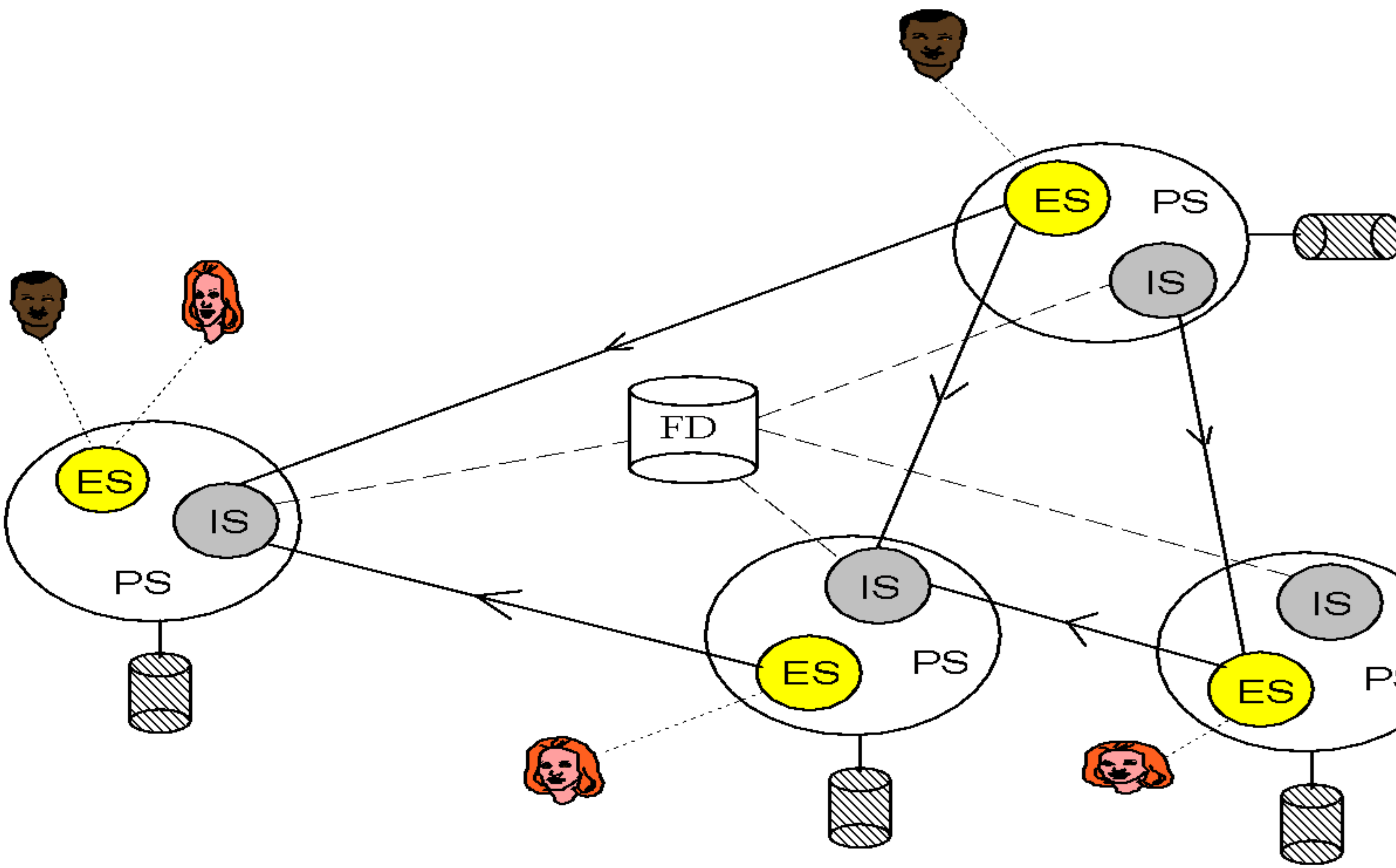


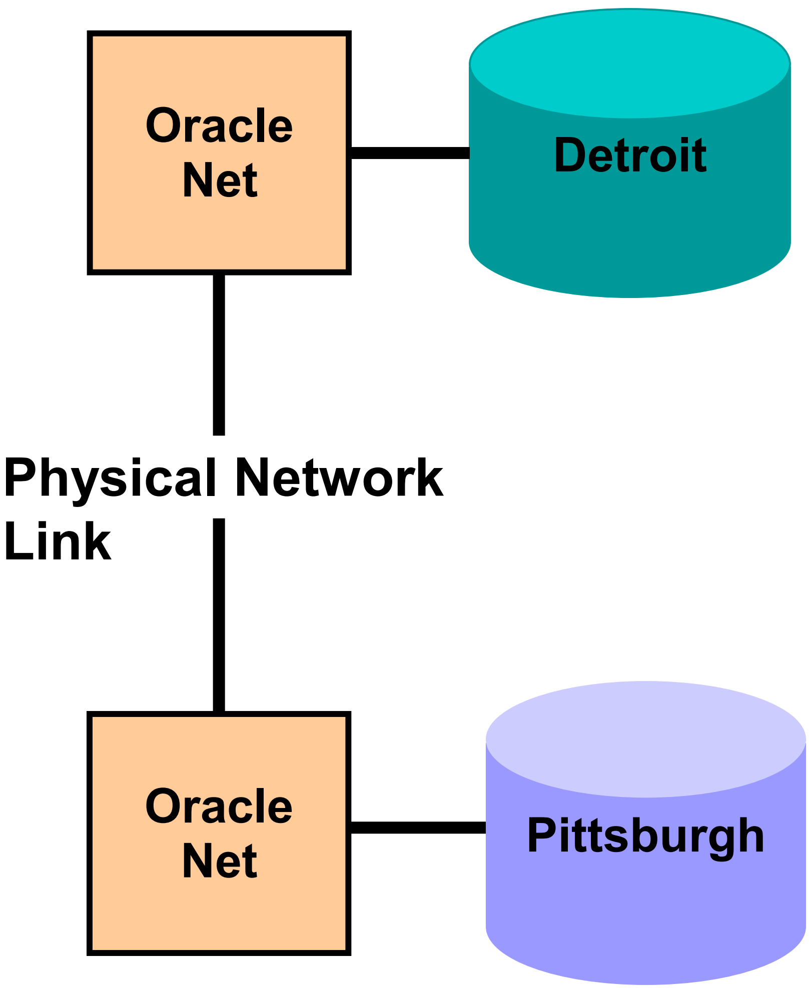
Fig 3. Federated databases architecture

ORACLE: Federated, homogeneous architecture

- Sending messages or data between sites requires network configuration on both sites.
- It is necessary to configure the following:
  - Network connectivity (for example, `tNSNames.ora`)
  - Database links

```
CREATE DATABASE LINK <remote_global_name>
CONNECT TO <user> IDENTIFIED BY <pwd>
USING '<Connect_string_for_remote_db>';
```

Server-Server Connectivity




```
CREATE DATABASE LINK
fin_flowers CONNECT
TO hr IDENTIFIED BY hr
USING 'FINANCE';

SELECT * FROM
employees@fin_flowers;
```

'FINANCE' DB is in Pittsburgh

Types of Database Links

Database links have three dimensions:

- Global, public, and private
- Fixed user, connected user, current user
- Shared or non-shared

```
CREATE DATABASE LINK yellocom_ceci
CONNECT TO yellowcom IDENTIFIED BY yellowcom
USING '192.168.1.3:1521/orcl';
SELECT * FROM CUSTOMERS@yellocom_ceci;


SQL> descr customers;
 Name                                      Null?    Type
 ----------------------------------------- -------- --------------

 NAME                                               VARCHAR2(20)
 SURNAME                                            VARCHAR2(30)
 CF                                        NOT NULL VARCHAR2(16)
 CITY                                               VARCHAR2(30)
 ADDRESS                                            VARCHAR2(50)
 BIRTH                                              DATE
 EMAIL                                              VARCHAR2(50)
 ID                                        NOT NULL NUMBER(38)

SQL>
```

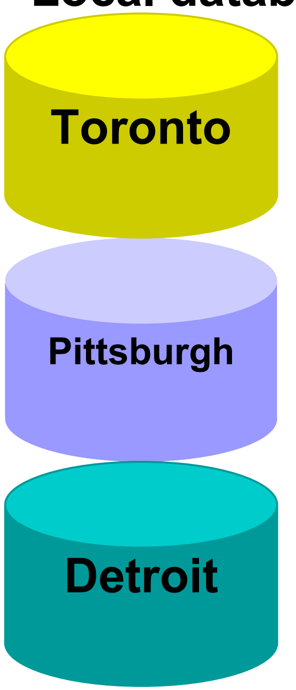
Distributed Query

- The user is connected to a local database and issues a query involving tables from at least two databases.
- The query is always executed from a local database.
- The local database decomposes the query into subqueries to be sent to each remote database.
- The local database retrieves data from remote databases and performs any necessary post-processing.

Executing a Distributed Query

```
SELECT e.last_name,j.job_title
FROM employees@pittsburgh e, jobs@detroit j,
     job_history h
WHERE e.job_id = j.job_id AND e.employee_id =
h.employee_id AND h.end_date >= SYSDATE-30;
```

Local database




```
SELECT employee_id, end_date
FROM job_history;
```

```
SELECT last_name, job_id
FROM employees;
```

```
SELECT job_id, job_title FROM jobs;
```

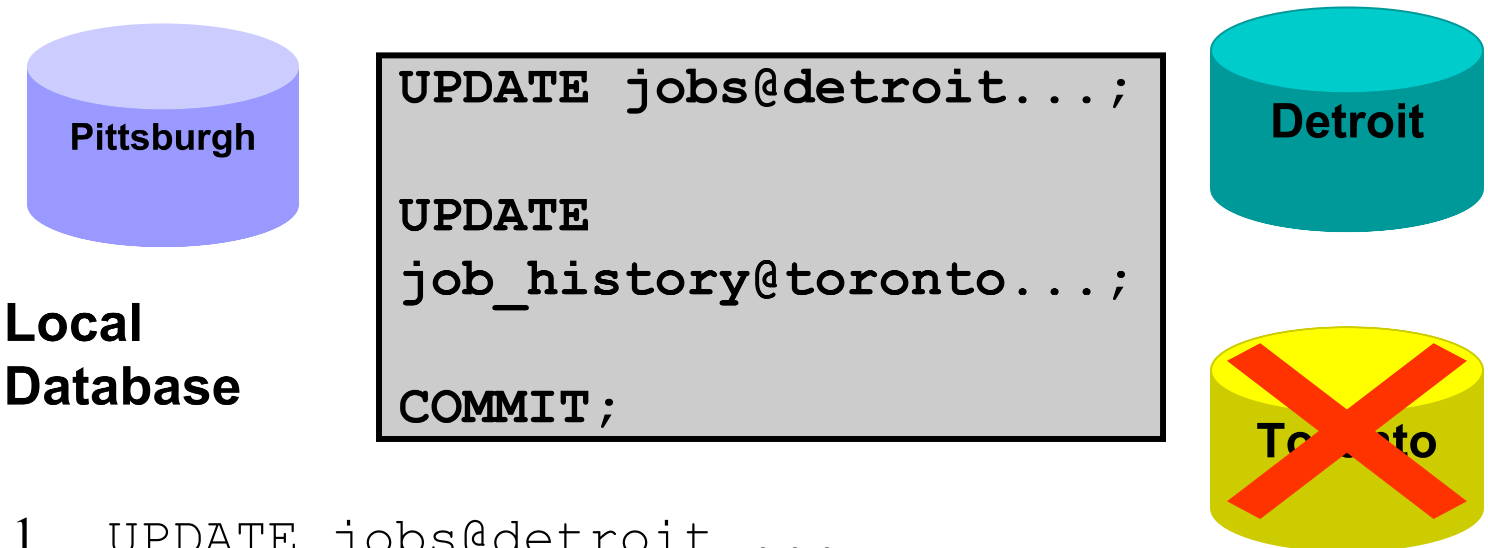
Distributed and Remote Transactions

```
INSERT INTO jobs@detroit VALUES
('CEO','Chief Executive Officer',18000,34000);

UPDATE employees@pittsburgh
SET job_id = 'CEO'
WHERE last_name = 'Hart';

COMMIT;
```

In-Doubt Transactions



1. `UPDATE jobs@detroit ...`
2. `UPDATE job_history@toronto ...`
3. Toronto Database Becomes unavailable.
4. Commit fails

References

P. Atzeni, S. Ceri, P. Fraternali, S. & R. Paraboschi Torlone
Databases: Architectures and lines of evolution. Second edition.
McGraw-Hill Books Italy, 2007
Chapter: 6