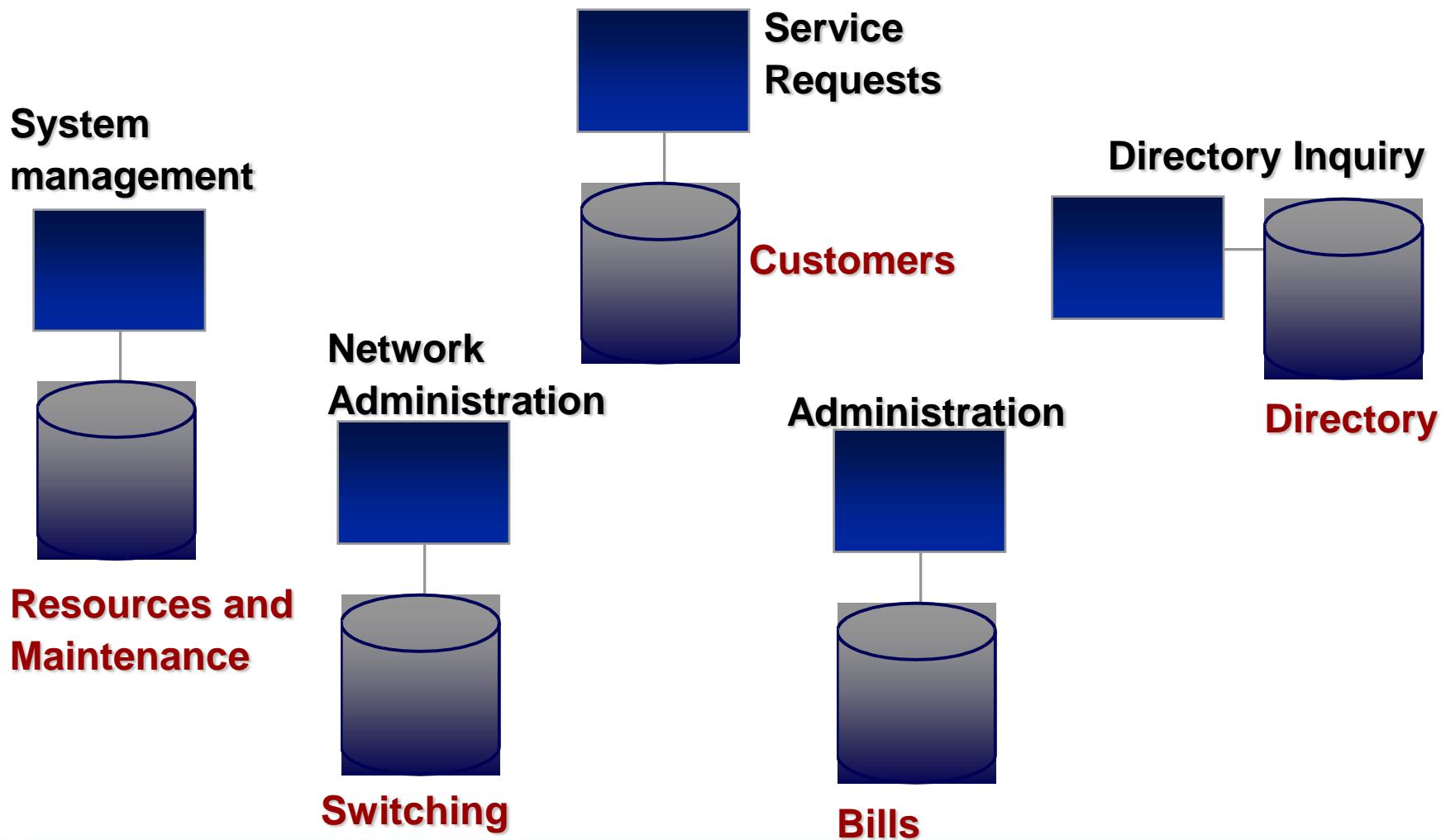


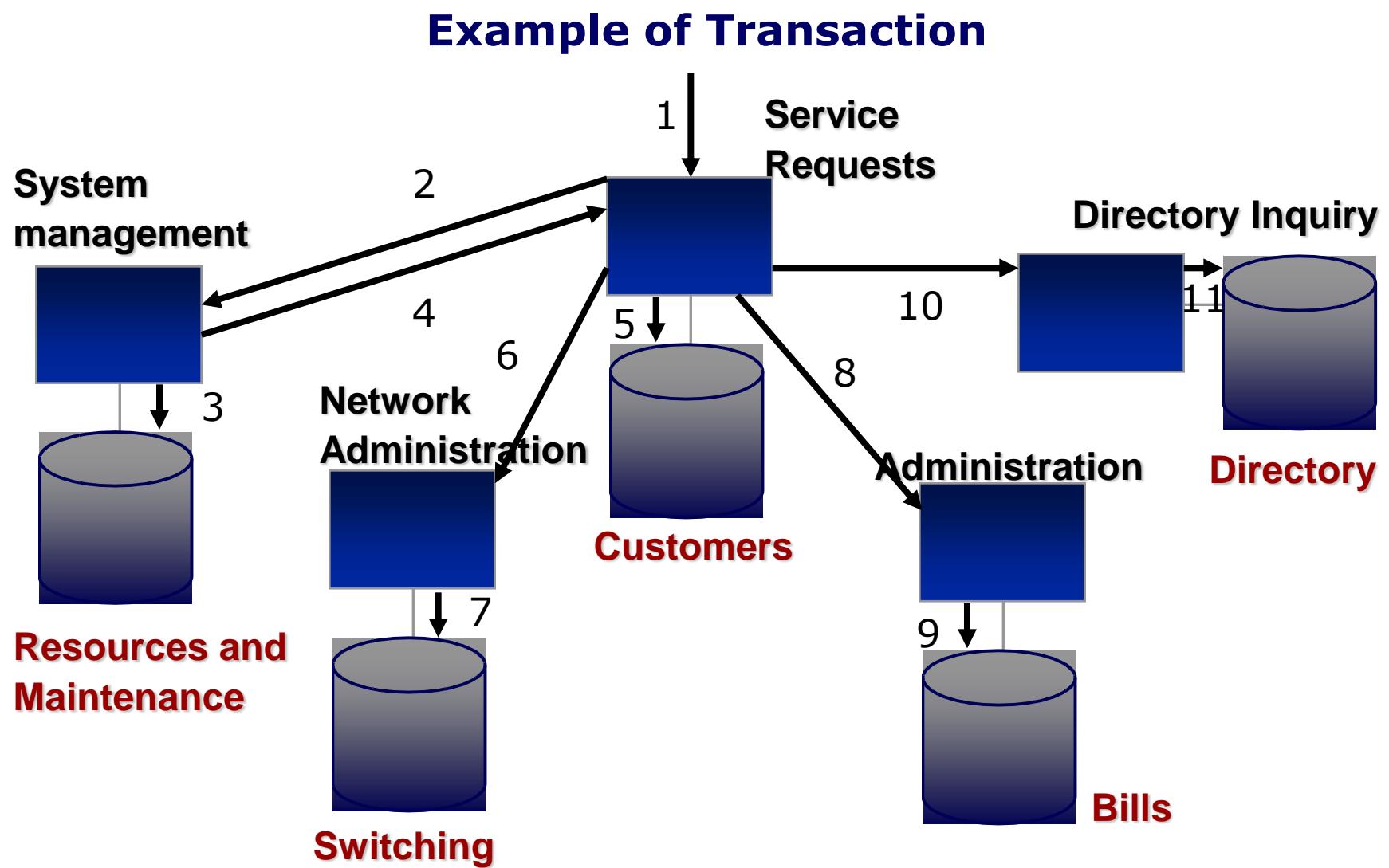
Advanced Databases

1

Transactional Systems

Example of Information System

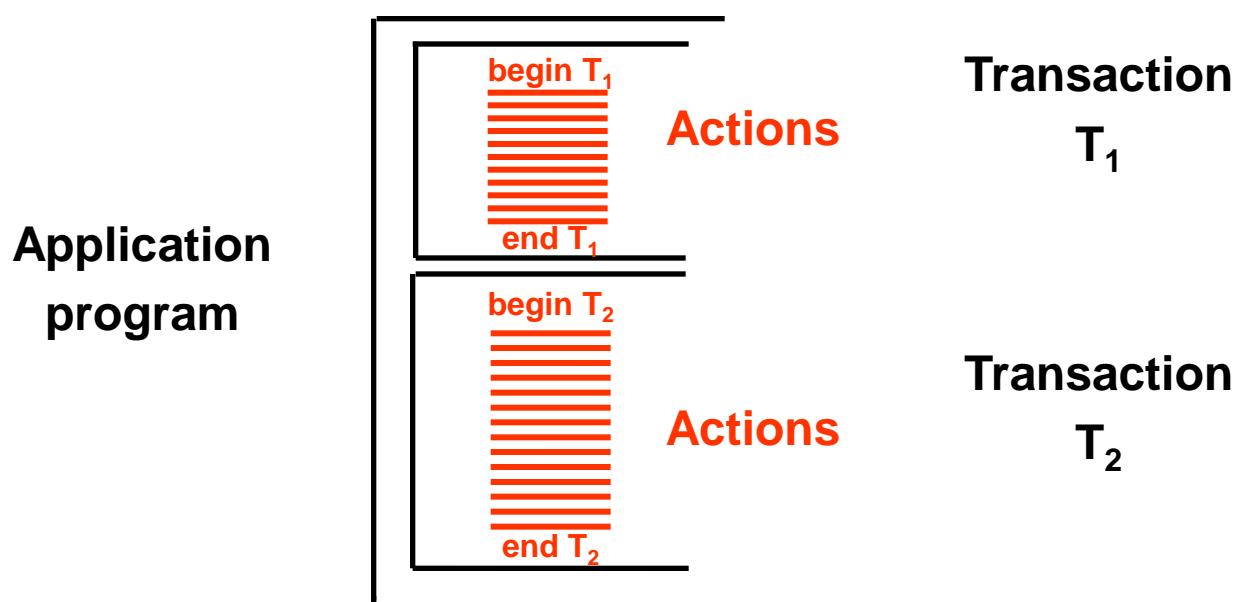




Definition of Transaction

- An elementary unit of work performed by an application
- Each transaction is encapsulated within two commands:
 - **begin transaction** (bot)
 - **end transaction** (eot)
- Within a transaction one of the commands below is executed (exactly once):
 - **commit work** (commit)
 - **rollback work** (abort)
- **Transactional System (OLTP)**: a system capable of providing the definition and execution of transactions on behalf of multiple, concurrent applications

Difference between Application and Transaction



Transaction: Example

```
start transaction;  
update Account  
    set Balance = Balance + 10 where AccNum = 12202;  
update Account  
    set Balance = Balance - 10 where AccNum = 42177;  
commit work;  
end transaction;
```

Transaction: Example with Alternative

```
start transaction;  
update Account  
    set Balance = Balance + 10 where AccNum = 12202;  
update Account  
    set Balance = Balance - 10 where AccNum = 42177;  
select Balance into A from Account  
    where AccNum = 42177;  
if (A>=0)    then commit work  
                else rollback work;  
end transaction;
```

Transactions in JDBC

- Transaction mode is chosen via a method defined in the **Connection** interface
 - setAutoCommit(boolean autoCommit)**
- **con.setAutoCommit(true)**
 - (Default) "autocommit": every single operation is a transaction
- **con.setAutoCommit(false)**
 - Transactions are handled in the program
 - **con.commit()**
 - **con.rollback()**
 - There is no **start transaction**

Well-formed Transactions

- **begin transaction**
- code for data manipulation (reads and writes)
- **commit work – rollback work**
- no data manipulation
- **end transaction**

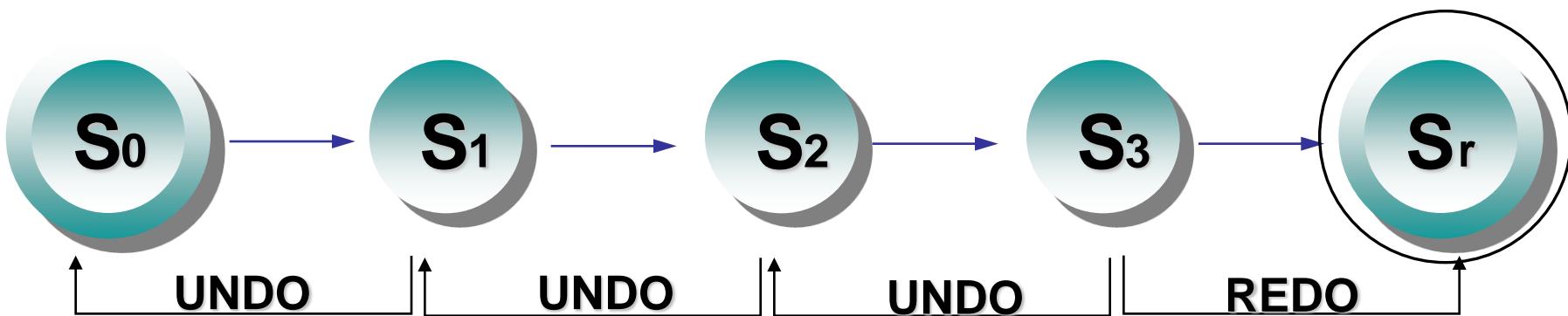


ACID Properties of Transactions

- A transaction is a unit of work enjoying the following properties:
 - Atomicity
 - Consistency
 - Isolation
 - Durability

Atomicity

- A transaction is an atomic transformation from the initial state to the final state
- Possible behaviors:
 1. Commit work: SUCCESS
 2. Rollback work or error prior to commit: UNDO
 3. Fault after commit: REDO

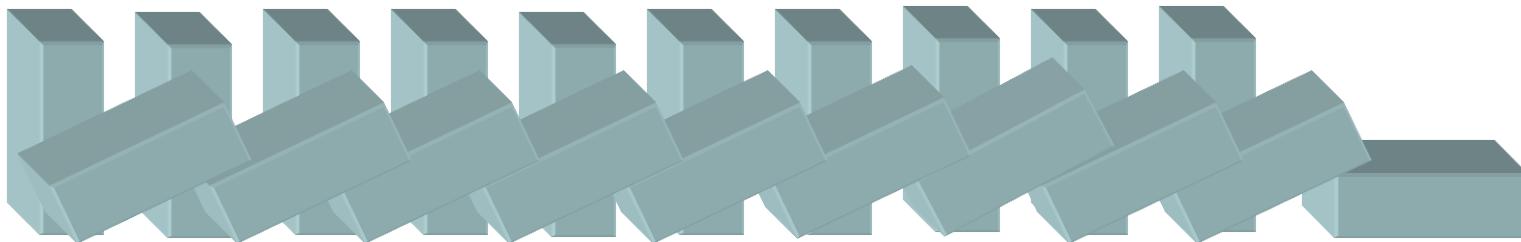


Consistency

- The transaction satisfies the integrity constraints
- Consequence:
 - If the initial state is consistent
 - Then the final state is also consistent

Isolation

- A transaction is not affected by the behavior of other, concurrent transactions
- Consequence:
 - Its intermediate states are not exposed
 - The “domino effect” is avoided



Durability

- The effect of a transaction that has successfully committed will last “forever”
 - Independently of any system fault

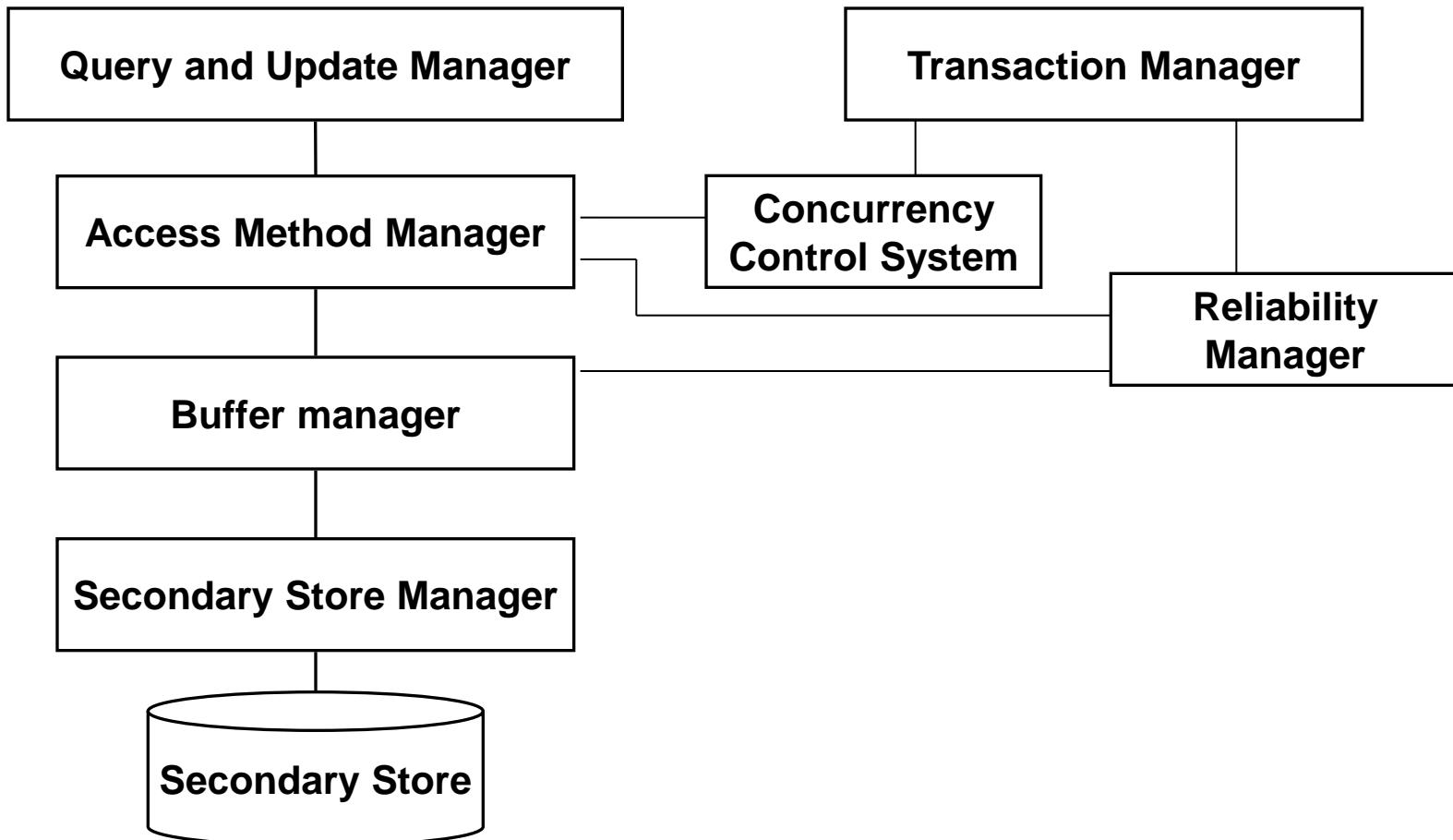
Transaction Properties and Mechanisms

- **A**tomicity
 - Abort-rollback-restart
 - Commit protocols
- **C**onsistency
 - Integrity checking of the DBMS
- **I**solation
 - Concurrency control
- **D**urability
 - Recovery management

Transactions and DBMS modules

- Atomicity and durability
 - Reliability Manager
- Isolation
 - Concurrency Control System
- Consistency
 - Integrity Control System at query execution time
(with the support of the DDL Compilers)

Logical Architecture of a DBMS



Next Topics...

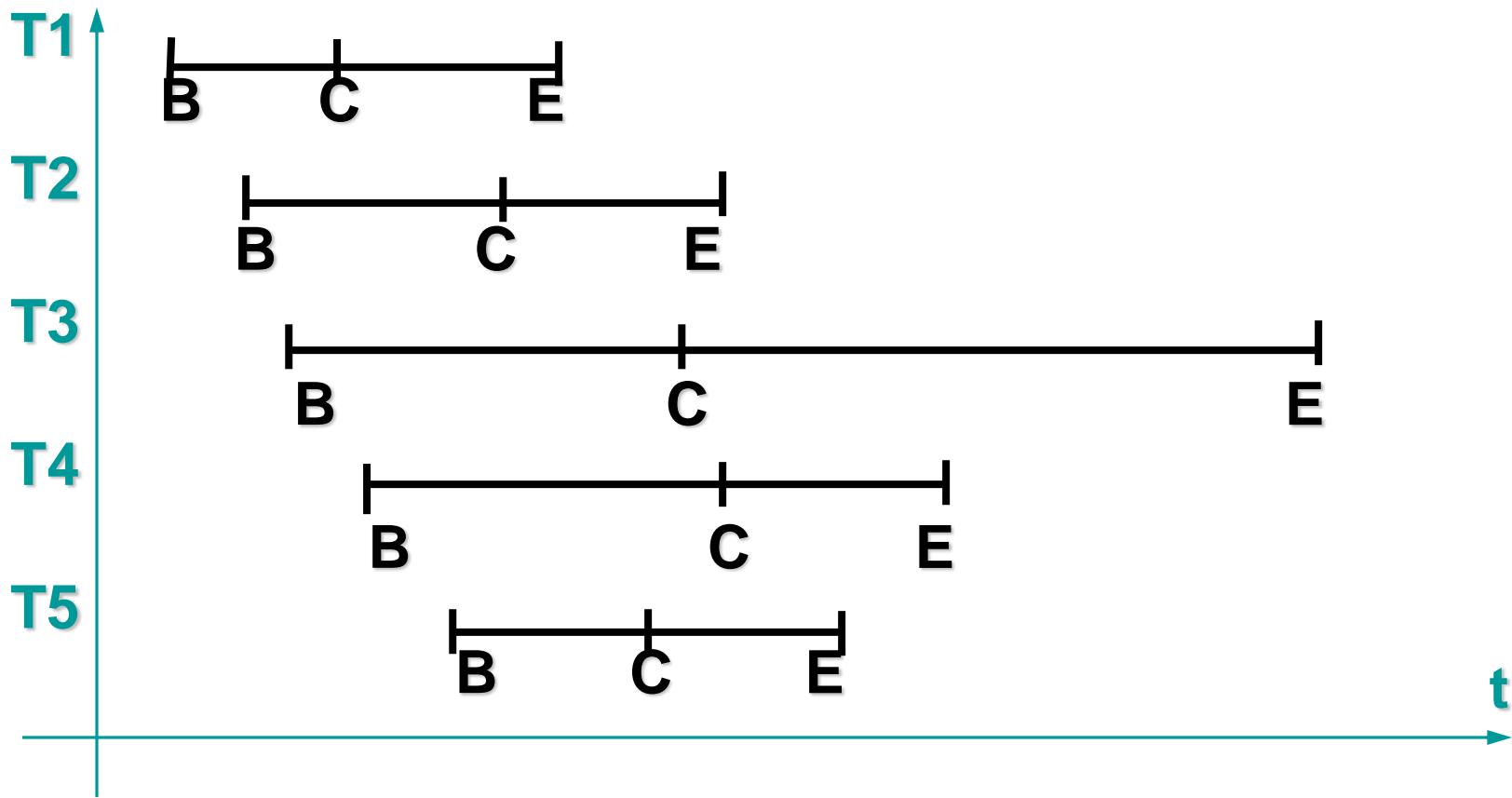
- Concurrency Control
 - Theory
 - 2PL method
- Reliability Control
 - Logging and recovery on a single DBMS
 - Commit protocols and 2PC
- Database Architectures
 - Distribution
 - Parallelism
 - Replication
 - Warehousing

Advanced Databases

2

Concurrency Control

Advantages of Concurrency



Two concurrent transactions

```
T1 : begin transaction  
      UPDATE account  
          SET balance = balance + 3  
          WHERE client = 'Smith'  
      commit work  
      end transaction
```

```
T2 : begin transaction  
      UPDATE account  
          SET balance = balance + 6  
          WHERE client = 'Smith'  
      commit work  
      end transaction
```

Low level view of the transactions/1

```
T1 : begin transaction  
      X:= X + 3  
      commit work  
      end transaction
```

```
T2 : begin transaction  
      X:= X + 6  
      commit work  
      end transaction
```

Low level view of the transactions/2

```
T1 : begin transaction  
      read(X,v1)  
      v1:= v1 + 3  
      write(v1,X)  
      commit work  
      end transaction
```

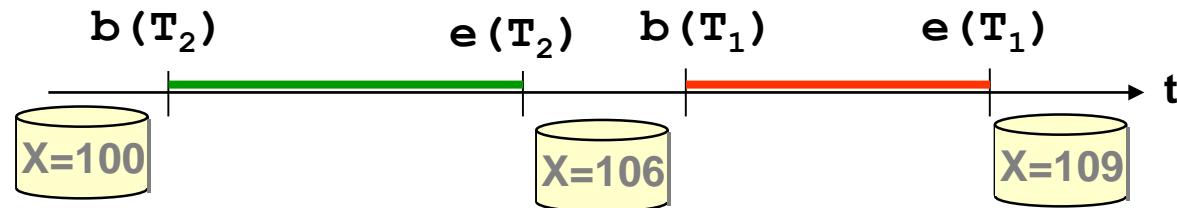
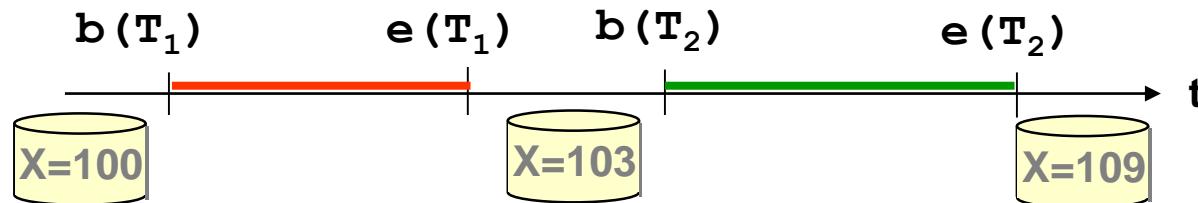
```
T2 : begin transaction  
      read(X,v2)  
      v2:= v2 + 6  
      write(v2,X)  
      commit work  
      end transaction
```

Serial Executions

bot (T₁)
 r (X, v1)
 v1 = v1 + 3
 w (v1, X)
 commit
eot (T₁)

bot (T₂)
 r (X, v2)
 v2 = v2 + 6
 w (v2, X)
 commit
eot (T₂)

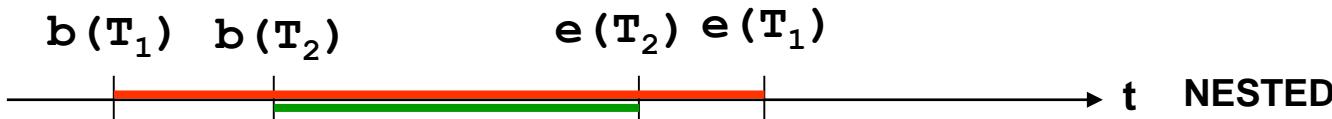
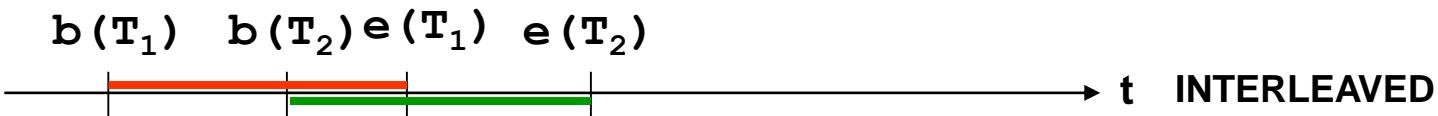
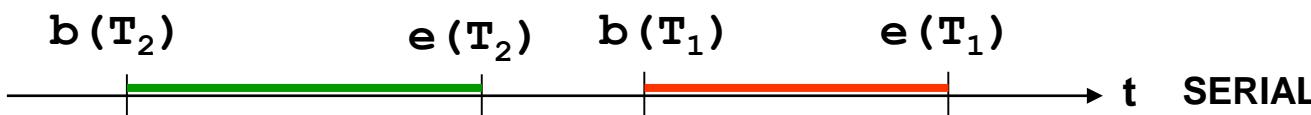
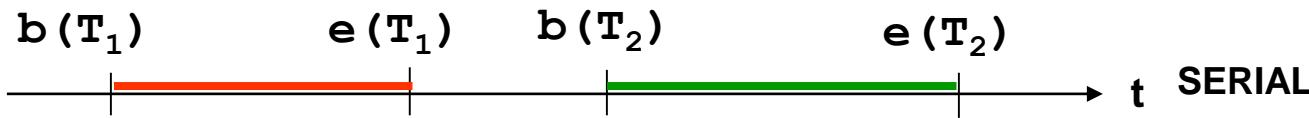
$$x_0 = 100$$



Concurrency Control

- Concurrency is fundamental
 - Tens, hundreds, thousands of transactions per second cannot be executed serially
- Examples: banks, ticket reservations
- Problem: concurrent execution may cause anomalies
 - Concurrency needs to be controlled

Concurrent Executions



Problems due to Concurrency

```
T1 : UPDATE account  
      SET balance = balance + 3  
      WHERE client = 'Smith'
```

```
T2 : UPDATE account  
      SET balance = balance + 6  
      WHERE client = 'Smith'
```

Execution with Lost Update

X=100

- 1 T1: R(X,V1)
- 2 V1 = V1 + 3
- 3 T2: R(X,V2)
- 4 V2 = V2 + 6
- 5 T1: W(V1,X) X=103
- 6 T2: W(V2,X) X=106!

Sequence of I/O Actions producing the Error



or



“Dirty” Read

X=100

- 1 T1: R(X,V1)
- 2 T1: V1 = V1 + 3
- 3 T1: W(V1,X) X=103
- 4 T2: R(X,V2)
- 5 T1: ROLLBACK
- 6 T2: V2 = V2 + 6
- 7 T2: W(V2,X) X=109!

“Nonrepeatable” Read

X=100

1 T1: R(X,V1)

2 T2: R(X,V2)

3 T2: V2 = V2 + 6

4 T2: W(V2,X) X=106

5 T1: R(X,V3) V3<>V1!

Ghost Update

$X+Y+Z=100, X=50, Y=30, Z=20$

T1: R(X,V1), R(Y,V2)

T2: R(Y,V3), R(Z,V4)

T2: $V3 = V3 + 10, V4 = V4 - 10$

T2: W(V3,Y), W(V4,Z) ($Y=40, Z=10$)

T1: R(Z,V5) (for T1, $V1+V2+V5=90!$)

Phantom Insert

T1: $C = \text{AVG}(B : A=1)$

T2: Insert (A=1, B=2)

T1: $C = \text{AVG}(B : A=1)$

- Note: this anomaly does not depend on data already present in the DB when T1 executes, but on a “phantom” tuple that is inserted and satisfies the conditions of a previous query

Anomalies

Lost update

R1-R2-W2-W1

Dirty read

R1-W1-R2-abort1-W2

Nonrepeatable read

R1-R2-W2-R1

Ghost update

R1-R1-R2-R2-W2-W2-R1

Phantom insert

R1-W2 (new data)-R1

Schedule

- Sequence of input/output operations performed by concurrent transactions

$S_1: r_1(x) \ r_2(z) \ w_1(x) \ w_2(z)$

$r_1, w_1 \in T_1$

$r_2, w_2 \in T_2$

Principles of Concurrency Control

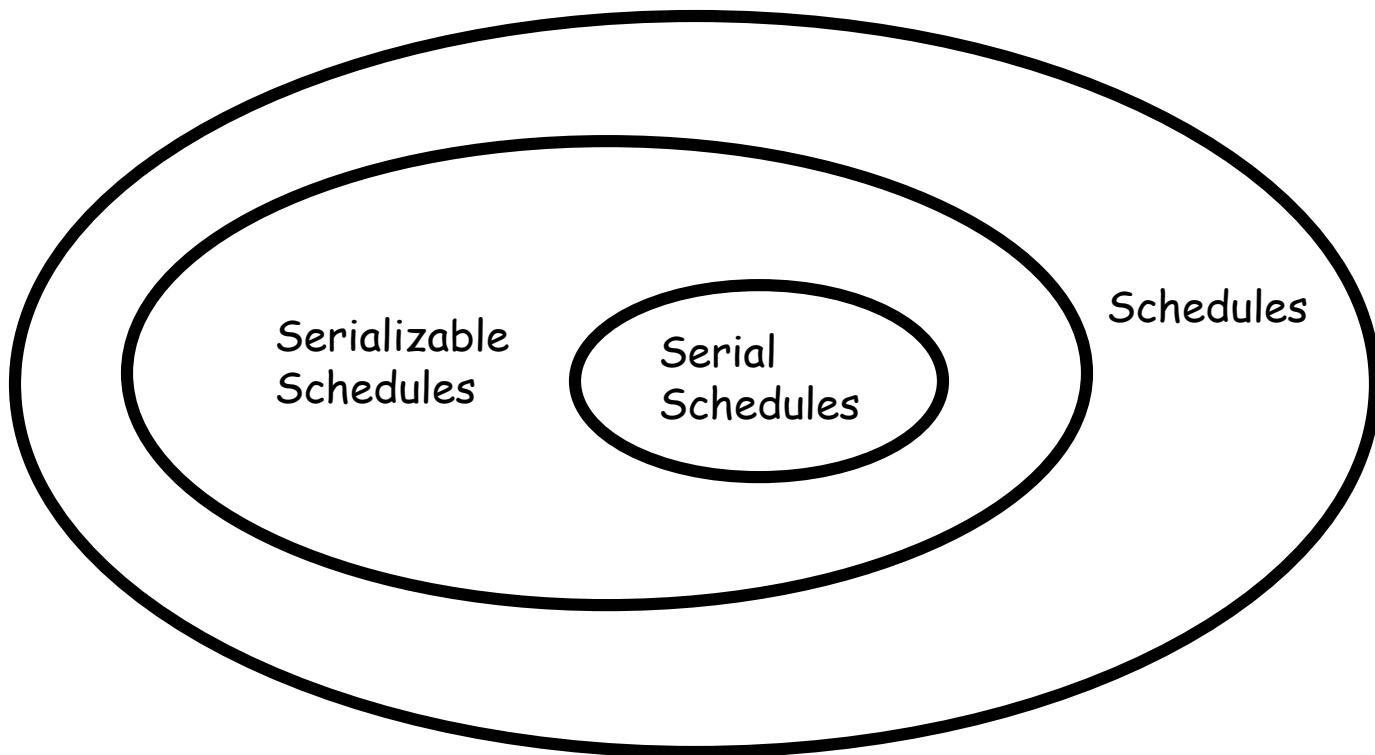
- Goal: to reject schedules that cause anomalies
- *Scheduler*: component that accepts or rejects the operations requested by the transactions
- *Serial schedule*: the actions of each transaction occur in contiguous sequences

$S_2: r_0(x) \ r_0(y) \ w_0(x) \ r_1(y) \ r_1(x) \ w_1(y) \ r_2(x) \ r_2(y) \ r_2(z) \ w_2(z)$

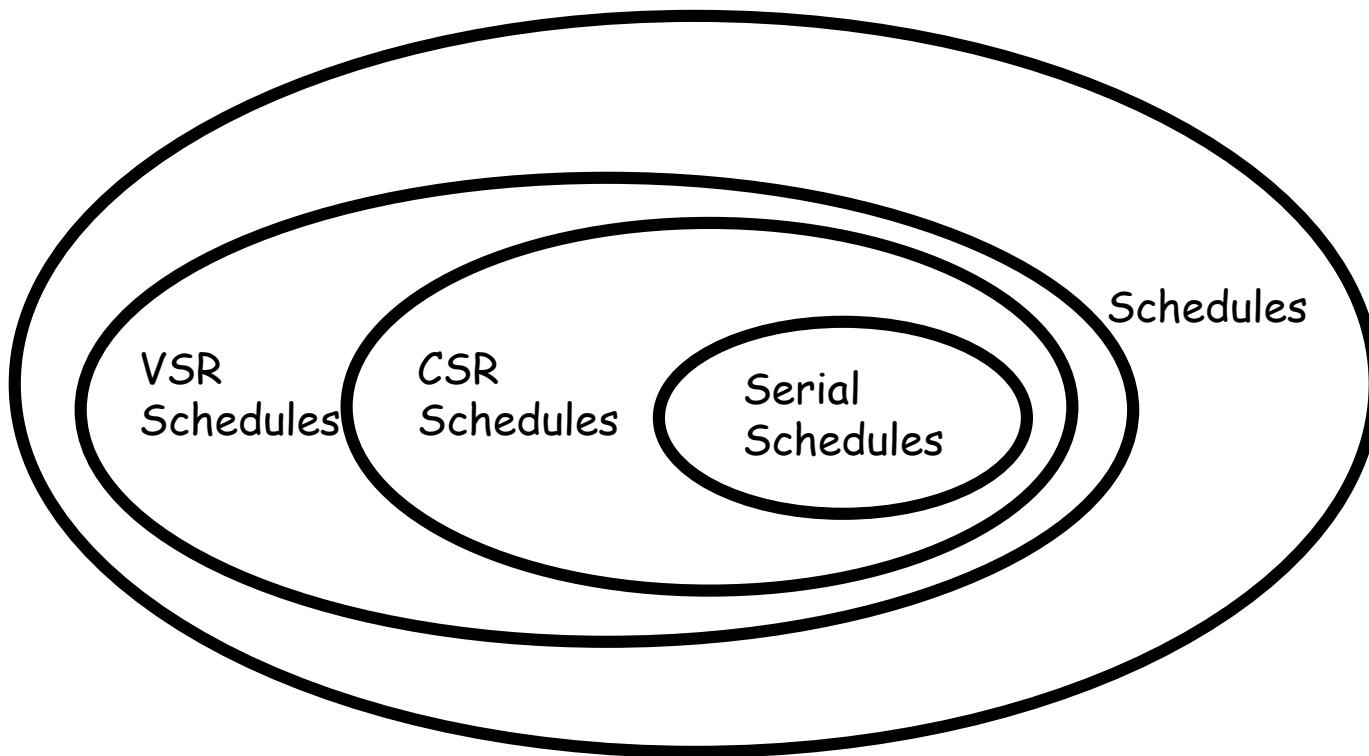
Principles of Concurrency Control

- *Serializable schedule*
 - produces the same results as some serial schedule on the same transactions
 - requires a notion of schedule equivalence
- Note: the class of acceptable schedules produced by a scheduler depends on the cost of equivalence checking
- Assumption
 - We assume that transactions are observed in the “past” (commit-projection) and we want to decide whether the corresponding schedule is correct
 - In practice, schedulers need to decide while the transaction is running

Basic Idea



CSR and VSR



View-serializability

- Preliminary definitions:
 - $r_i(x)$ *reads-from* $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ in S and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$ in S
 - $w_i(x)$ in a schedule S is a *final write* if it is the last write on x that occurs in S
- Two schedules are *view-equivalent* ($S_i \approx_v S_j$): if they have the same reads-from relations and the same final writes
- A schedule is *view-serializable* if it is equivalent to a serial schedule
- **VSR** is the set of view-serializable schedules

Examples of View-serializability

$S_3 : w_0(x) \ r_2(x) \ r_1(x) \ w_2(x) \ w_2(z)$

$S_4 : w_0(x) \ r_1(x) \ r_2(x) \ w_2(x) \ w_2(z)$

$S_5 : w_0(x) \ r_1(x) \ w_1(x) \ r_2(x) \ w_1(z)$

$S_6 : w_0(x) \ r_1(x) \ w_1(x) \ w_1(z) \ r_2(x)$

- S_3 is view-equivalent to serial schedule S_4 (so it is view-serializable)
- S_5 is not view-equivalent to S_4 , but it is view-equivalent to serial schedule S_6 , so it is also view-serializable

Examples of View-serializability 2

$S_7 : r_1(x) \ r_2(x) \ w_1(x) \ w_2(x)$

$S_8 : r_1(x) \ r_2(x) \ w_2(x) \ r_1(x)$

$S_9 : r_1(x) \ r_1(y) \ r_2(z) \ r_2(y) \ w_2(y) \ w_2(z) \ r_1(z)$

- S_7 corresponds to a lost update
- S_8 corresponds to a non-repeatable read
- S_9 corresponds to a ghost update
- They are all not view-serializable

More Complex Example

S_{10} : $w_0(x), r_1(x), w_0(z), \underline{r_1(z)}, r_2(x), w_0(y), r_3(z), w_3(z), w_2(y), w_1(x), w_3(y)$

S_{11} : $w_0(x), w_0(z), w_0(y), r_1(x), r_1(z), \underline{w_1(x)}, r_2(x), w_2(y), r_3(z), w_3(z), w_3(y)$

- The serial order T0, T1, T2, T3 is not view equivalent to the above schedule.
- Let's try T0, T2, T1, T3

More Complex Example

S_{10} : $w_0(x), r_1(x), w_0(z), r_1(z), r_2(x), w_0(y), r_3(z), w_3(z), w_2(y), w_1(x), w_3(y)$

S_{12} : $w_0(x), w_0(z), w_0(y), r_2(x), w_2(y), r_1(x), r_1(z), w_1(x), r_3(z), w_3(z), w_3(y)$

reads-from's OK: $r_1(x)$ da $w_0(x)$,

$r_1(z)$ da $w_0(z)$,

$r_2(x)$ da $w_0(x)$,

$r_3(z)$ da $w_0(z)$,

final writes OK: $w_1(x), w_3(y), w_3(z)$

It is VSR

Complexity of View-serializability

- Deciding view-equivalence of two given schedules can be done in polynomial time
- Deciding view-serializability of a generic schedule is an NP-complete problem

Conflict-serializability

- Preliminary definition:
 - Action a_i is **conflicting** with a_j ($i \neq j$) if both are operations on a common data item and at least one of them is a write operation.
 - *read-write* conflicts (*rw* or *wr*)
 - *write-write* conflicts (*ww*)

Conflict-serializability

- Conflict-equivalent schedules ($S_i \approx_C S_j$): S_i and S_j contain the same operations and all conflicting operation pairs occur in the same order
- S is a conflict-serializable schedule if it is conflict-equivalent to a serial schedule
- CSR is the set of conflict-serializable schedules

CSR and VSR

- Every conflict-serializable schedule is also view-serializable, but the converse is not necessarily true
- Counter-example:

$$r_1(x) \; w_2(x) \; w_1(x) \; w_3(x)$$

- View-serializable: view-equivalent to

$$r_1(x) \; w_1(x) \; w_2(x) \; w_3(x)$$

- Not conflict-serializable, due to the presence of:

$$r_1(x) \; w_2(x) \text{ and } w_2(x) \; w_1(x)$$

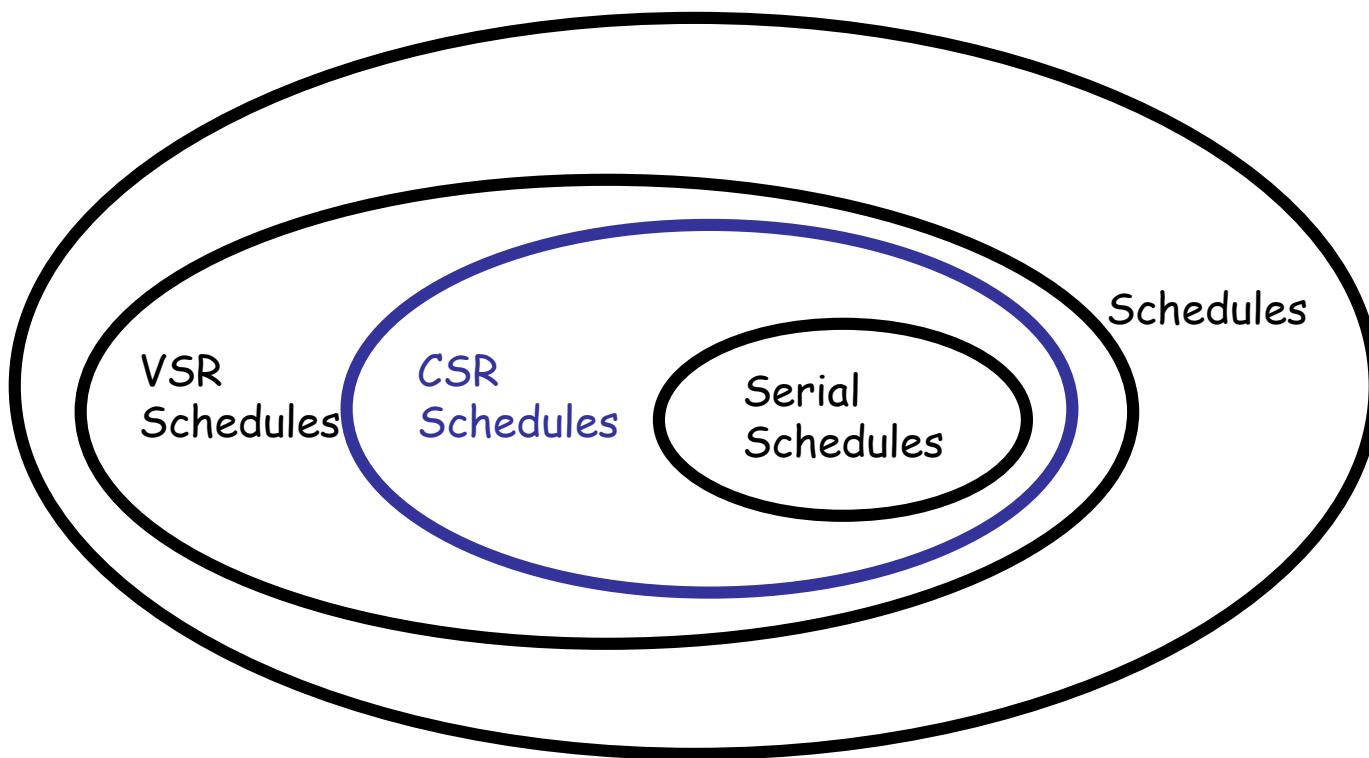
CSR implies VSR

- In order to prove that CSR implies VSR it suffices to prove that
 - Conflict-equivalence \approx_C implies view-equivalence \approx_V ,
 - i.e., if two schedules are \approx_C then they also are \approx_V

CSR implies VSR

- Let's suppose $S_1 \approx_C S_2$. We prove that $S_1 \approx_V S_2$. These schedules have:
 - The same final writes: if they didn't, there would be at least two writes with a different order, and since two writes are conflicting operations, the schedules would not be \approx_C
 - The same "reads-from" relations: if not, there would be read-write pairs in a different order and therefore, as above, \approx_C would be violated

CSR and VSR

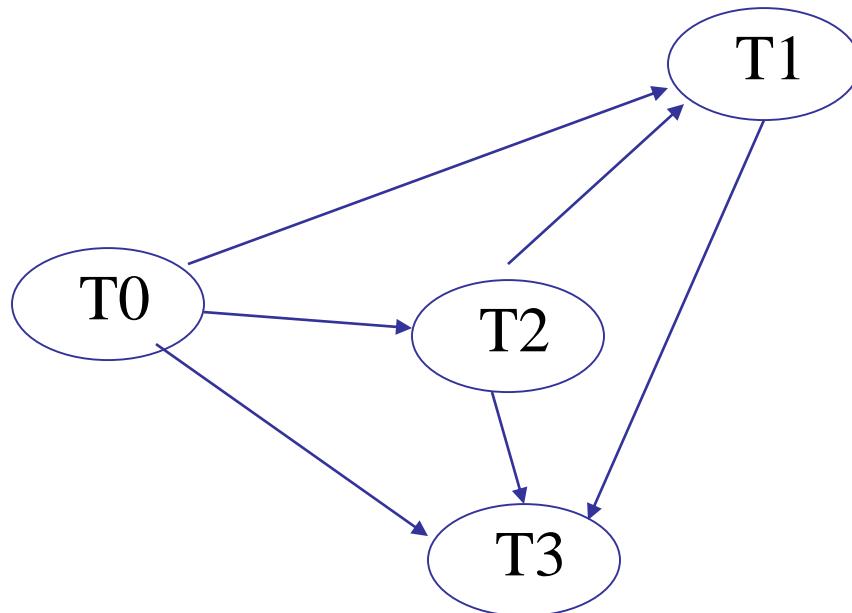


Testing Conflict-serializability

- Is done with a *conflict graph* that has:
 - One node for each transaction T_i
 - One arc from T_i to T_j if there exists at least one conflict between an action a_i of T_i and an action a_j of T_j such that a_i precedes a_j
- Theorem:
 - A schedule is in CSR if and only if its conflict graph is acyclic

Example Test

- $w_0(x), r_1(x), w_0(z), r_1(z), r_2(x), w_0(y), r_3(z), w_3(z), w_2(y), w_1(x), w_3(y)$



CSR implies Acyclicity of the Conflict Graph

- Consider a schedule S in CSR. As such it is \approx_C to a serial schedule.
- Suppose the order of transactions in the serial schedule is: t_1, t_2, \dots, t_n
- Since the serial schedule has all conflicting pairs in the same order as schedule S, in S's graph there can only be arcs (i,j) , with $i < j$
- Then the graph is acyclic, as a cycle requires at least an arc (i,j) with $i > j$

Properties of the Conflict Graph

- If S's graph is acyclic then it has a *topological sort*, i.e., an ordering of the nodes such that the graph only contains arcs (i,j) with $i < j$
- The serial schedule whose transactions are ordered according to the topological sort is conflict-equivalent to S, because for all conflicting pairs (i,j) it is always $i < j$
 - In the example before: $T_0 < T_2 < T_1 < T_3$
 - In general there can be MANY topological sorts (i.e. serializations for the same acyclic graph)

Concurrency Control in Practice

- This technique would be efficient if we knew the graph from the beginning — but we don't
- A scheduler must work “incrementally”, i.e., for each requested operation it should decide whether to execute it immediately or do something else
- It is not feasible to maintain the graph, update it and verify its acyclicity at each operation request

Locking

- It's the most common method in commercial systems
- A transaction is well-formed wrt locking if
 - **read** operations are preceded by **r_lock** (SHARED LOCK) and followed by **unlock**
 - **write** operations are preceded by **w_lock** (EXCLUSIVE LOCK) and followed by **unlock**
- When a transaction first reads and then writes an object it can:
 - Use a **w_lock**
 - Modify a **r_lock** into a **w_lock** (lock escalation)

Lock Primitives

- Primitives:
 - **r-lock**: read lock
 - **w-lock**: write lock
 - **unlock**
- Possible states of an object:
 - **free**
 - **r-locked** (locked by a reader)
 - **w-locked** (locked by a writer)

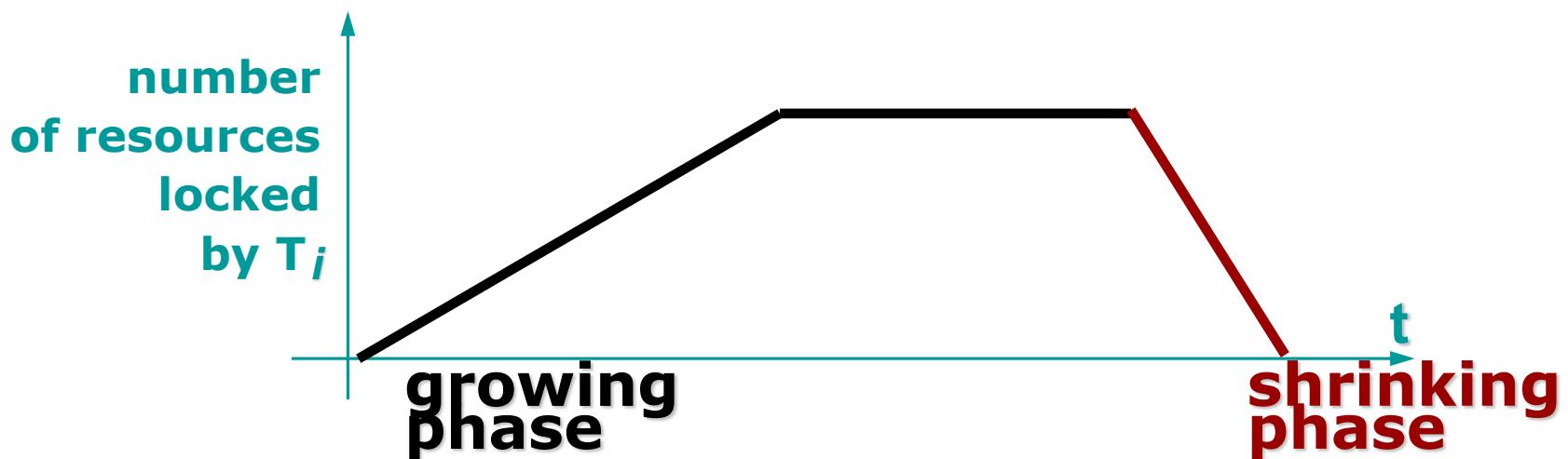
Behavior of the Lock Manager

- The lock manager receives the primitives from the transactions and grants resources according to the **conflict table**
 - When a **lock** request is granted, the resource is acquired
 - When an **unlock** is executed, the resource becomes available

REQUEST	RESOURCE STATE		
	FREE	R_LOCKED	W_LOCKED
r_lock	OK R_LOCKED	OK R_LOCKED	NO W_LOCKED
w_lock	OK W_LOCKED	NO R_LOCKED	NO W_LOCKED
unlock	ERROR	OK DEPENDS	OK FREE

Two-Phase Locking

- Requirements:
 - A transaction cannot acquire any other lock after releasing a lock



Serializability

- If a scheduler
 - uses well-formed transactions
 - grants locks according to conflicts
 - is two-phase
- Then it produces the schedule class called **2PL**,

Schedules in 2PL are serializable

2PL and CSR

- Every 2PL schedule is also conflict-serializable, but the converse is not necessarily true
- Counter-example:

$$r_1(x) \text{ } w_1(x) \text{ } r_2(x) \text{ } w_2(x) \text{ } r_3(y) \text{ } w_1(y)$$

- It violates 2PL

$$r_1(x) \text{ } w_1(x) \mid r_2(x) \text{ } w_2(x) \text{ } r_3(y) \mid w_1(y)$$

T1 releases T1 acquires

- It is conflict-serializable

$$T3 < T1 < T2$$

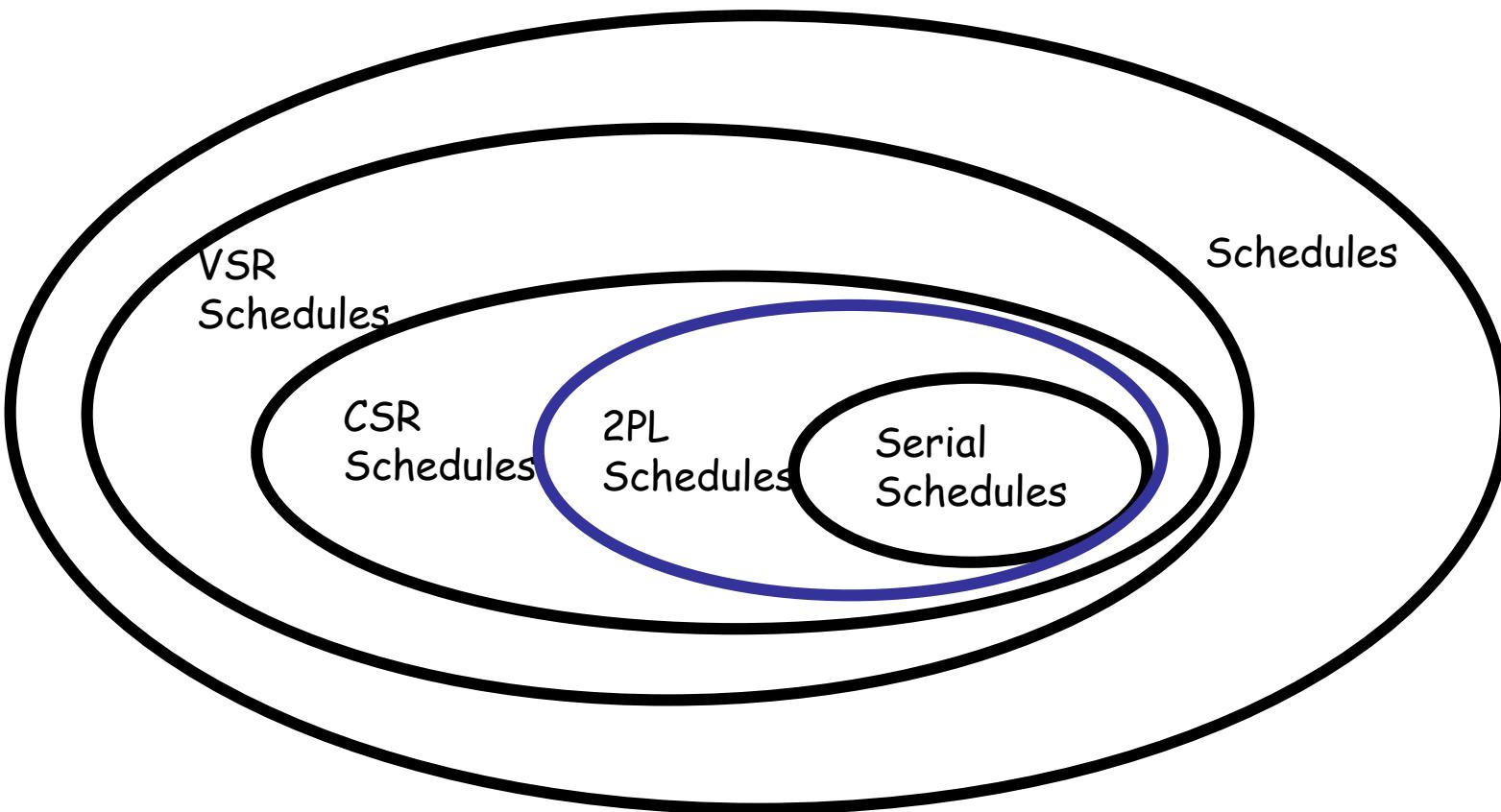
2PL implies CSR

- Consider for each transaction the moment in which it has all resources and is going to release the first one
- We sort the transactions by this temporal value and consider the corresponding serial schedule

2PL implies CSR

- We want to prove that this schedule is conflict-equivalent to S:
 - We then consider a conflict between an action from t_i and an action from the t_j 's with $i < j$
 - Can they occur in the reverse order in S?
 - No, because then t_j should have released the resource in question before t_i has acquired it

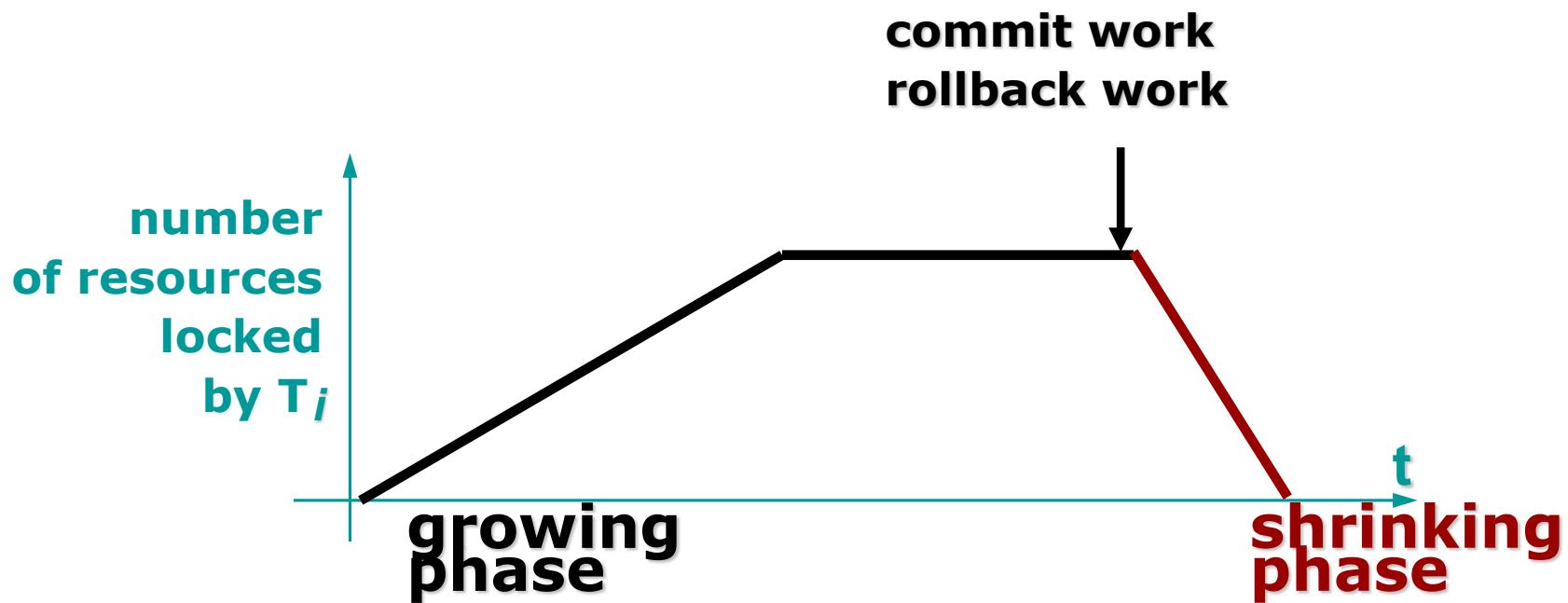
CSR, VSR and 2PL



Strict 2PL

- We were still using the hypothesis of commit-projection
- To remove this hypothesis, we need to add a constraint to 2PL, thus obtaining **strict 2PL**:
 - *Locks on a transaction can be released only after commit/rollback*
- This version of 2PL is used in commercial DBMSs

Strict 2PL in Practice



Implementation of 2-Phase Locking

- Lock tables are in reality **main memory data structures**.
 - Resource state is either *Free*, or *Read-Locked*, or *Write-locked*
 - To keep track of readers, every resource has also a “read counter”
 - Some late-ninety systems only supported exclusive locks (means: binary info for resources, no counter)
- A transaction asking for a lock is either granted a lock or **queued and suspended**, the queue is first-in first-out; there is a danger of:
 - Deadlock: endless wait
 - Starvation: individual transaction waiting forever
 - Starvation can occur for write transactions waiting for resources which are highly used for reading (e.g. index roots).

Isolation Levels in SQL:1999 (and JDBC)

- Writes are always applied strict 2PL (so update loss is avoided)
- **READ UNCOMMITTED** allows dirty reads, nonrepeatable reads and phantoms:
 - No read lock (and ignores locks of other transactions)
- **READ COMMITTED** prevents dirty reads but allows nonrepeatable reads and phantoms:
 - Read locks (and complies with locks of other transactions), but without 2PL

Isolation Levels in SQL:1999 (and JDBC)

- **REPEATABLE READ** avoids dirty reads, nonrepeatable reads and phantom updates, but allows for phantom inserts:
 - 2PL also for reads, with data locks
- **SERIALIZABLE** avoids all anomalies:
 - 2PL with predicate locks

Predicate Locks

- With $R(A,B)$, let: $T = \text{update } R \text{ set } B=1 \text{ where } A=1$
- Then, the lock is on predicate $A=1$
 - Cannot insert, delete, update any tuple satisfying such predicate
- Worst case:
 - On the entire relation
- If we are lucky:
 - On the index

Hierarchical Locking

- In many real systems, locks are specified with different granularities, e.g., database, table, fragment, page, tuple, field. These resources are in a hierarchy (or in a DAG).
- The choice of the lock level depends on the application:
 - Too coarse: many locked resources
 - Too fine: many lock requests

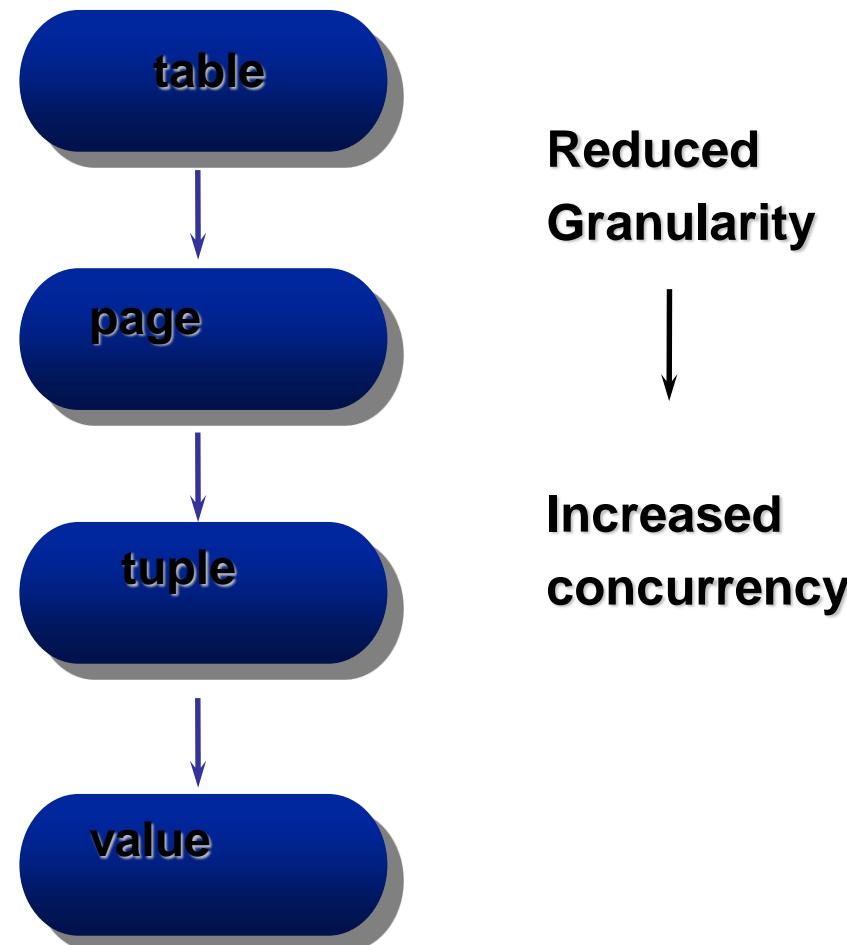
Hierarchical Locking

- Concept:
 - Locking can be done upon objects at various levels of granularity.
- Objectives:
 - Setting the minimum number of lockings
 - Recognizing conflicts as soon as possible
- Organization: asking locking upon resources organized as a hierarchy
 - Requesting resources top-down until the right level is obtained
 - Releasing locks bottom-up.

Resources Managed through Hierarchical Locking

lock
at the level of:

table
page
tuple
Attribute value



Enhanced Locking Scheme

- 5 Lock modes:
 - In addition to read lock (r-lock) and write lock (w-lock), renamed for historical reasons into Shared Locks (SL) and Exclusive Locks (XL).
- The new modes define “intention of locking at lower levels of granularity”.
 - ISL: Intention of locking in shared mode
 - IXL: Intention of locking in exclusive mode
 - SIXL: Lock in shared mode with intention of locking in exclusive mode (SL+IXL)

Conflicts in Hierarchical Locks

Request	Resource state				
	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Hierarchical Locking Protocol

- Locks are requested starting from the root and going down in the hierarchy
- Locks are released starting from the leaves and then going up in the hierarchy
- To request an SL or ISL lock on a non-root node, a transaction must hold an ISL or IXL lock on its parent node
- To request an IXL, XL or SIXL lock on a non-root note, a transaction must hold a SIXL or IXL lock on its parent node

Example

t1	t5
t2	t6
t3	t7
t4	t8

Page 1: t1,t2,t3,t4
Page 2: t5,t6,t7,t8

Transaction 1:
read(P1)
write(t3)
read(t8)

t1	t5
t2	t6
t3	t7
t4	t8

Transaction 2:
read(t2)
read(t4)
write(t5)
write(t6)

They are NOT in conflict!

Lock Sequences

t1	t5
t2	t6
t3	t7
t4	t8

Transaction 1:

IXL(root)
SIXL(P1)
XL(t3)
ISL(P2)
SL(t8)

Transaction 2:

IXL(root)
ISL(P1)
SL(t2)
SL(t4)
IXL(P2)
XL(t5)
XL(t6)

t1	t5
t2	t6
t3	t7
t4	t8

They are NOT in conflict!

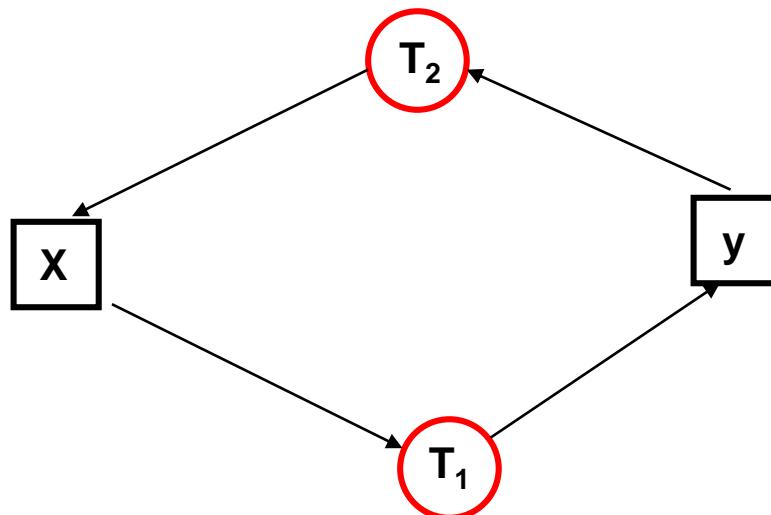
Deadlock

- Occurs because concurrent transactions hold and, in turn, require resources held by other transactions
- $T_1: r_1(x) w_1(y)$
- $T_2: r_2(y) w_2(x)$

S: $r_lock_1(x) r_lock_2(y) r_1(x) r_2(y) w_lock_1(y) w_lock_2(x)$

Deadlock

- A deadlock is represented by a cycle in the WAIT-FOR graph of the resources



Deadlock Resolution Techniques

- Timeout
 - Transactions killed after a long wait
- Deadlock prevention
 - Transactions killed when they COULD BE in deadlock
- Deadlock detection
 - Transactions killed when they ARE in deadlock

Timeout Method

- A transaction is killed after given waiting, assuming it is involved in a deadlock
- Simplest, most used method
- Timeout value is system-determined (sometimes it can be altered by the database administrator)
- The problem is choosing a proper value
 - Too long: useless wait
 - Too short: unrequired kills

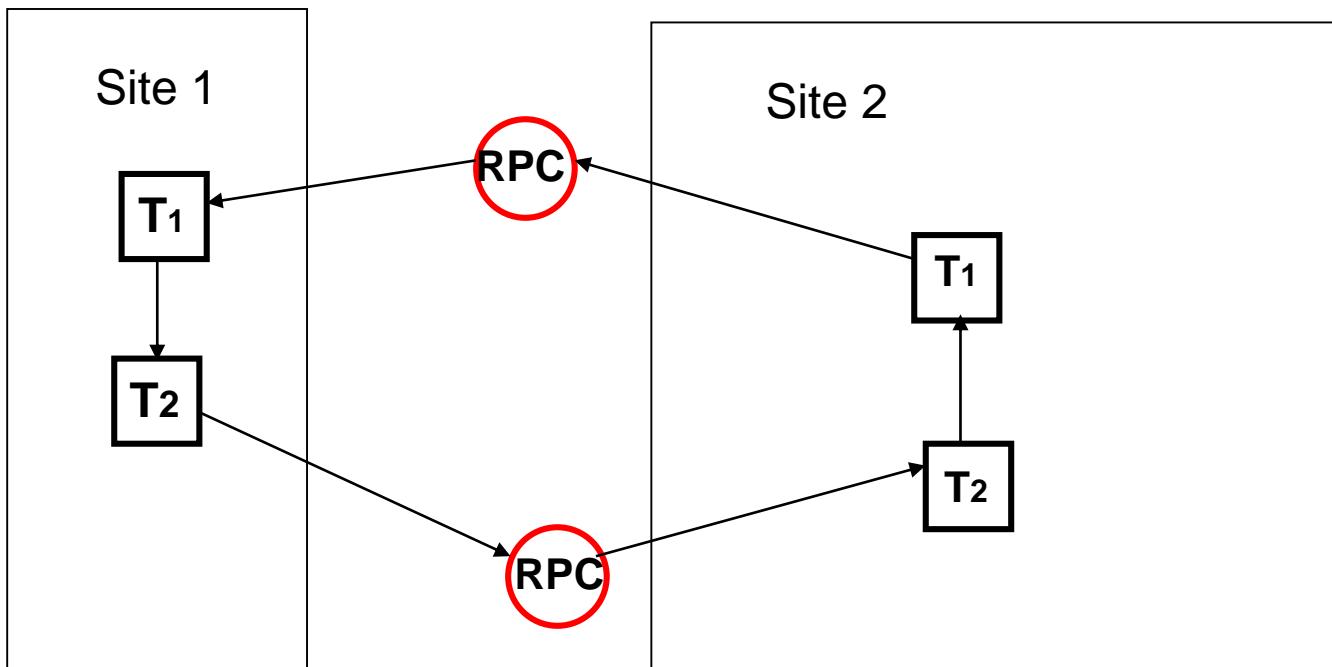
Deadlock Prevention

- Kills transactions that could cause cycles
- One possible scheme is:
 - assigning transaction numbers (assigning transactions an “age”)
 - killing transactions when “older” transactions wait for “younger” transactions
- Options for choosing the transaction to kill
 - Pre-emptive (killing the waiting transaction)
 - Non-pre-emptive (killing the requesting transaction)
- The problem: too many “killings” (waiting probability vs deadlock probability)

Deadlock Detection

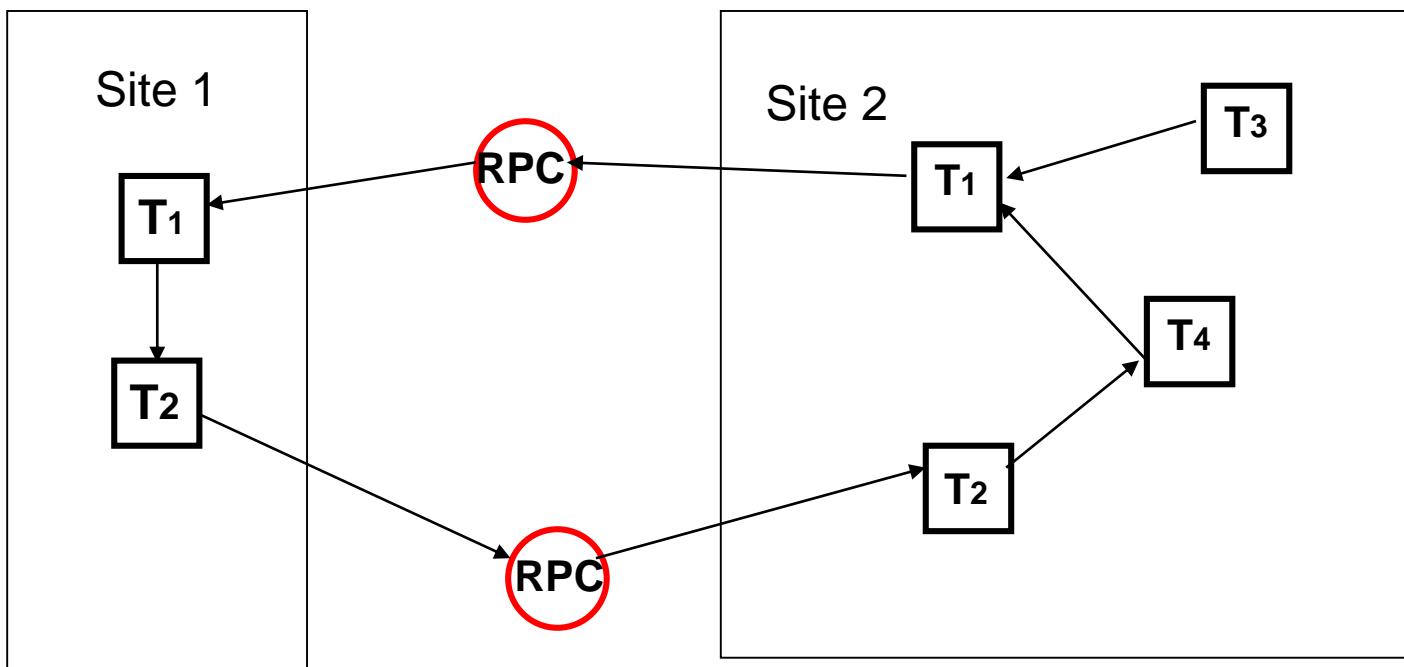
- Requires an algorithm for finding real cycles in the wait-for graph
 - Must work with distributed resources
 - Must be efficient and reliable
- The best solution: Obermark's algorithm (DB2-IBM, published on ACM-Transactions on Database Systems)
 - Assumes synchronous transactions, each transaction works at a single site
 - Assumes communications via "remote procedure calls"
 - Both assumptions can be easily removed.

Distributed Deadlock Detection: Problem Setting



Potential Deadlock: at Site 1: E - T1 - T2 - E
at Site 2: E - T2 - T1 - E

Distributed Deadlock Detection: Problem Setting

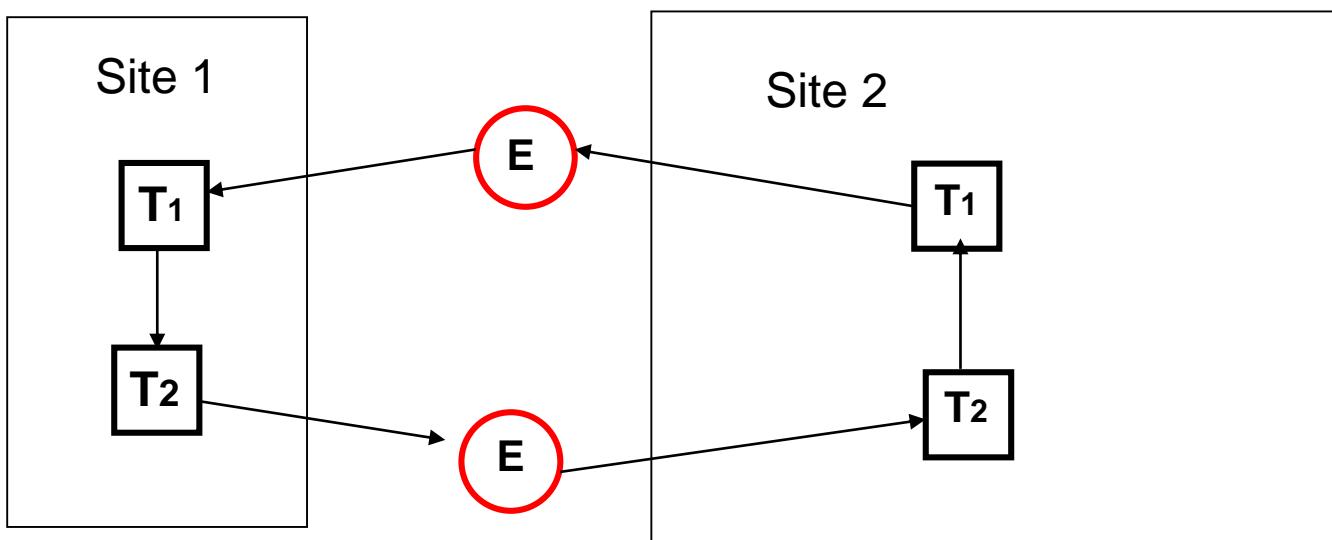


Potential Deadlock: at Site 1: E - T₁ - T₂ - E
at Site 2: E - T₂ - T₁ - E

Obermark's Algorithm

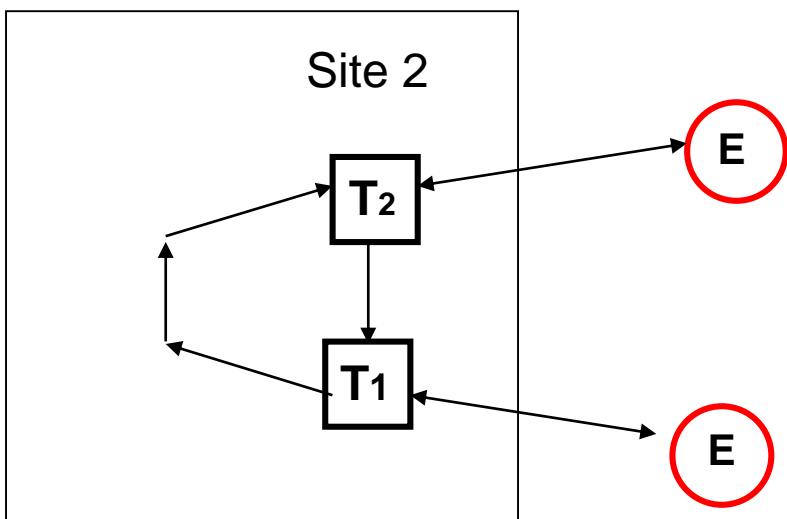
- Runs periodically at each site
- Consists of 4 steps
 - *Get potential deadlock from the “previous” nodes*
 - *Integrate deadlock cycles with local wait-for graph*
 - *Search deadlocks, if found kill one transaction*
 - *Build potential deadlock and transmit to the “next” node*
- Secondary objective: detect every cycle only once; achieved by define “previous” and “next” node as follows:
 - Potential deadlock transmitted along the RPC chain
 - Potential deadlock $E - Ti - Tj - E$ transmitted only if $i < j$

Algorithm execution, 1



Potential Deadlock at Site 1: E - T₁ - T₂ - E sent to 2
at Site 2: E - T₂ - T₁ - E not sent to 1

Algorithm execution, 2



1. E - T₁ - T₂ - E sent to 2
2. at Site 2:
E - T₁ - T₂ - E added
Deadlock detected
T₁ or T₂ killed (rollback)

Another example

- Initially: at site 1, $E > T_1 > T_2 > E$
 2, $E > T_2 > T_3 > E$
 3, $E > T_3 > T_1 > E$
- Sites 1 and 2 can send info, 3 cannot
- Start with 1 sending to 2
- At site 2, a new potential deadlock $E > T_1 > T_3 > E$ is found by combining the incoming $E > T_1 > T_2 > E$ and present $E > T_2 > T_3 > E$; this is sent to 3.
- At site 3, $E > T_1 > T_3 > E$ is combined with $E > T_3 > T_1 > E$, the deadlock is found, and either T_1 or T_3 is killed.

Deadlocks in practice

- Their probability is much less than the conflict probability
 - Consider a file with n records and two transactions doing two accesses to their records (uniform distribution); then:
 - Conflict probability is $O(1/n)$
 - Deadlock probability is $o(1/n^2)$
- They do occur (once every minute for a mid-size bank)
- Probability rises linearly with the number of transactions and quadratically with transaction size (number of lock requests)

Update Lock

- The most frequent deadlock occurs when 2 concurrent transactions start by reading the same resource and then they decide to write and escalate or write-lock it.
- To avoid this situation, systems offer the UPDATE LOCK (UL) – used by transactions that may change the resource value.

Request	State		
	SL	UL	XL
SL	OK	OK	No
UL	OK	No	No
XL	No	No	No

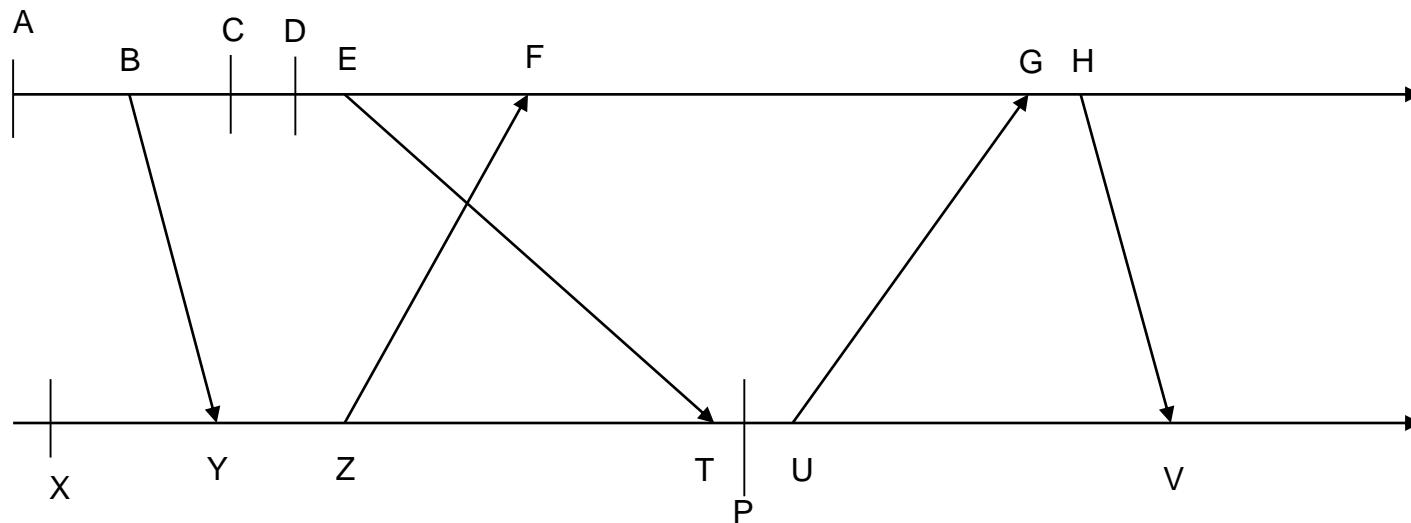
Concurrency Control Based on Timestamps

- Alternative to 2PL
- Timestamp:
 - *Identifier defining a total ordering of the events of a system*
- Each transaction has a timestamp representing the time at which the transaction begins
- A schedule is accepted only if it reflects the serial ordering of the transactions induced by their timestamps

Assigning timestamps

- Timestamp: an indicator of the “current time”
- Assumption: no “global time available”
- Mechanism: a system’s function gives out timestamps on requests.
- Syntax: $\text{timestamp} = \text{event-id}.\text{node-id}$ (event-ids are unique at each node).
- Synchronization: send-receive of messages (for a given message m , $\text{send}(m)$ precedes $\text{receive}(m)$)
- Algorithm: cannot receive a message from “the future”, if this happens the “bumping rule” is used to bump the timestamp of the receive beyond the timestamp of the send.

Example of timestamp assignment



A(1,1), B(2,1), X(1,2), Y(2,2), C(3,1), D(4,1), E(5,1),
Z(3,2), F(6,1), T(5,2), P(6,2), U(7,2), G(8,1), H(9,1),
V(9,2)

Timestamp Mechanism

- The scheduler has two counters: $\text{RTM}(x)$ and $\text{WTM}(x)$ for each object
- The scheduler receives read and write requests with timestamps:
 - $\text{read}(x, ts)$:
 - If $ts < \text{WTM}(x)$ the request is rejected and the transaction killed
 - Else, the request is granted and $\text{RTM}(x)$ is set to $\max(\text{RTM}(x), ts)$
 - $\text{write}(x, ts)$:
 - If $ts < \text{WTM}(x)$ or $ts < \text{RTM}(x)$ the request is rejected and the transaction killed
 - Else, the request is granted and $\text{WTM}(x)$ is set to ts
- Many transactions are killed
- To work w/o the commit-projection hypothesis, it needs to "buffer" write operations until commit, which introduces waits

Example

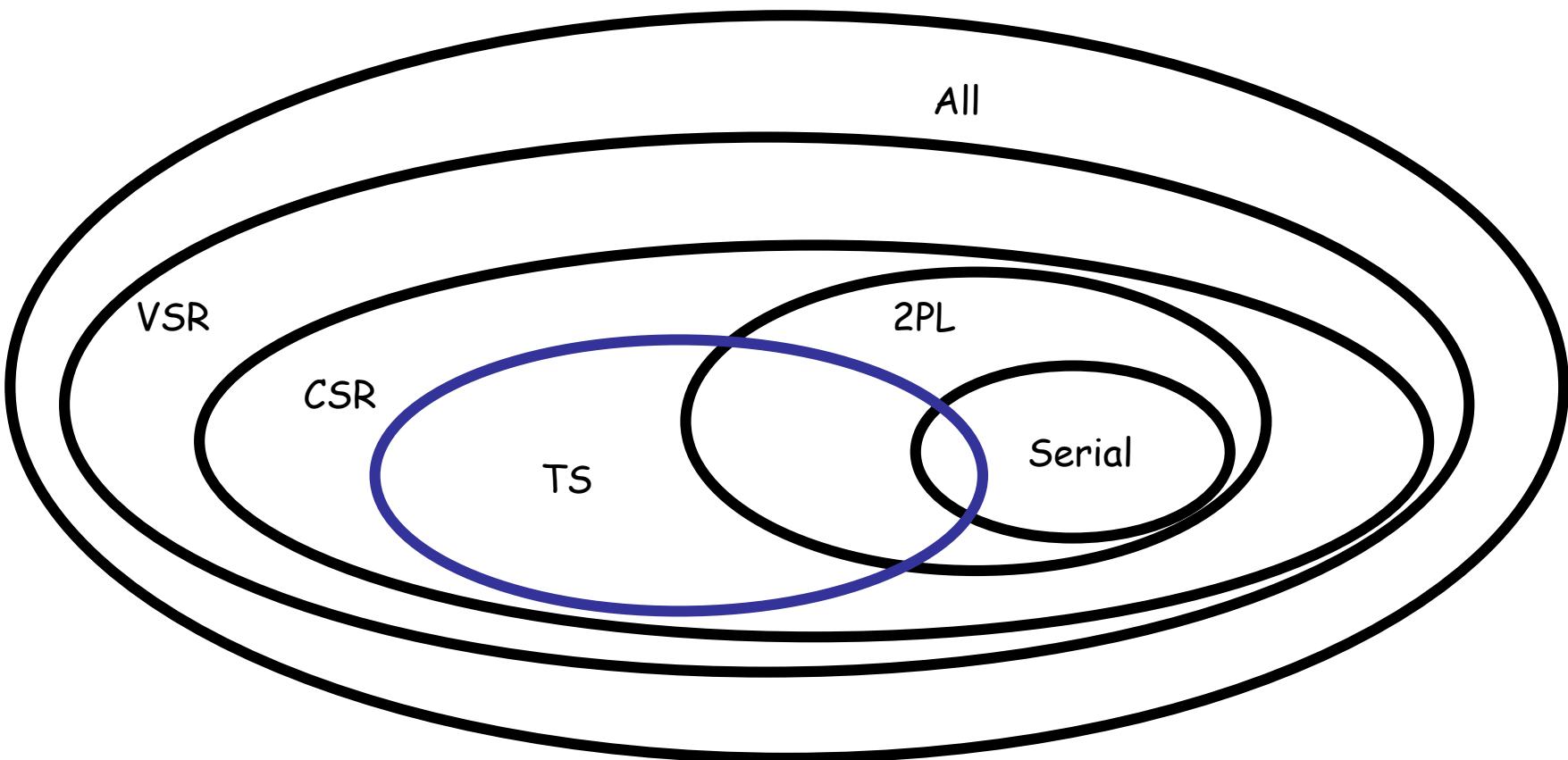
Assume $\text{RTM}(x) = 7$
 $\text{WTM}(x) = 4$

Request	Response	New value
$\text{read}(x,6)$	ok	
$\text{read}(x,8)$	ok	$\text{RTM}(x) = 8$
$\text{read}(x,9)$	ok	$\text{RTM}(x) = 9$
$\text{write}(x,8)$	no	t_8 killed
$\text{write}(x,11)$	ok	$\text{WTM}(x) = 11$
$\text{read}(x,10)$	no	t_{10} killed

2PL vs. TS

- They are incomparable
 - Schedule in TS but not in 2PL
$$r_1(x) \; w_1(x) \; r_2(x) \; w_2(x) \; r_0(y) \; w_1(y)$$
 - Schedule in 2PL but not in TS
$$r_2(x) \; w_2(x) \; r_1(x) \; w_1(x)$$
 - Schedule in TS and in 2PL
$$r_1(x) \; r_2(y) \; w_2(y) \; w_1(x) \; r_2(x) \; w_2(x)$$
- Besides: $r_2(x) \; w_2(x) \; r_1(x) \; w_1(x)$ is serial but not in TS

CSR, VSR, 2PL and TS



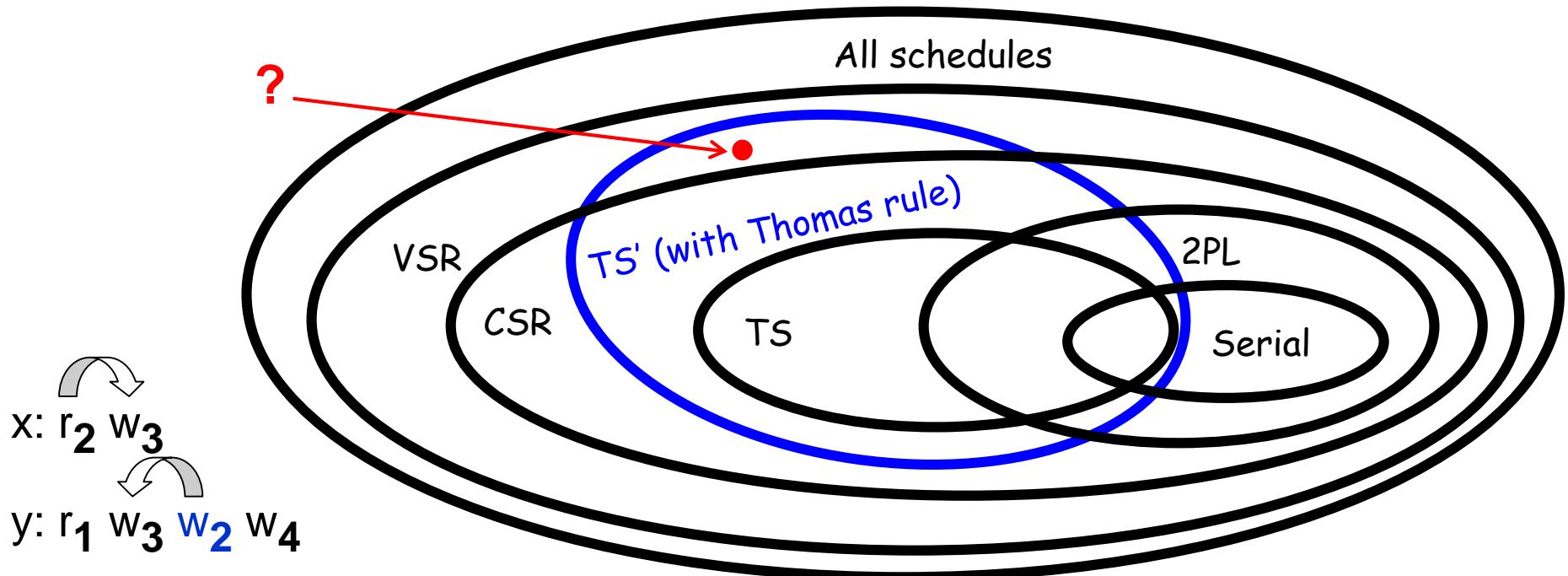
2PL vs. TS

- In *2PL* transactions can be *waiting*. In *TS* they are *killed* and restarted
- The *serialization order* with *2PL* is imposed by conflicts, while in *TS* it is imposed by the timestamps
- The necessity of *waiting for commit* of transactions causes long delays in strict *2PL*
- *2PL* can cause *deadlocks* (but also *TS* if care is not taken)
- Restarting a transaction costs more than waiting: *2PL wins!*

TS-based concurrency control: a variant (Thomas Rule)

- The scheduler has two counters: $\text{RTM}(x)$ and $\text{WTM}(x)$ for each object
- The scheduler receives read/write requests tagged with timestamps:
 - *read*(x, ts):
 - If $ts < \text{WTM}(x)$ the request is **rejected** and the transaction is killed
 - Else, the request is **granted** and $\text{RTM}(x)$ is set to $\max(\text{RTM}(x), ts)$
 - *write*(x, ts):
 - If $ts < \text{RTM}(x)$ the request is **rejected** and the transaction is killed
 - Else, if $ts < \text{WTM}(x)$ then our write is "obsolete": it can be **skipped**
 - Else, the request is **granted** and $\text{WTM}(x)$ is set to ts
- Does this modification affect the taxonomy of the serialization classes?

TS' (TS with Thomas Rule)



$r_1(y) \ r_2(x) \ w_3(y) \ w_2(y) \ w_3(x) \ w_4(y)$

Multiversion Concurrency Control

- Idea: writes generate new copies, reads access the “right” copy
- Writes generate new copies, each one with a new WTM. Each object x always has $N > 1$ active copies with $\text{WTM}_N(x)$. There is a unique global RTM(x)
- Old copies are discarded when there are no transactions that need these values

Multiversion Concurrency Control

- Mechanism:
 - $read(x, ts)$ is always accepted. A copy x_k is selected for reading such that:
 - If $ts > \text{WTM}_N(x)$, then $k = N$
 - Else take k such that $\text{WTM}_k(x) < ts < \text{WTM}_{k+1}(x)$
 - $write(x, ts)$:
 - If $ts < \text{RTM}(x)$ the request is rejected
 - Else a new version is created (N is incremented) with $\text{WTM}_N(x) = ts$

Example

Assume $RTM(x) = 7$

$N=1$ $WTM(x_1) = 4$

Request	Response	New Value
<i>read(x,6)</i>	ok	
<i>read(x,8)</i>	ok	$RTM(x) = 8$
<i>read(x,9)</i>	ok	$RTM(x) = 9$
<i>write(x,8)</i>	no	t_8 killed
<i>write(x,11)</i>	ok	$N=2$, $WTM(x_2) = 11$
<i>read(x,10)</i>	ok on 1	$RTM(x) = 10$
<i>read(x,12)</i>	ok on 2	$RTM(x) = 12$
<i>write(x,13)</i>	ok	$N=3$, $WTM(x_3) = 13$

Snapshot isolation

- The realization of multi-TS gives the opportunity to introduce into SQL another isolation level, **SNAPSHOT ISOLATION**
- In this level, no RTM is used on the objects, only WTM
- Every transaction reads the version consistent with its timestamp (**snapshot**), and defers writes to the end
- If a transaction notices that its writes damage writes occurred after the snapshot, it aborts
 - It is called an **optimistic** approach

Anomalies in Snapshot isolation

- Snapshot isolation does not guarantee **serializability**

T1: update Balls set Color=White where Color=Black

T2: update Balls set Color=Black where Color=White

- A serializable execution of T1 and T2 would produce at the end a configuration with balls that are either all white or all black
- An SI execution may just swap the colors

Advanced Databases

3

Reliability Control

Topics

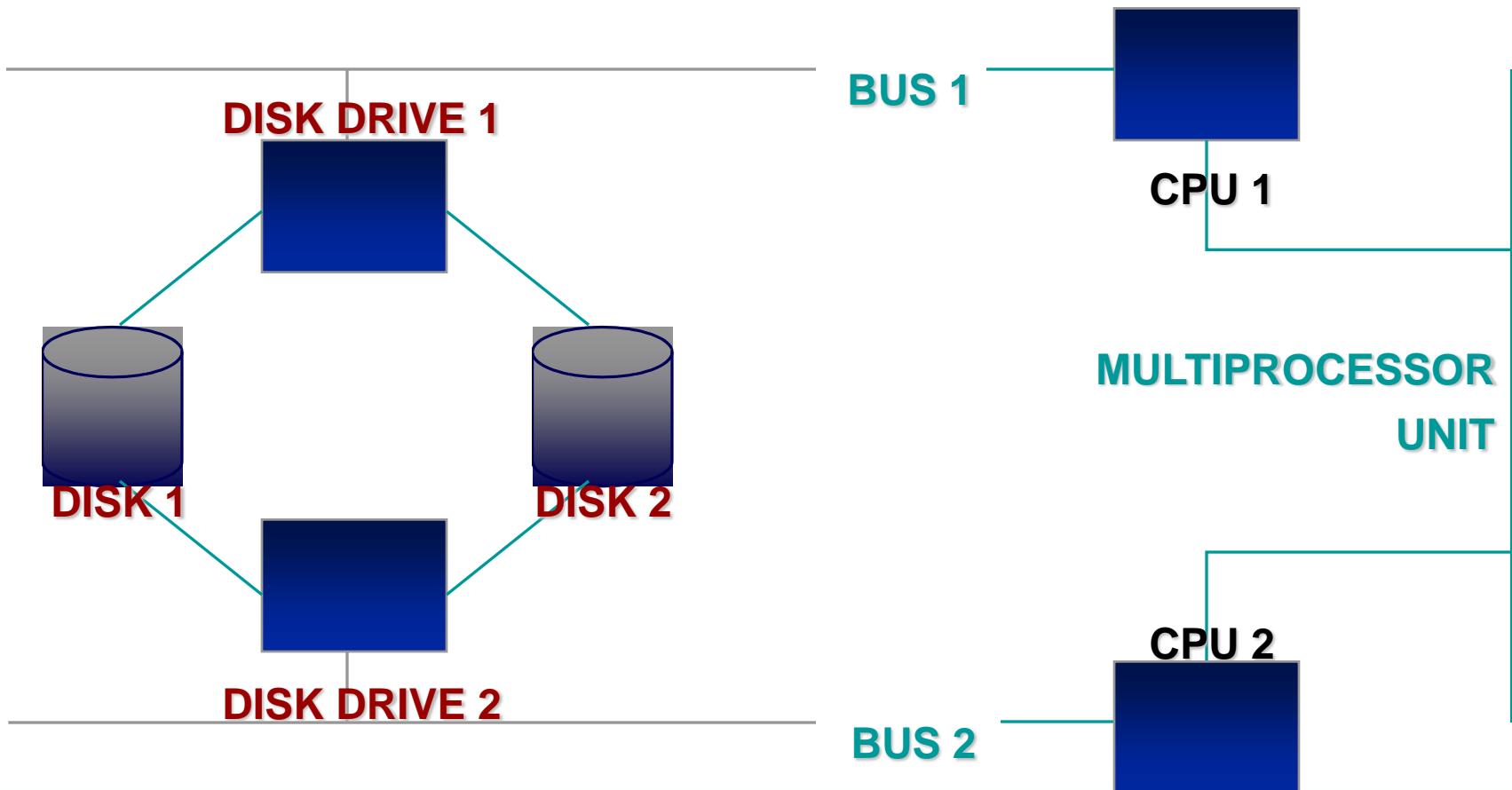
- Persistence of memory and backup
- Buffer management
- Reliable transaction management
- Log management
- Recovery after malfunctions

Persistence of Memories

- Main memory
 - Not persistent
- Mass memory
 - Persistent but can be damaged
- Stable memory
 - Cannot be damaged (it's an abstraction)

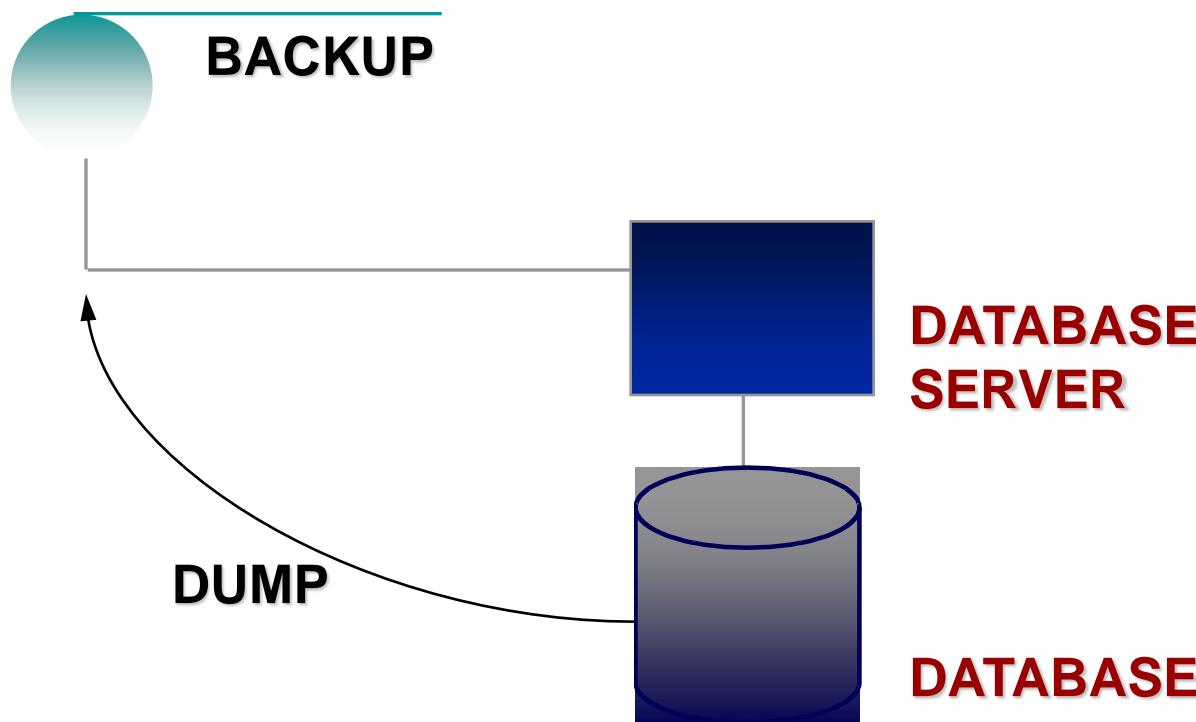
How to Guarantee Stable Memory

- On-line replication: mirroring of two disks

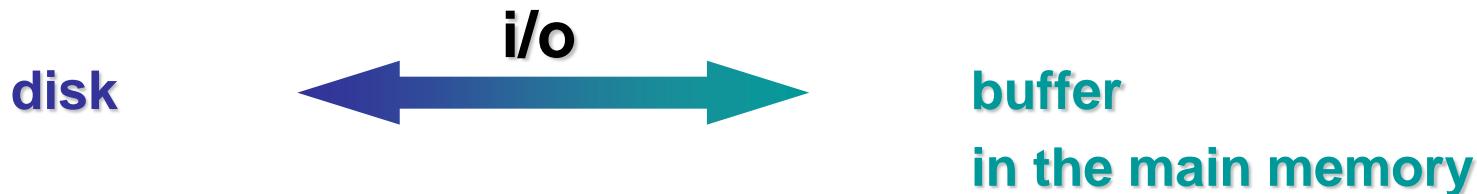


How to Guarantee Stable Memory

- Off-line replication: tape unit (backup)

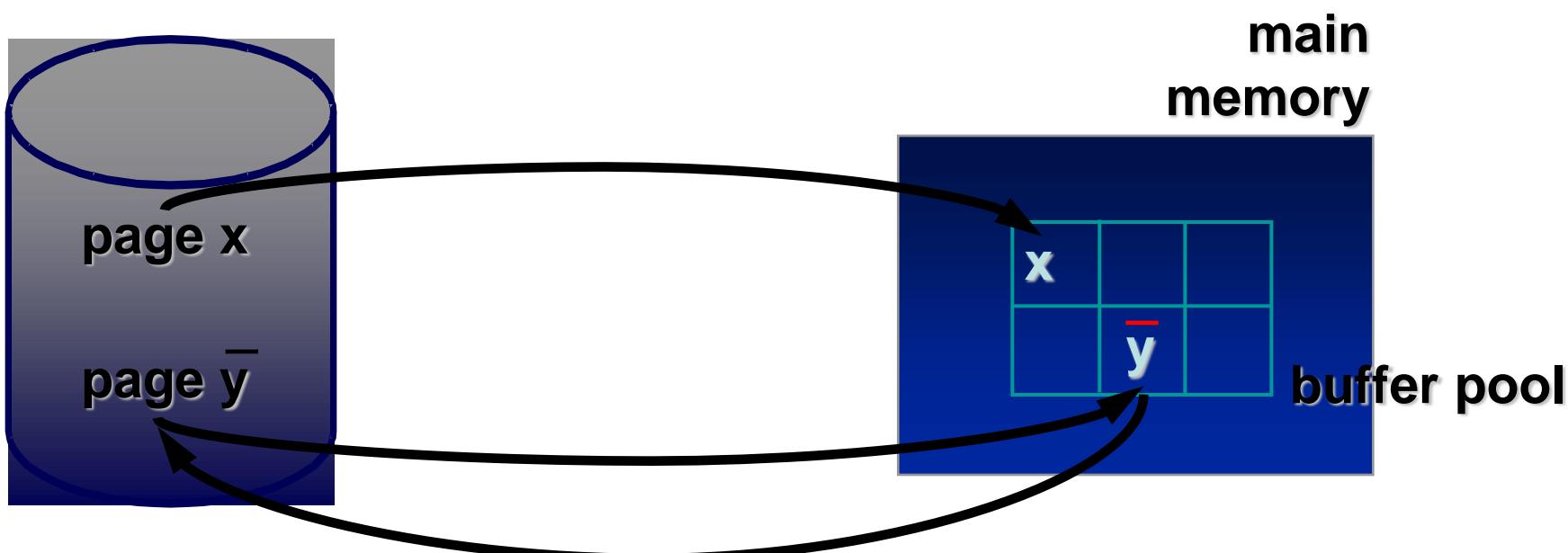


Main Memory Management



- Rationale
 - Reuse of data in the buffer
 - Deferred writing into the database

Use of Main Memory



Buffer Management

- Based on four primitives:
 - **fix**
 - Used to load a page into the buffer. After the operation, the page is allocated to a transaction
 - **use**
 - Use of a page in the buffer
 - **unfix**
 - De-allocation of a page
 - **force**
 - Synchronously transfers a page from the main memory to the database

Using the Buffer (Transactions)

- Follows the scheme

fix

repeat **use** until (end of transaction)

unfix

- Pages are written by the buffer asynchronously
- flush**
 - This primitive is controlled by the buffer manager
 - Asynchronously transfers a page from the main memory to the database

Executing a fix Primitive

- Searching for the target page
 - Selection of a free page
 - Otherwise, selection of a de-allocated page, which, if necessary, is copied onto the disk
 - Otherwise (if STEAL policy) a page is taken away from an active transaction. The page is copied onto the disk
 - Otherwise (if NO STEAL policy) the search fails
- Reading
 - If a target page exists, it is read from the database into the buffer in main memory

Buffer Management Policies

- STEAL (pages taken away from an active transaction)
- NO STEAL

- FORCE (pages written at commit-work)
- NO FORCE

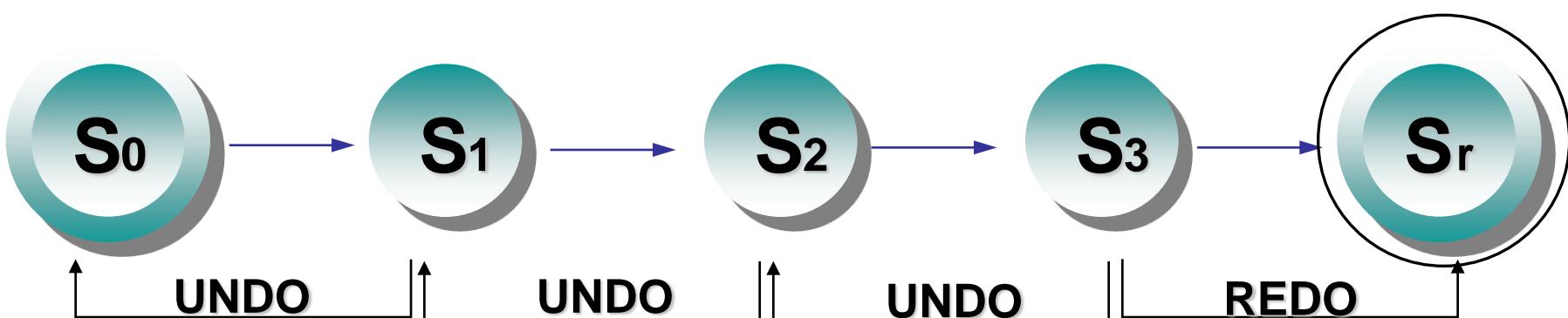
- Normally:
 - NO STEAL
 - NO FORCE

Buffer Management Policies

- PRE-FETCHING
 - anticipates reading of pages
 - especially useful in sequential reading
- PRE-FLUSHING
 - anticipates writing of de-allocated pages
 - useful for accelerating page **fix**

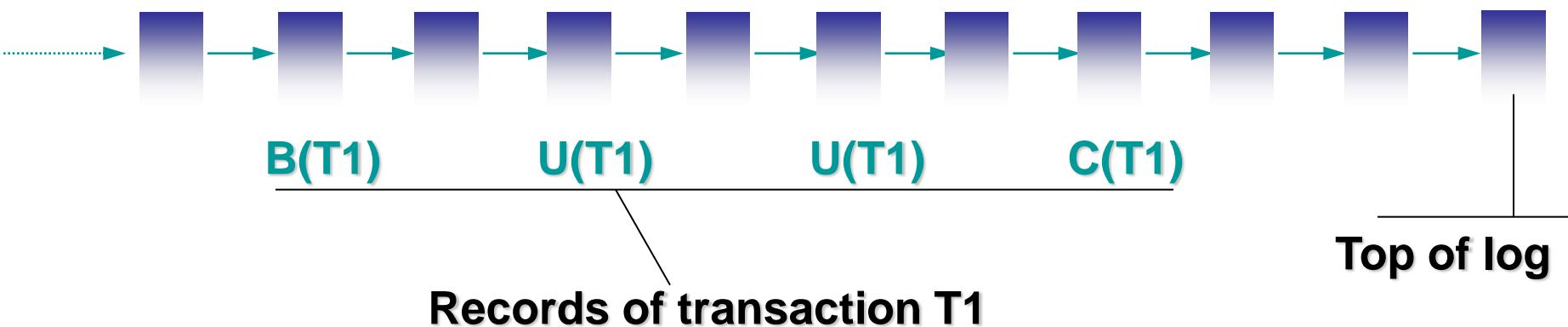
Reminder: Atomicity Requirements

- A transaction is an atomic transformation from an initial state into a final state
- Possible behaviors:
 - **commit work**: success
 - **rollback work or error before commit**: undo
 - Fault after **commit**: redo



Transaction Log

- Sequential file consisting of records that describe the actions carried out by the various transactions
- Written sequentially to the top block (top = current instant)



Main Function of the Log

- It records in the stable memory the actions carried out by the various transactions under the form of state transitions

If **UPDATE (U)**
transforms **O** from value **o₁** to value **o₂**

then the log records:

BEFORE-STATE (U) = o₁

AFTER-STATE (U) = o₂

Using the Log

- After **rollback-work** or failure
 - UNDO T1: O = O1
- After failure after **commit**
 - REDO T1: O = O2
- **Idempotency** of UNDO and REDO:
 $\text{UNDO}(T) = \text{UNDO}(\text{UNDO}(T))$
 $\text{REDO}(T) = \text{REDO}(\text{REDO}(T))$

Types of Log Records

- Records relevant to transactional commands:
 - `begin`
 - `commit`
 - `abort`
- Records relevant to operations
 - `insert`
 - `delete`
 - `update`
- Records relevant to recovery actions
 - `dump`
 - `checkpoint`

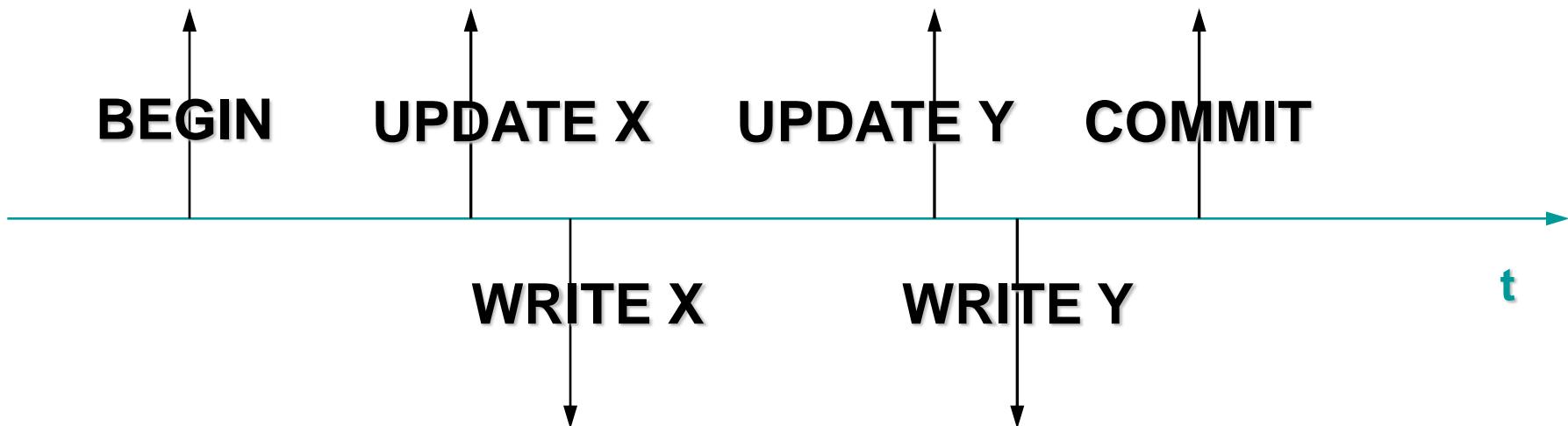
Types of Log Records

- Records relevant to transactional commands:
 - **B(T), C(T), A(T)**
- Records relevant to operations
 - **I(T,O,AS), D(T,O,BS), U(T,O,BS,AS)**
- Records relevant to recovery actions
 - **DUMP, CKPT(T1,T2,...,Tn)**
- Record fields:
 - **T:** transaction identifier
 - **O:** object identifier
 - **BS, AS:** before state, after state

Transactional Rules

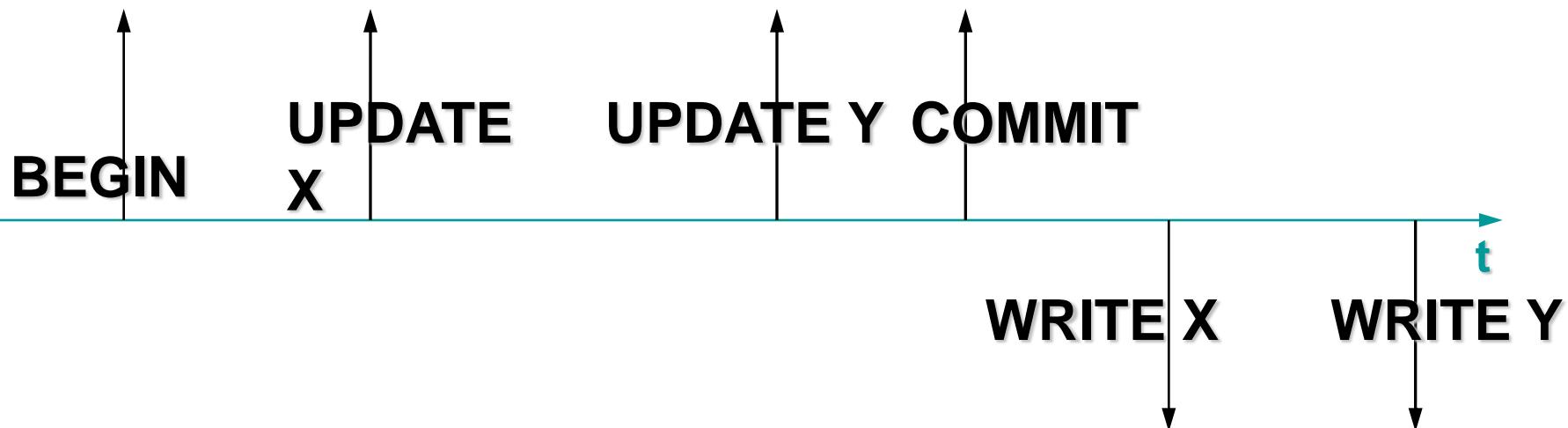
- Write-Ahead-Log
 - Before-state parts of the log records must be written in the log before carrying out the corresponding operation on the database
 - Actions can be undone
- Commit Rule
 - After-state parts of the log records must be written in the log before carrying out the commit
 - Actions can be redone

Writing onto Log and Database



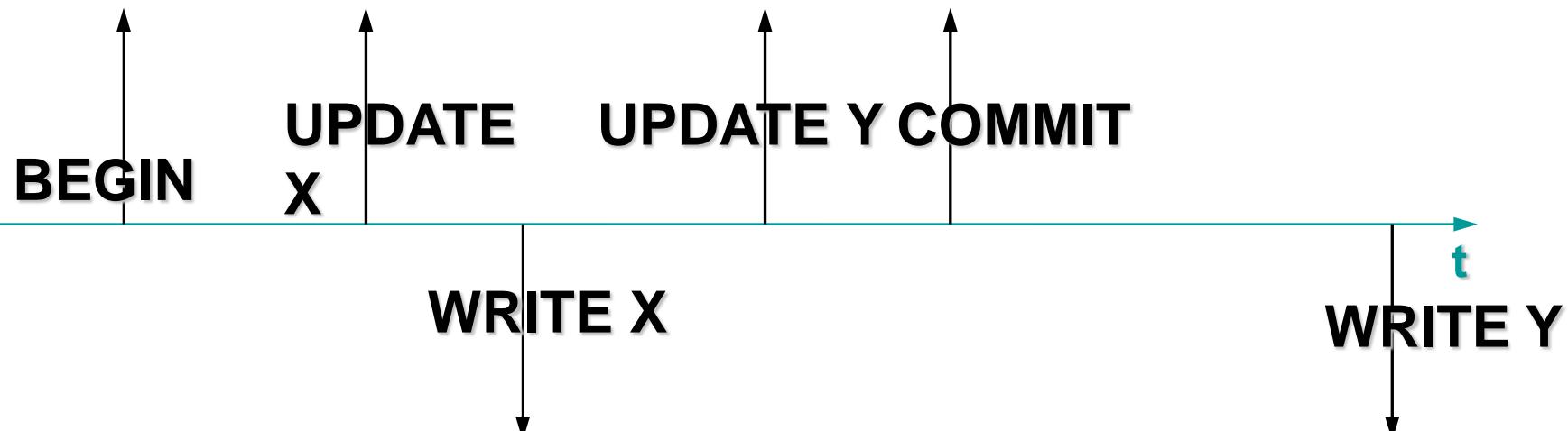
- Writing onto the database before commit
 - Requires writing in order to abort

Writing onto Log and Database



- Writing onto the database after commit
 - Does not require writing in order to abort

Writing onto Log and Database

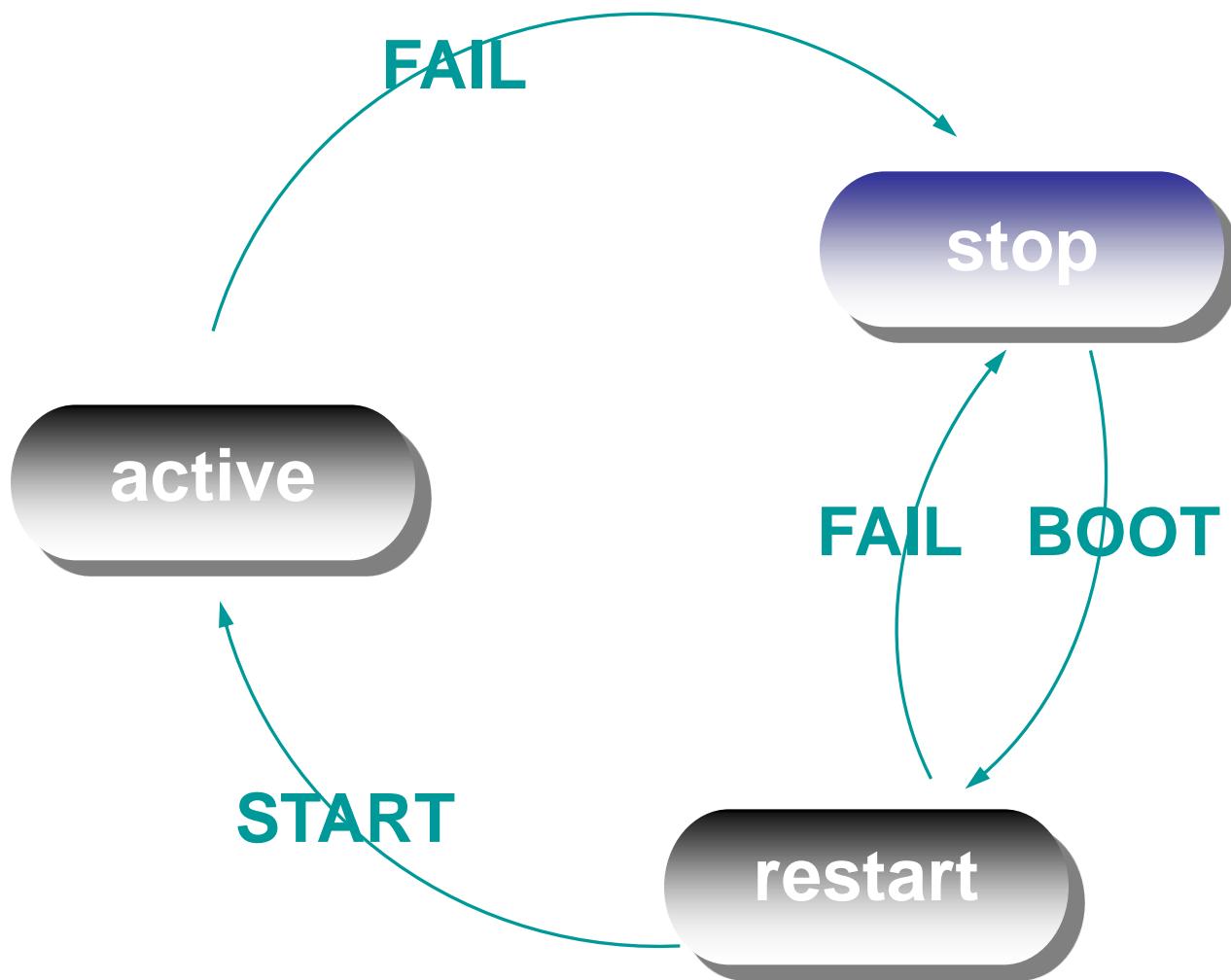


- Writing onto the database in an arbitrary moment
 - Allows optimizing buffer management

In Case of Failure

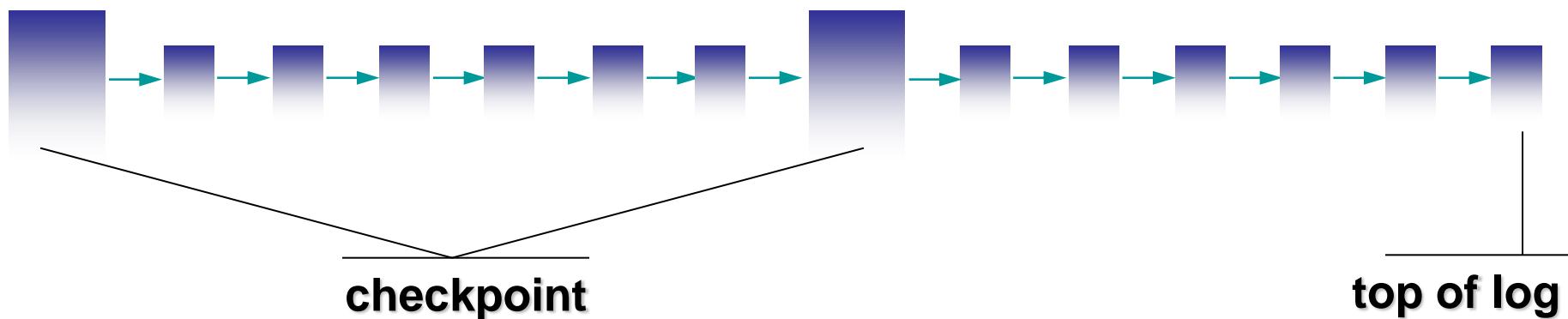
- Soft failure
 - Loss of the contents of the main memory
 - Requires **warm restart**
- Hard failure
 - Failure of secondary memory devices
 - Requires **cold restart**

Fail-stop Failure Model



Checkpoint

- “Consistent” time point
(in which all transactions write their data from the buffer to the disk)
- All active transactions are recorded



Checkpoint

- Operation used to “sum things up”, by simplifying the subsequent restore operations
 - Aim: to record which transactions are active at a given moment (and, dually, to confirm that the others either did not start or have finished)
- Parallel (extreme):
 - Closing the balance at the end of the year
 - Example: since November 25 no new “operation” request is accepted and all previously initiated operations must be concluded before new ones can be accepted

Checkpoint

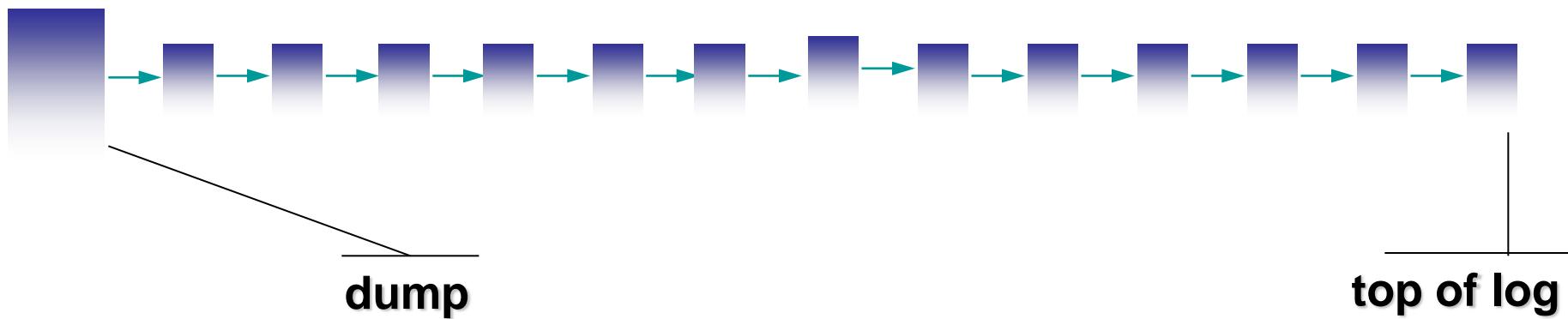
- Several possibilities – the simplest is as follows:
 1. Acceptance of commit requests is suspended.
 2. All dirty pages written by committed transactions are transferred to mass storage (via **force**).
 3. The identifiers of the transactions in progress are recorded on the log (via **force**); no new transaction can start while this recording takes place.

Then, acceptance of operations is resumed

- This way, we are sure that
 - For all committed transactions, the data are on mass storage
 - Transactions that are “half-way” are listed in the checkpoint

Dump

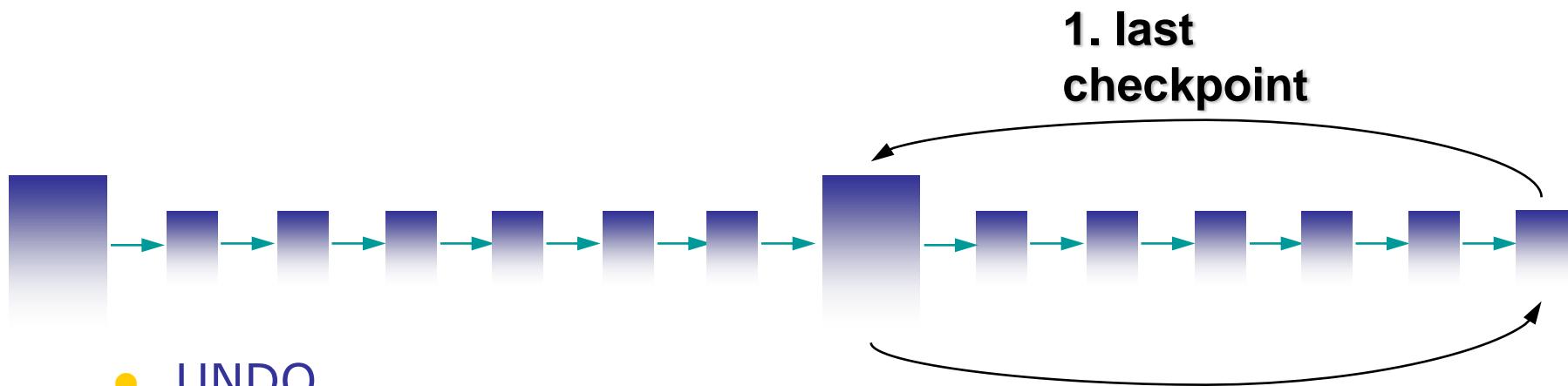
- Time point in which a complete copy of the database is created (typically during the night or the week-end)
- The presence of the dump is recorded



Warm Restart

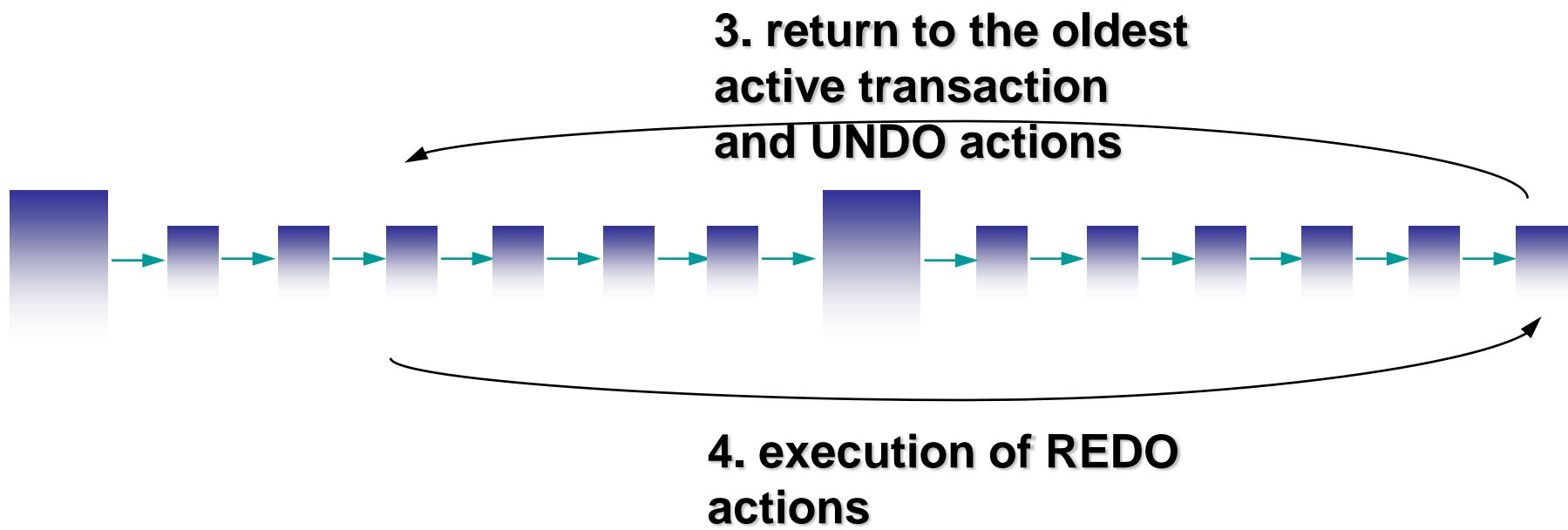
- Log records are read starting from the checkpoint
- Transactions are divided into:
 - UNDO set
 - REDO set
- UNDO and REDO actions are executed

Warm restart



- UNDO
 - Active transactions before commit
- REDO
 - Active transactions after commit

Warm Restart



Example of Warm Restart

- B(T1)
 - B(T2)
 - U(T1,O1,B1,A1)
 - I(T1,O2,A2)
 - U(T2,O3,B3,A3)
 - B(T3)
 - U(T3,O4,B4,A4)
 - D(T3,O5,B5)
 - CKPT(T1,T2,T3)
 - C(T2)
 - B(T4)
 - U(T4,O6,B6,A6)
 - A(T4)
 - failure
-
- UNDO=(T1,T2,T3)
REDO=()
 - UNDO=(T1,T3,T4)
REDO=(T2)

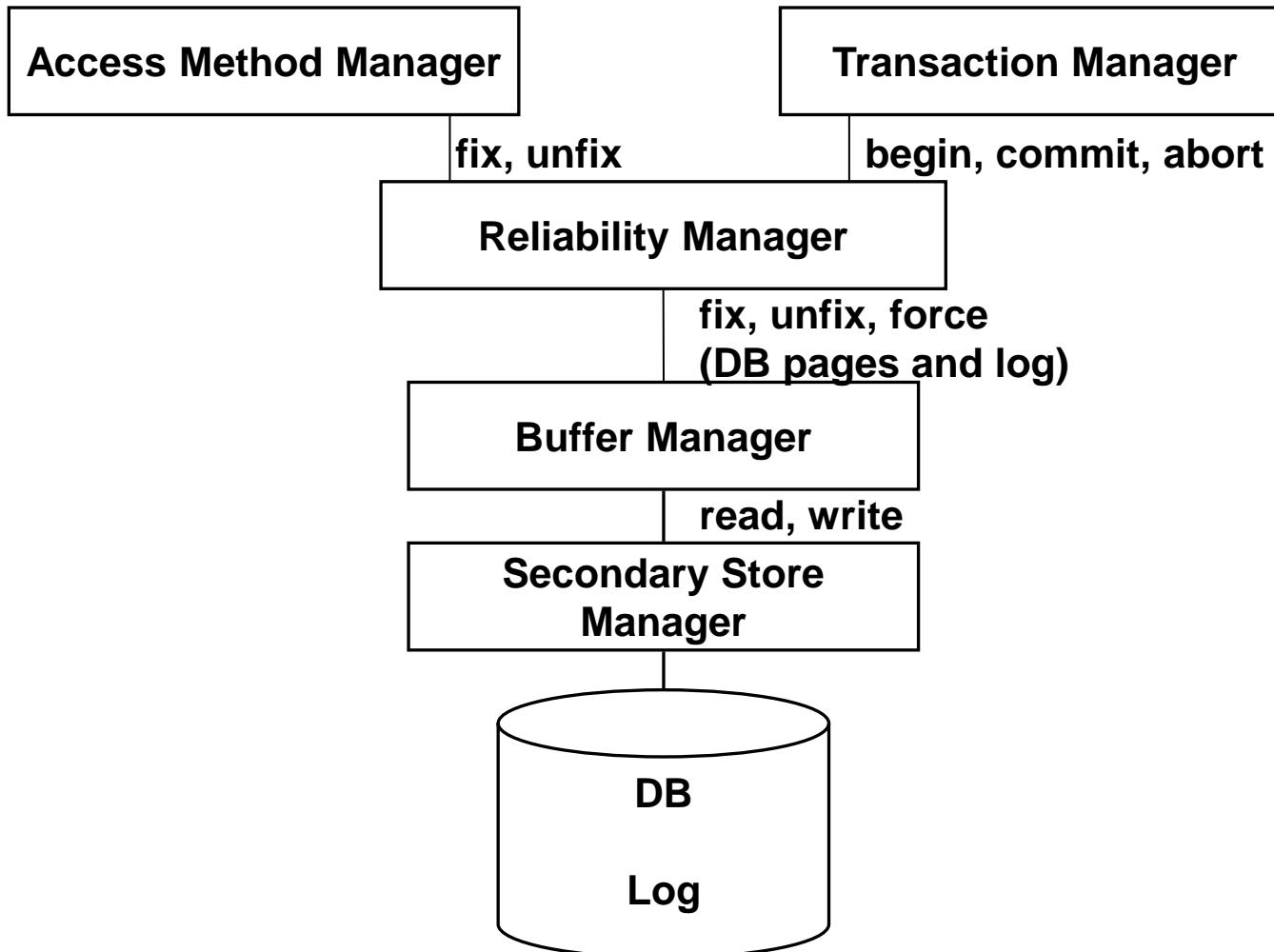
Example of Warm Restart

- B(T1) UNDO=(T1,T3,T4)
 - B(T2) REDO=(T2)
 - U(T1,O1,B1,A1)
 - I(T1,O2,A2)
 - U(T2,O3,B3,A3)
 - B(T3)
 - U(T3,O4,B4,A4)
 - D(T3,O5,B5)
 - CKPT(T1,T2,T3)
 - C(T2)
 - B(T4)
 - U(T4,O6,B6,A6)
 - A(T4)
 - failure
- O1 = B1
- DELETE(O2)
- O3 = A3
- O4 = B4
- O5 = B5
- O6 = B6
- RESTART

Cold Restart

- Data are restored starting from the backup
- The operations recorded onto the log until the failure are executed
- A warm restart is executed

Architecture of the Reliability Manager



Advanced Databases

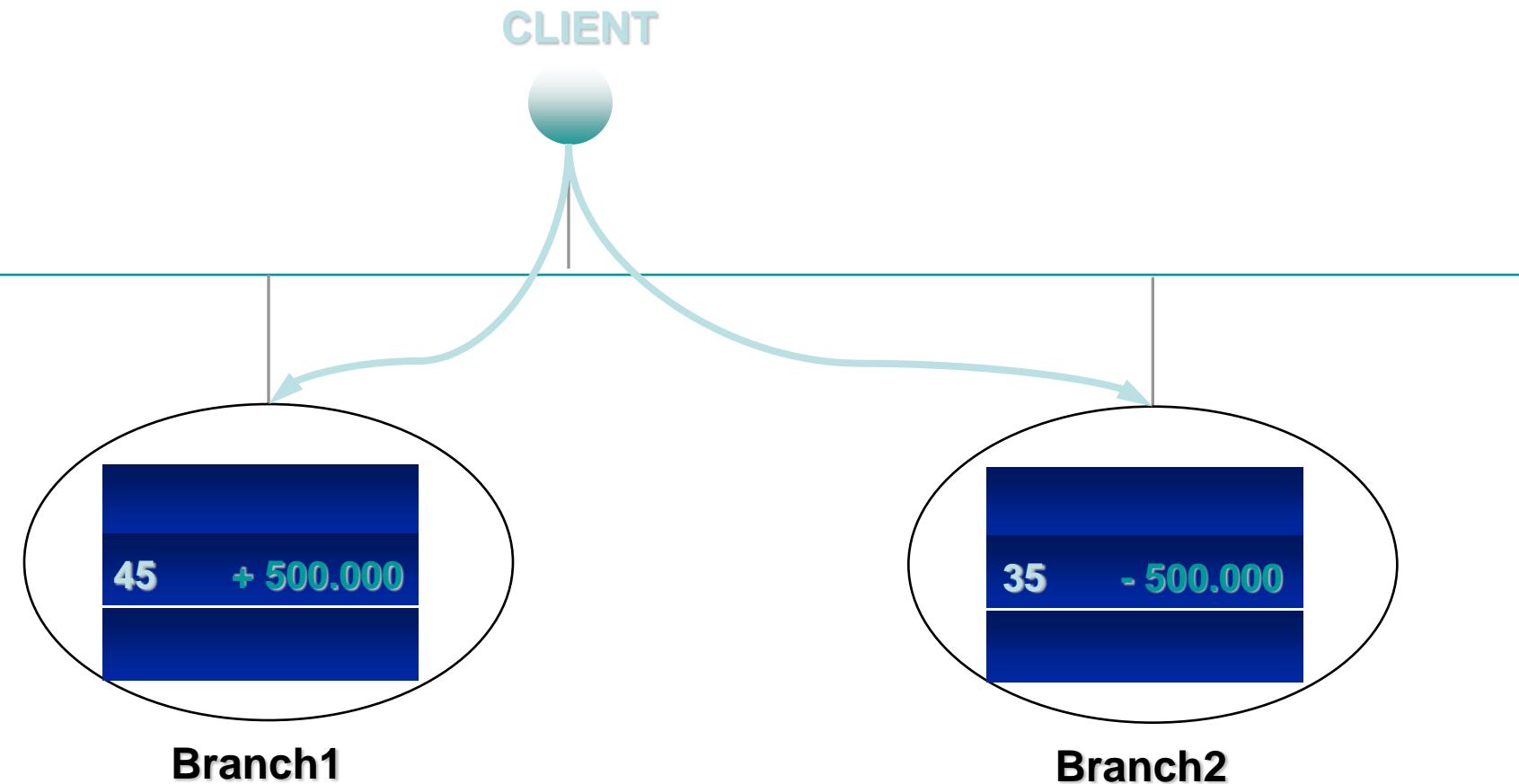
4

Commit Protocols

Distributed Transactions

```
begin transaction
    update Account1@1
        set Balance=Balance + 500.000
        where AccNum=45;
    update Account2@2
        set Balance=Balance - 500.000
        where AccNum=35;
commit work
end transaction
```

Distributed Transactions

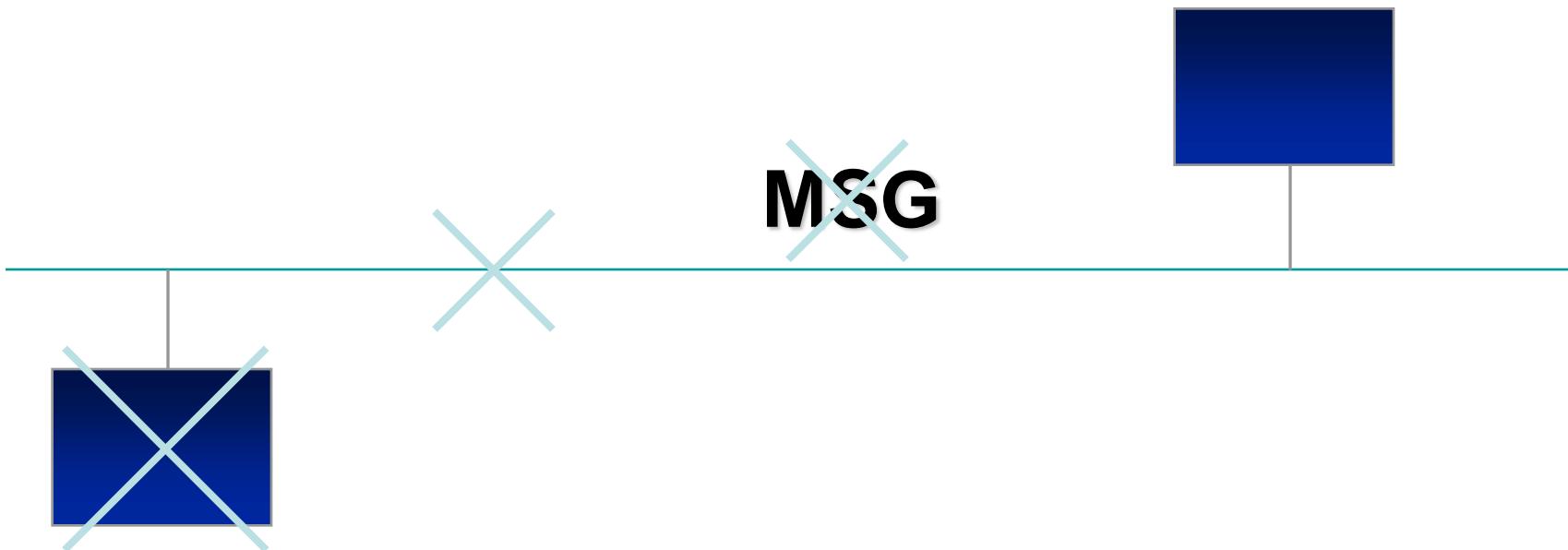


ACID Properties of Distributed Execution

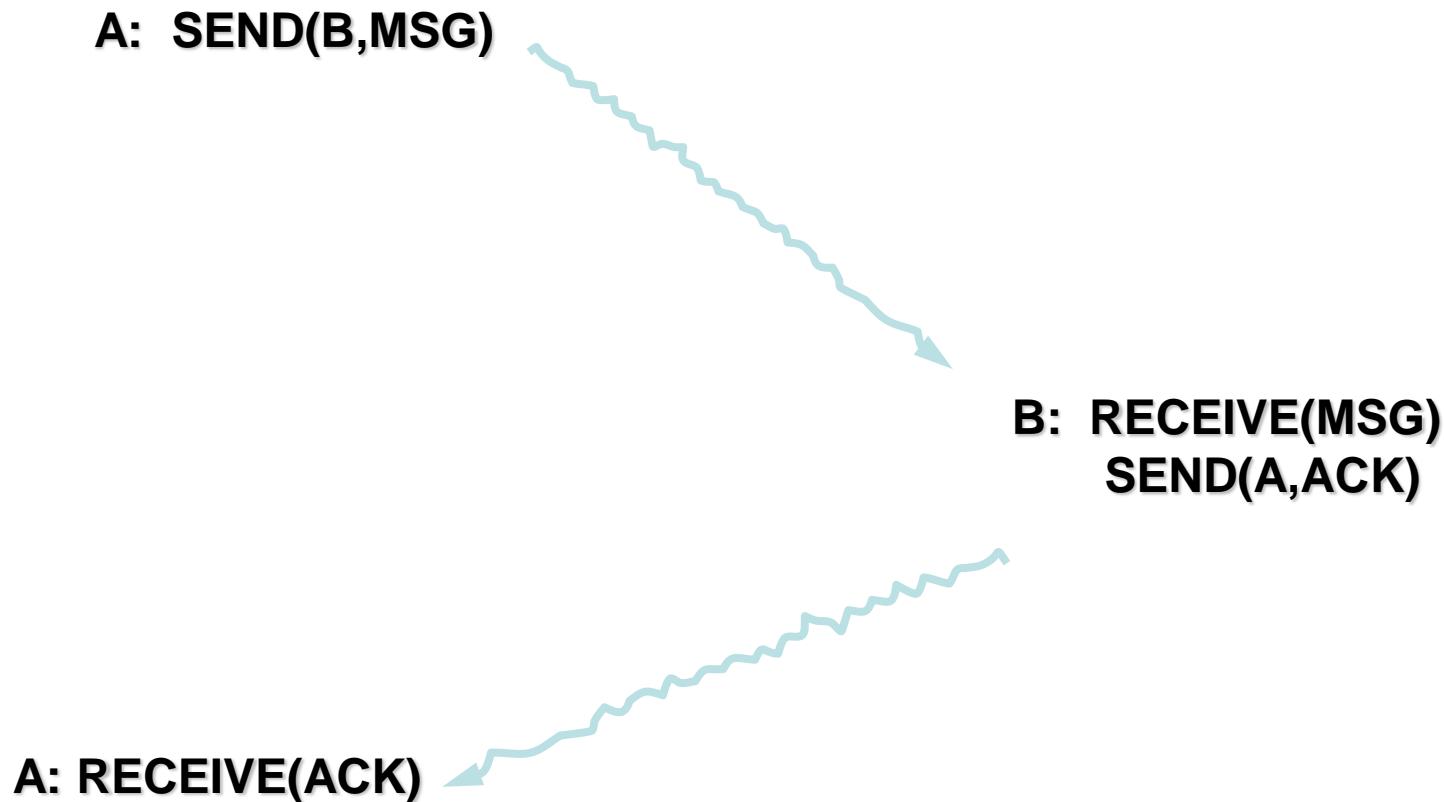
- Isolation
 - If each sub-transaction is two-phase, the transaction is globally serializable
- Durability
 - If each sub-transaction handles logs correctly, data are globally persistent
- Consistency
 - If each sub-transaction preserves local integrity, data are globally consistent
- Atomicity
 - It is the main problem of distributed transactions

Faults in a Distributed System

- Node failures
- Message losses
- Network partitioning



Distributed Protocols and Message Losses



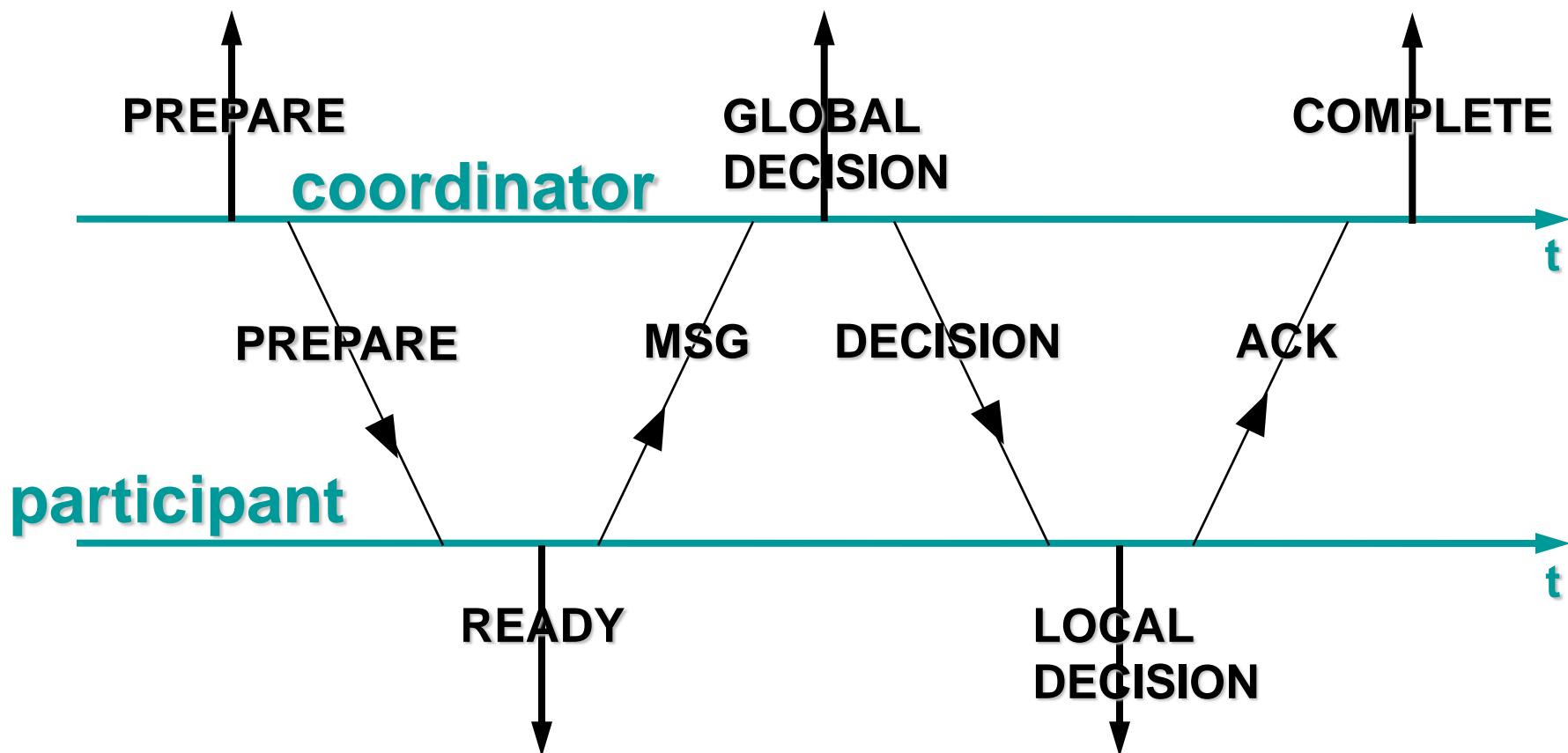
Two-phase Commit Protocol

- Protocol that guarantees atomicity of distributed sub-transactions
- Protagonists:
 - A coordinator ([Transaction Manager](#), TM)
 - Several participants ([Resource Manager](#), RM)
- Similar to a marriage
 - Phase one: the decision is declared
 - Phase two: the marriage is ratified

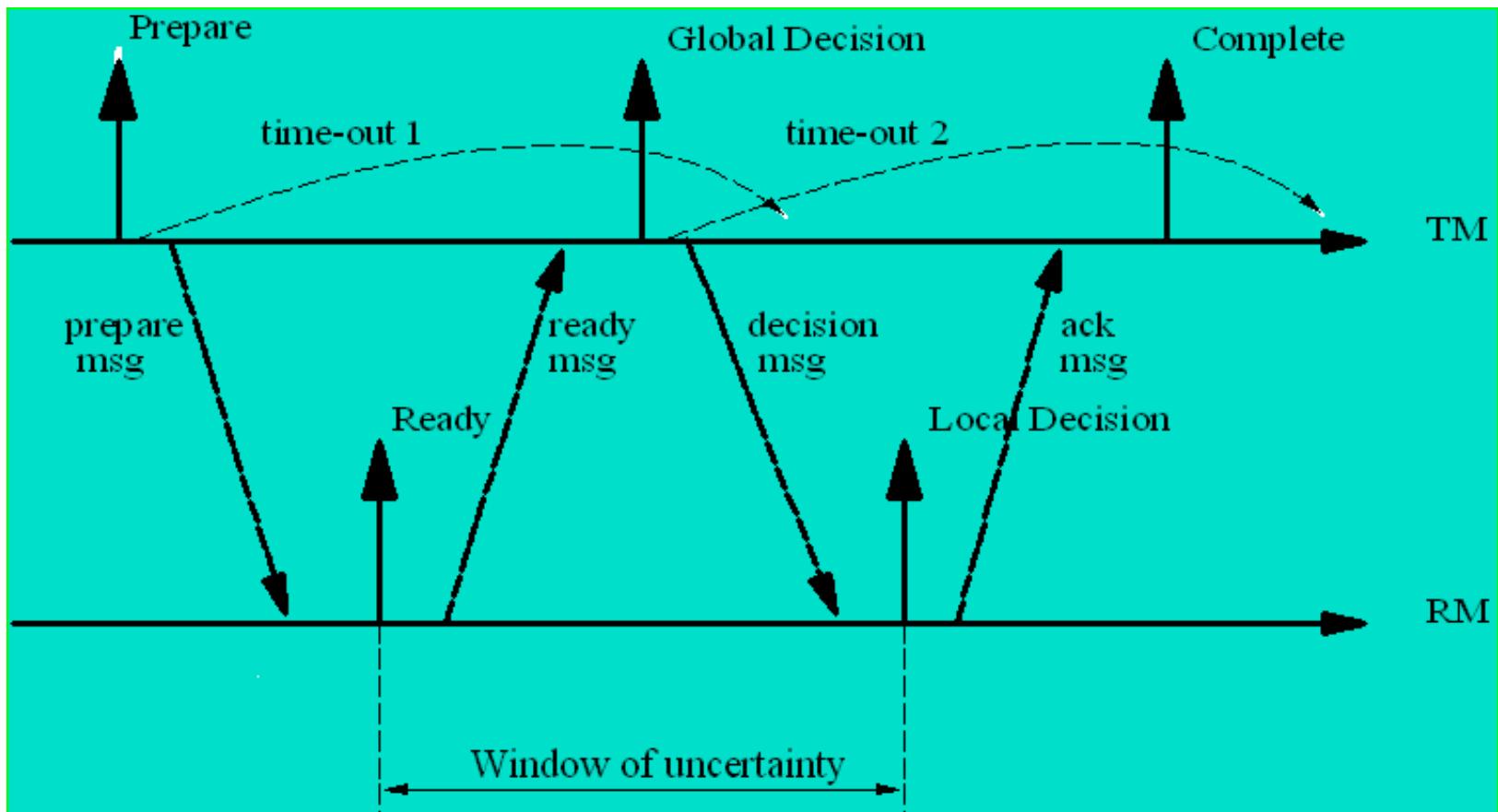
New Log Records

- In the coordinator's log
 - **prepare**: participants' identity
 - **global commit/abort**: decision
 - **complete**: end of protocol
- In the participant's log
 - **ready**: availability to participate to commit
 - **local commit/abort**: decision received

Diagram of the Two-phase Commit Protocol



Protocol with Time-out and Window of Uncertainty



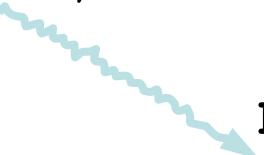
Phase 1

C: WRITE-LOG (PREPARE)
SET TIME-OUT
SEND (Pi , PREPARE)

Pi: RECEIVE (C , PREPARE)
IF OK THEN WRITE-LOG (READY)
READY=YES
ELSE READY=NO
SEND (C , READY)

Phase 2

```
C: RECEIVE (Ci ,MSG)
    IF TIME-OUT OR ONE (MSG) =NO
    THEN WRITE-LOG (GLOBAL-ABORT)
        DECISION=ABORT
    ELSE WRITE-LOG (GLOBAL-COMMIT)
        DECISION=COMMIT
    SEND (Pi ,DECISION)
```



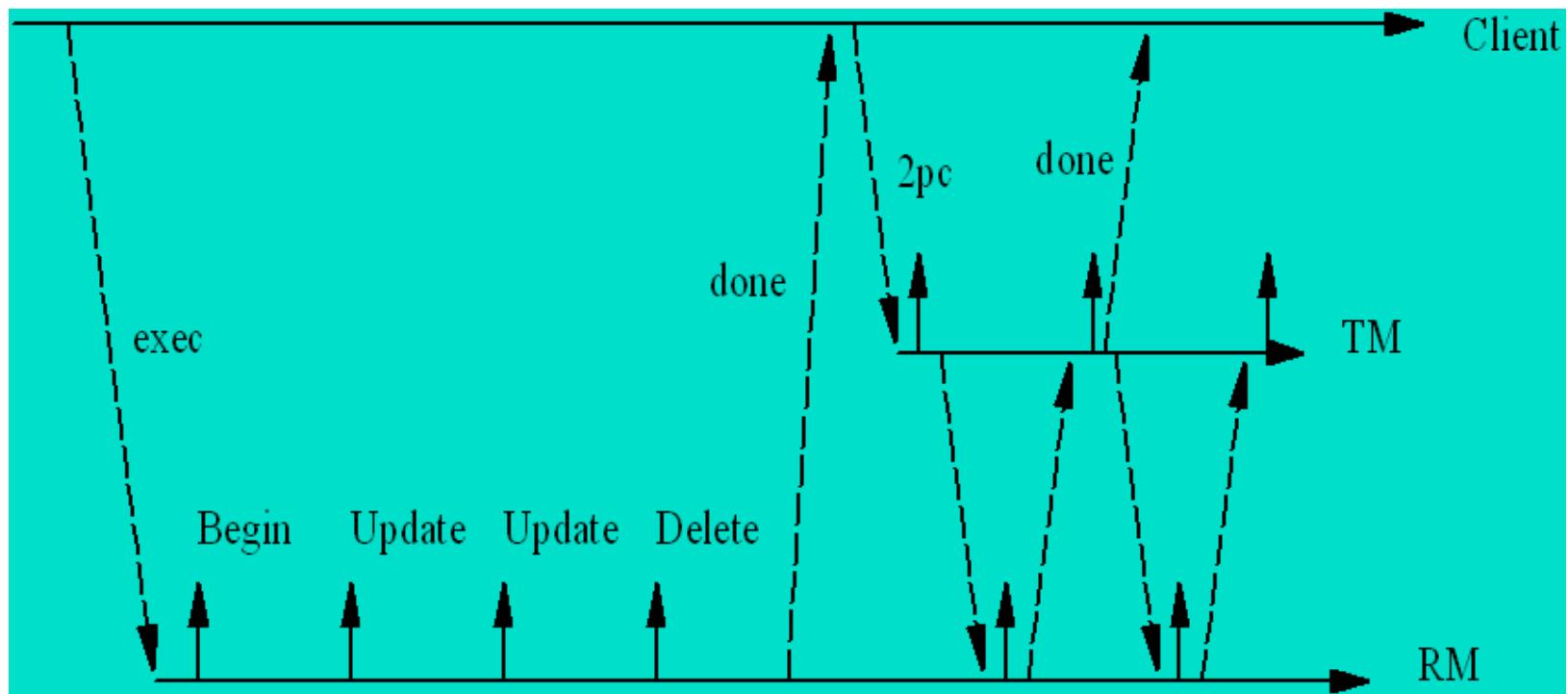
```
Pi: RECEIVE (C,DECISION)
    IF COMMIT THEN WRITE-LOG (COMMIT)
    ELSE WRITE-LOG (ABORT)
```

```
SEND (C,ACK)
```



```
C: RECEIVE (Pi ,ACK)
    WRITE-LOG (COMPLETE)
```

Protocol in the Context of a Complete Transaction



Complexity of the Protocol

- Must be able to handle all possible failures:
 - Failure of the coordinator
 - Failure of one or more participants
 - Message losses

Recovery of Participants

- Performed by the warm restart protocol. Depends on the last record written in the log:
 - If it is an action or **abort** record, the actions are *undone*; when it is a **commit**, the actions are *redone*; recovery doesn't depend on the protocol
 - If it is a **ready**, the failure has occurred during the two-phase commit. The participant is *in doubt* about the result of the transaction
- During the warm restart protocol, the identifier of the transactions in doubt are collected. For each of them the final transaction outcome must be requested to the TM (*remote recovery request*)

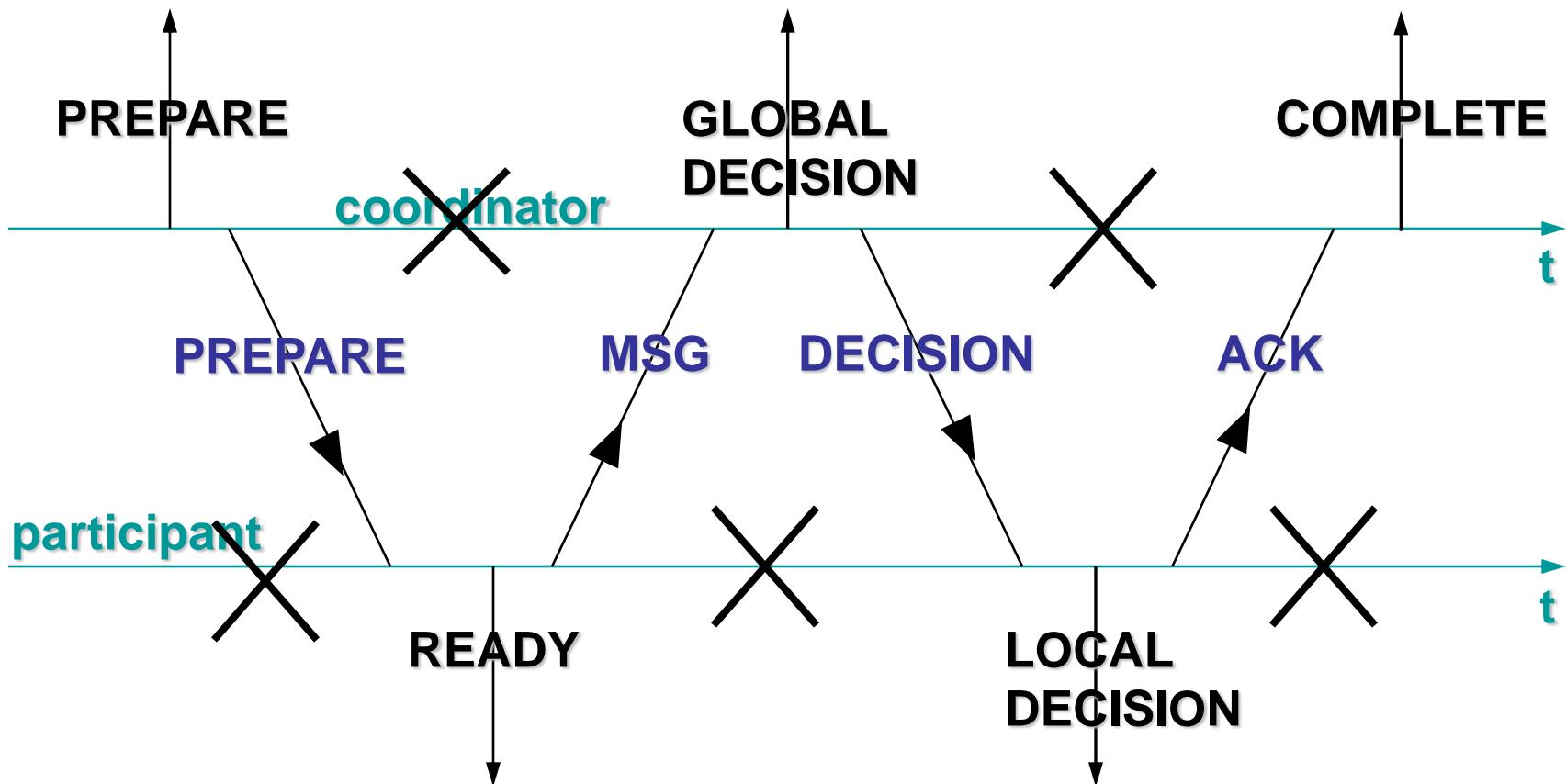
Recovery of the Coordinator

- When the last record in the log is a **prepare**, the failure of the TM might have placed some RMs in a blocked situation. Two recovery options:
 - Write **global abort** on the log, and then carry out the second phase of the two-phase commit protocol
 - Repeat the first phase, trying to reach a **global commit**
- When the last record is a “global decision”, some RMs may have been left in a blocked state. The TM must then repeat the second phase of the protocol

Message Loss and Network Partitioning

- The loss of a **prepare** or **ready** message are not distinguishable by the TM. In both cases, the TM reaches time-out and a **global abort** decision is made
- The loss of a **decision** or **ack** message are also indistinguishable. In both cases, the TM reaches time-out and the second phase is repeated
- A *network partitioning* does not cause further problems: **global commit** is reached only if the TM and all the RMs belong to the same partition

Recovery of the 2PC Protocol



Presumed Abort Protocol

- An optimization used by most DBMSs
 - If a TM receives a “remote recovery” request from an in-doubt RM and it does not know the outcome of that transaction, the TM returns a **global abort** decision as default
- As a consequence, if **prepare** and **global abort** are lost, the behavior is anyhow correct => it is not necessary to write them synchronously (force) onto the log
- Furthermore, the **complete** record can be omitted
- In conclusion the records to be forced are **ready**, **global commit** and **local commit**

Read-only Optimization

- When a participant is found to have carried out only read operations (no write operations)
 - It responds **read-only** to the **prepare** message and suspends the execution of the protocol
 - The TM ignores all read-only RMs in the second phase of the protocol

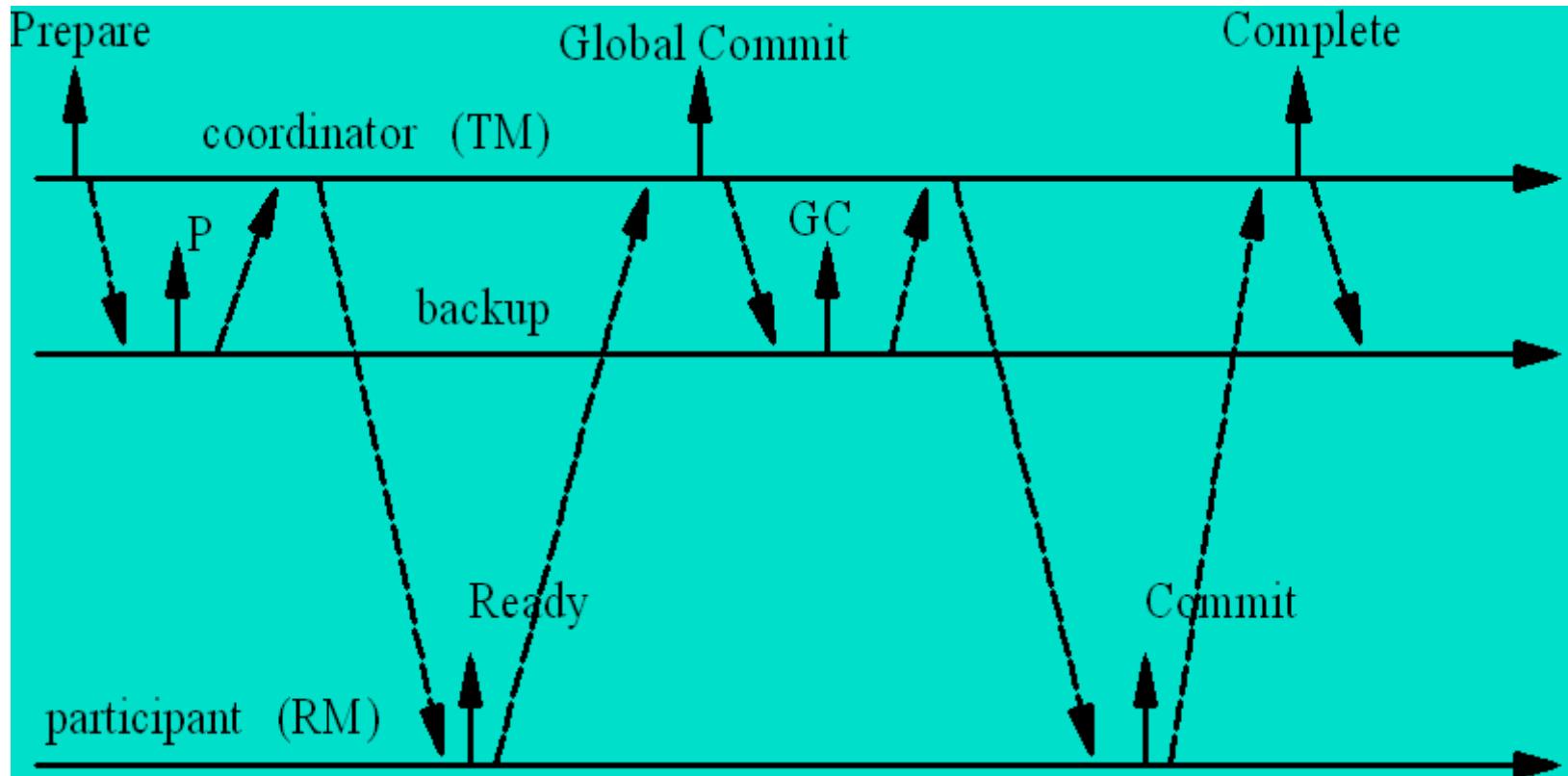
Blocking, Uncertainty, Recovery Protocols

- An RM in a “ready” state loses its autonomy and awaits the decision of the TM. A failure of the TM leaves the RM in an uncertain state. The resources acquired by using locks are blocked
- The interval between the writing on the RM’s log of the “ready” record and the writing of the **commit** or **abort** record is called the *window of uncertainty*. The protocol is designed to keep this interval to a minimum
- Recovery protocols are performed by the TM or RM after failures; they recover a correct final state which depends on the global decision of the TM

Four-phase Commit Protocol

- The TM process is replicated by a backup process, located on a different node.
 - The TM first informs the backup of its decisions and then communicates with the RMs
- The backup can replace the TM in case of failure
 - When a backup becomes TM, it first activates another backup
 - Then, it continues the execution of the commit protocol

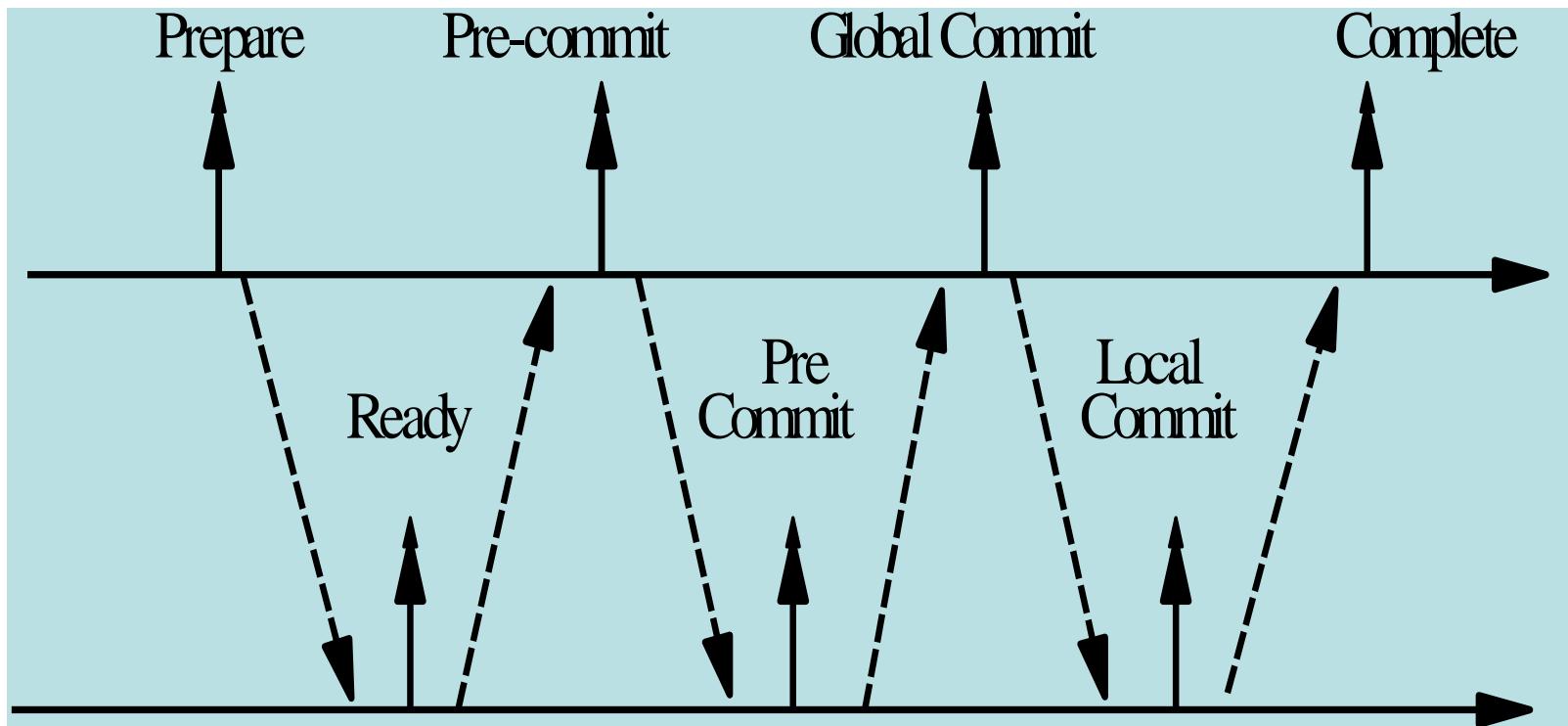
Diagram of the Four-phase Commit Protocol



Three-phase Commit Protocol

- Idea: thanks to a third phase, each participant can become a TM
- The “elected” participant looks at its log:
 - If the last record is **ready**, then it can impose a **global abort**
 - If the last record is **pre-commit**, it can impose a **global commit**
- Shortcomings:
 - It lengthens the window of uncertainty
 - It is not resilient to network partitioning

Diagram of the Three-phase Commit Protocol

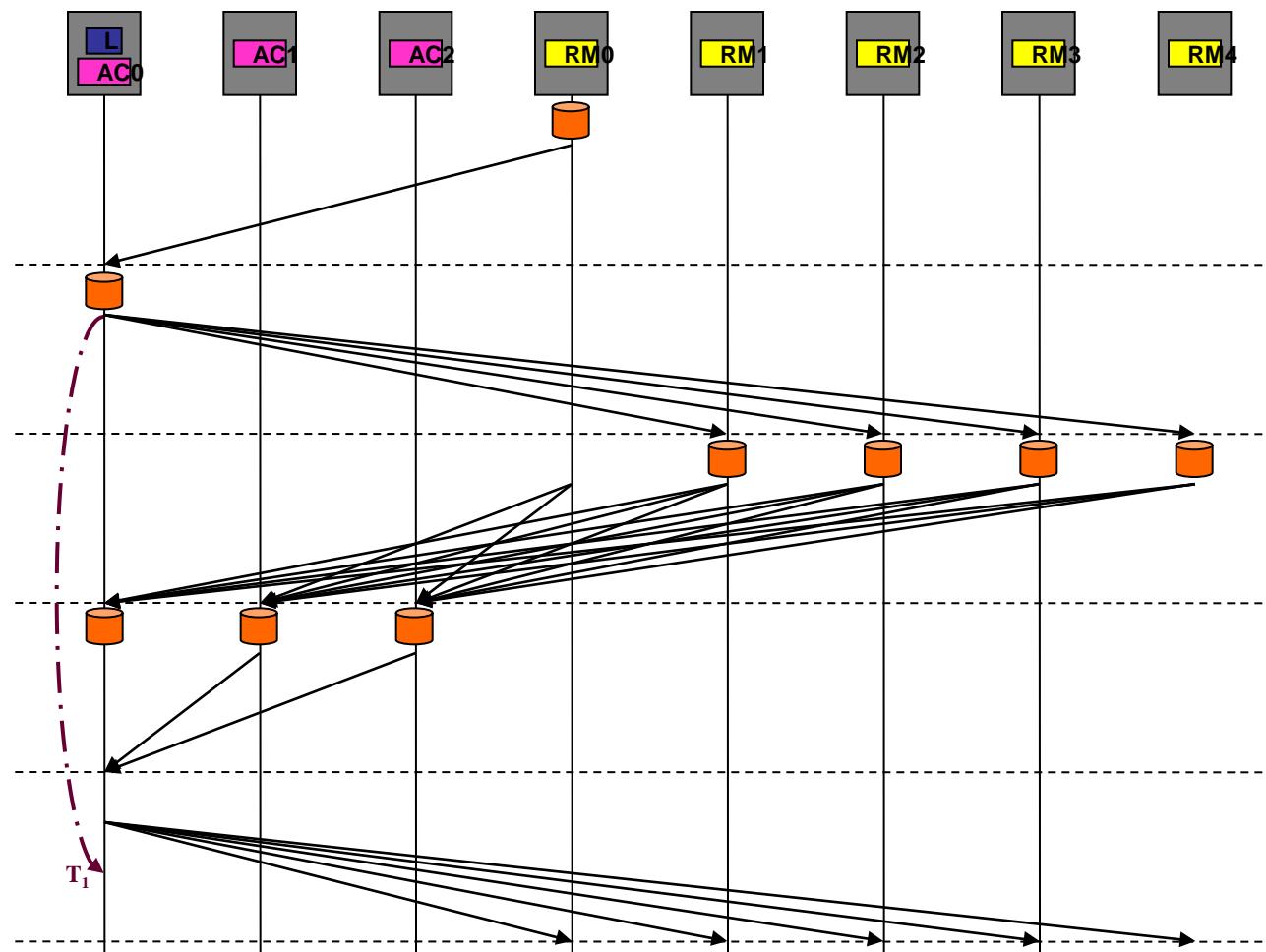


Paxos Commit Protocol

- Idea: combine a “consensus protocol” with a commit protocol so as to guarantee one decision even in the presence of partitions.
- Paxos consensus protocol: with a network that cannot fail more than F times, establish one “initiator” and $2F+1$ “acceptors”; the protocol guarantees consensus with a majority ($F+1$) of acceptors. Requires 3 phases.
- Paxos commit protocol (Gray-Lamport, ACM-TODS 2007) sets some RM (out of N) as acceptors and guarantees commit/rollback decision with $N(F+3) - 3$ messages.
- The idea: acceptors have to be co-ordinated to reach consensus, a lot of messages are saved by making RM and acceptors coincident.
- Two-phase commit is a special case of Paxos commit with $F=0$ (coordinator=acceptor).

Paxos Commit (base)

Writes on log



Standardization of the Protocol

- Standard X-Open Distributed Transaction Processing (DTP):
 - TM interface
 - Defines the services of the coordinator offered to a client in order to execute commit of heterogeneous participants
 - XA interface
 - Defines the services of passive participants that respond to calls from the coordinator (offered by several commercial DBMSs)

Features of X-Open DTP

- RMs are passive: they respond to remote procedure calls from the TMs
- Protocol: two-phase commit with optimizations (presumed abort and read-only)
- The protocol supports *heuristic decisions*: after a failure, an operator can impose a heuristic decision (abort or commit)
 - When heuristic decisions raise inconsistencies, the client processes are notified

TM Interface

- `tm_init` and `tm_exit` initiate and terminate the client-TM dialogue
- `tm_open` and `tm_term` open and close a session with the TM
- `tm_begin` begins a transaction
- `tm_commit` requests a global commit
- `tm_abort` requests a global abort

XA Interface

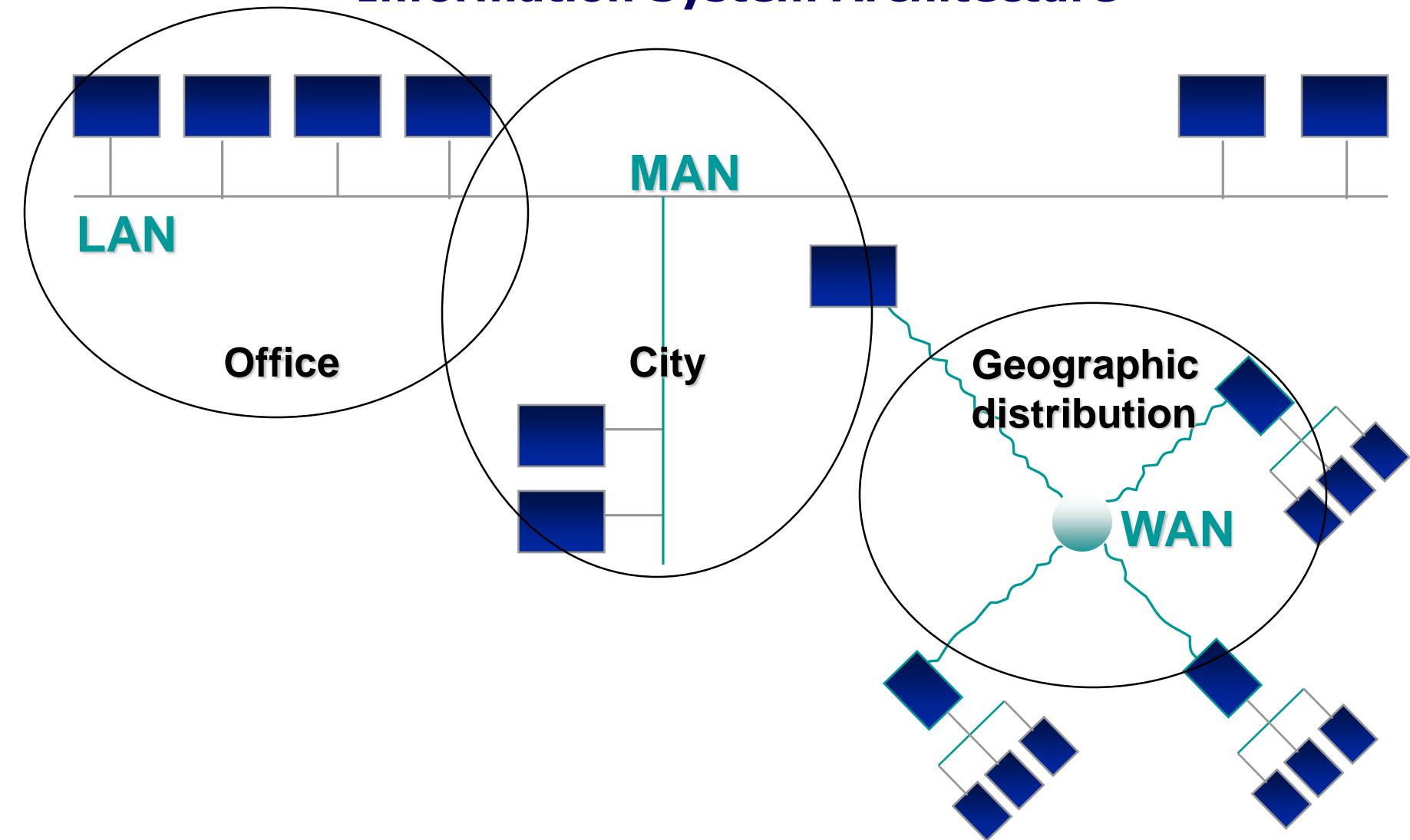
- **xa_open** and **xa_close** open and close a TM-RM dialog
- **xa_start** and **xa_end** activate and complete a new transaction
- **xa_precomm** requests that the RM carry out the first phase of the commit protocol
- **xa_commit** and **xa_abort** communicate the “global decision” to the RM
- **xa_recover** initiates an RM recovery; the RM responds to the request with three sets of transactions:
 - Transactions *in doubt*
 - Transactions decided by a *heuristic commit*
 - Transactions decided by a *heuristic abort*
- **xa_forget** allows an RM to forget transactions decided in a heuristic manner

Advanced Databases

5

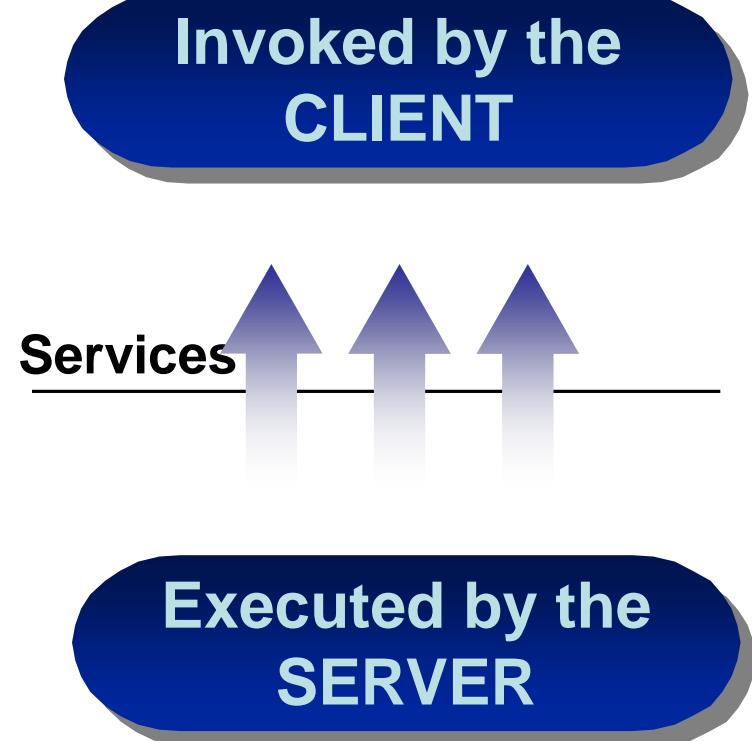
Distributed Databases

Information System Architecture



Client-Server Paradigm

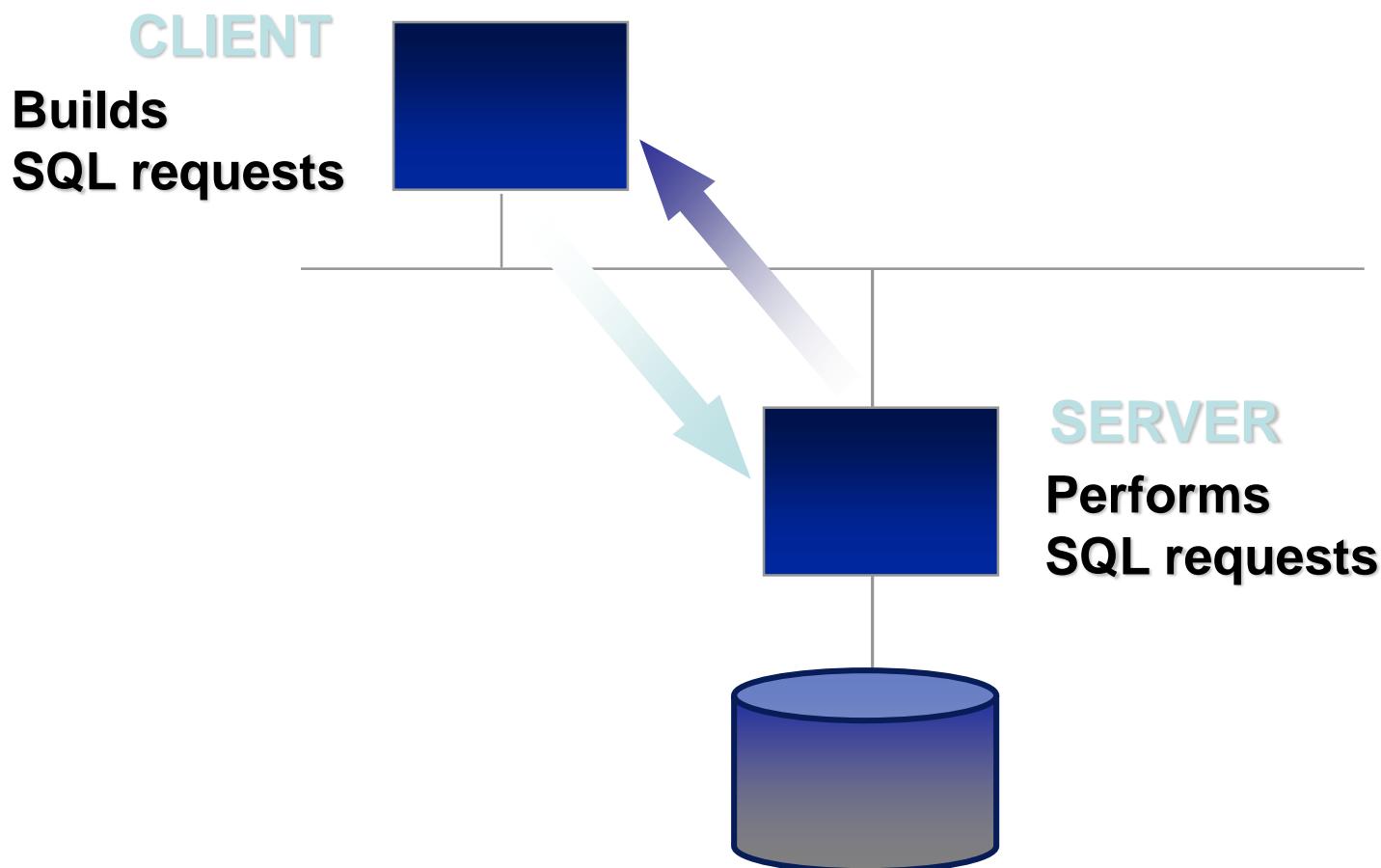
- Technique for software system design
- Two systems are involved:
 - **Client:** invokes services
 - **Server:** provides services
- The client process performs an active role, the server process is reactive
- A service interface is published by the server



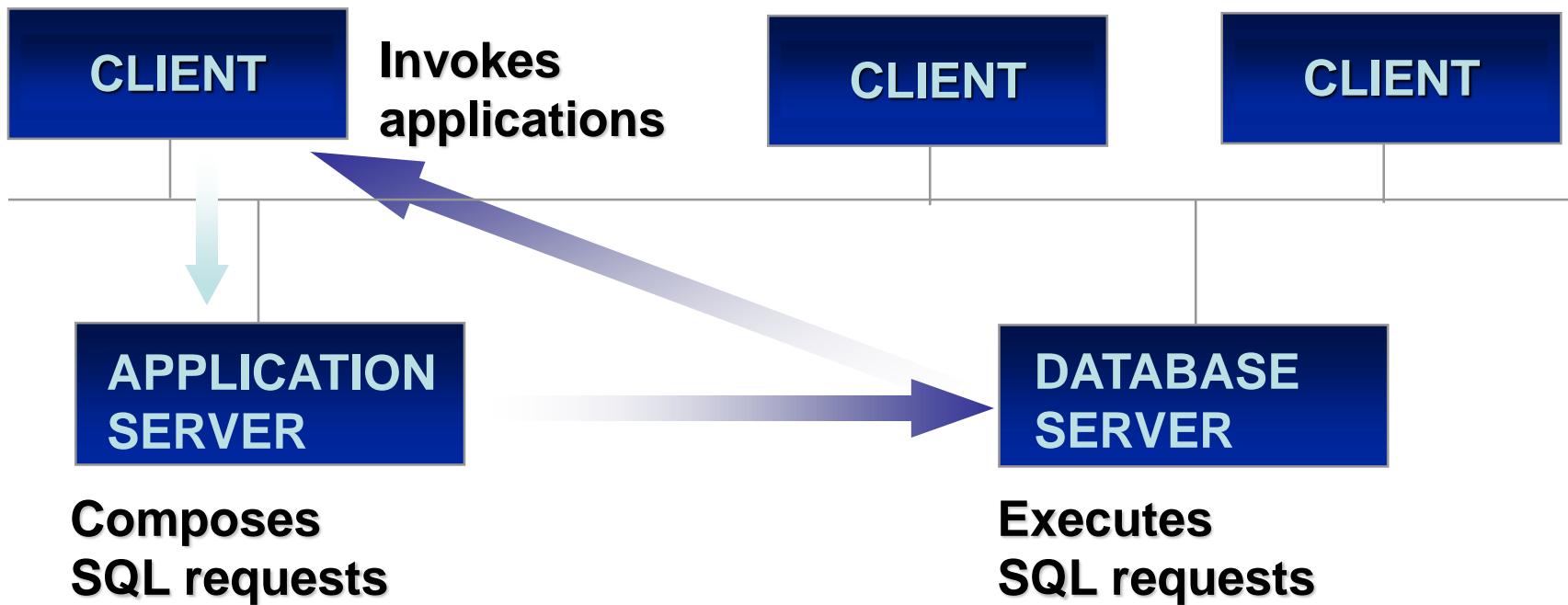
Client-Server in Information Systems

- Ideal functional separation
 - Client: presentation layer
 - Server: data management
- SQL: the perfect language for enacting separation
 - Client: formulates queries and shows results
 - Server: performs queries and calculates the results
 - Network: transfers activation commands (e.g., of SQL procedures) and query results

Traditional Client-Server architecture



Application Server Architecture



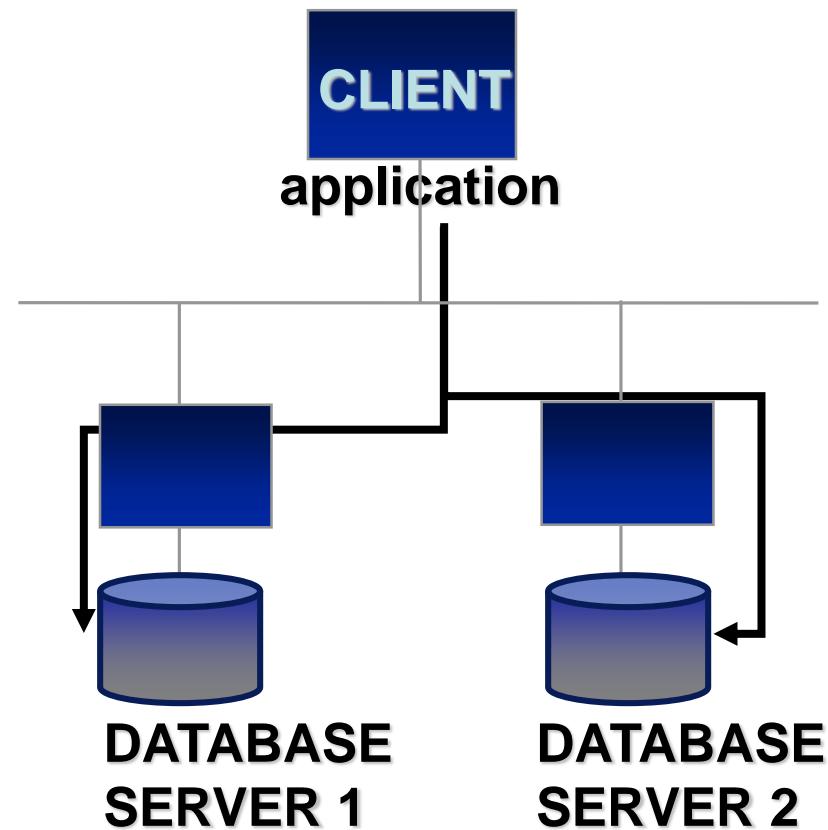
Processing of requests

- Input requests from clients are queued
- A dispatcher routes them to the appropriate software service (typically: multiple simple requests to the same “active server”, no need of loading into memory)
- Request is processed and an output message is put on the output queue, next routed to the requesting client
- Some systems can dynamically configure the number of software services depending on the input load
 - load balancing for given service classes
- Performance is achieved by parallelism and multi-processing (see next)

Data distribution

- Not only
 - Several databases
- But also
 - Applications that use data of different data sources

→Distributed Database

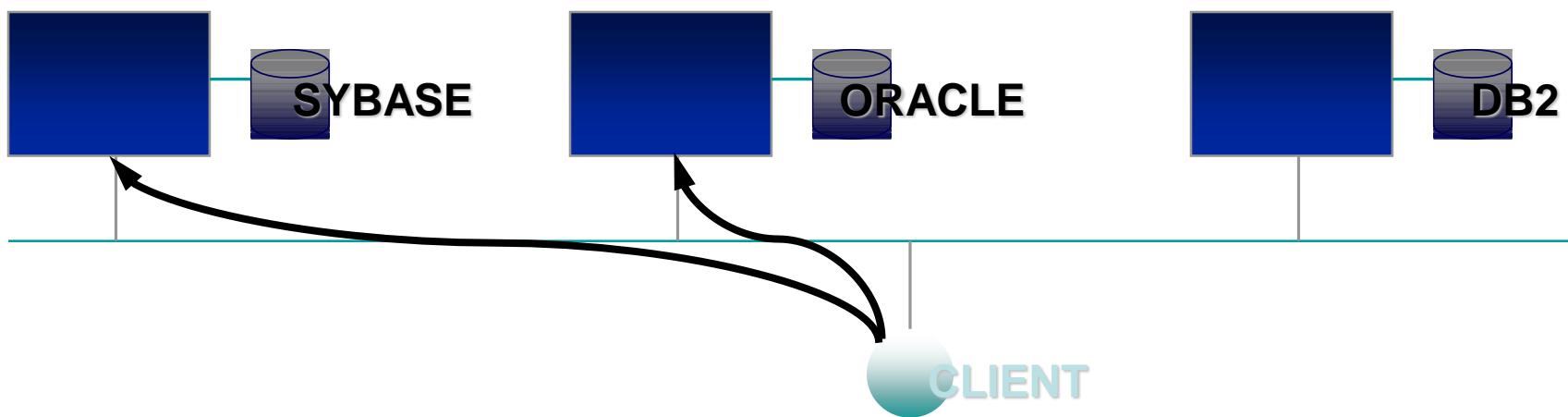


Motivations of Data Distribution

- Intrinsic distributed nature of the applications
- Evolution of computers:
 - Increased computation power
 - Reduced price
- Evolution of DBMS technology
- Interoperability standards

Distributed Database Types

- Classification based on the network:
 - LAN (Local Area Network)
 - WAN (Wide Area Network)
- Classification based on the involved databases:
 - Homogeneous system: all the same DBMS
 - Heterogeneous system: various DBMS



Typical Application Examples

	LAN	WAN
HOMOGENEOUS	Intra-division company management	Travel management and financial applications
HETEROGENEOUS	Inter-division company management	Integrated booking systems and inter- banking systems

Problems of Distributed Databases

- Independency and cooperation
- Transparency
- Efficiency
- Reliability

Independency and cooperation

- Independency needs are driven by:
 - Reaction to EDP divisions in the enterprise
 - Bringing knowledge and control local to the place where data are produced, used, and managed
 - Localizing most of the data flows and giving local autonomy over processing
- Cooperation needs are driven by:
 - Company-wide or business-to-business applications
 - Integrating several needs into a single, higher-level need (served by an application as a whole)

Data fragmentation

- Decomposition of the tables for allowing their distribution
- Properties:
 - Completeness: each data item of a table T must be present in one of its fragments T_i
 - Restorability: the content of a table T must be restorable from its fragments

Horizontal Fragmentation

- Fragments:
 - Sets of tuples
- Completeness:
 - availability of all the tuples
- Restorability:
 - UNION

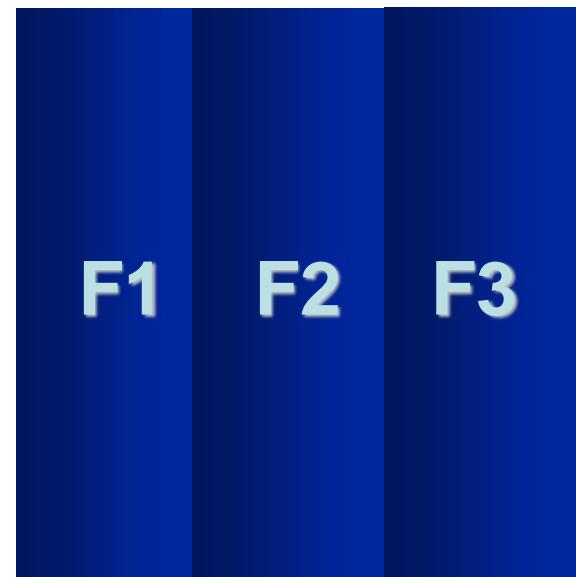
Fragment1

Fragment2

Fragment3

Vertical Fragmentation

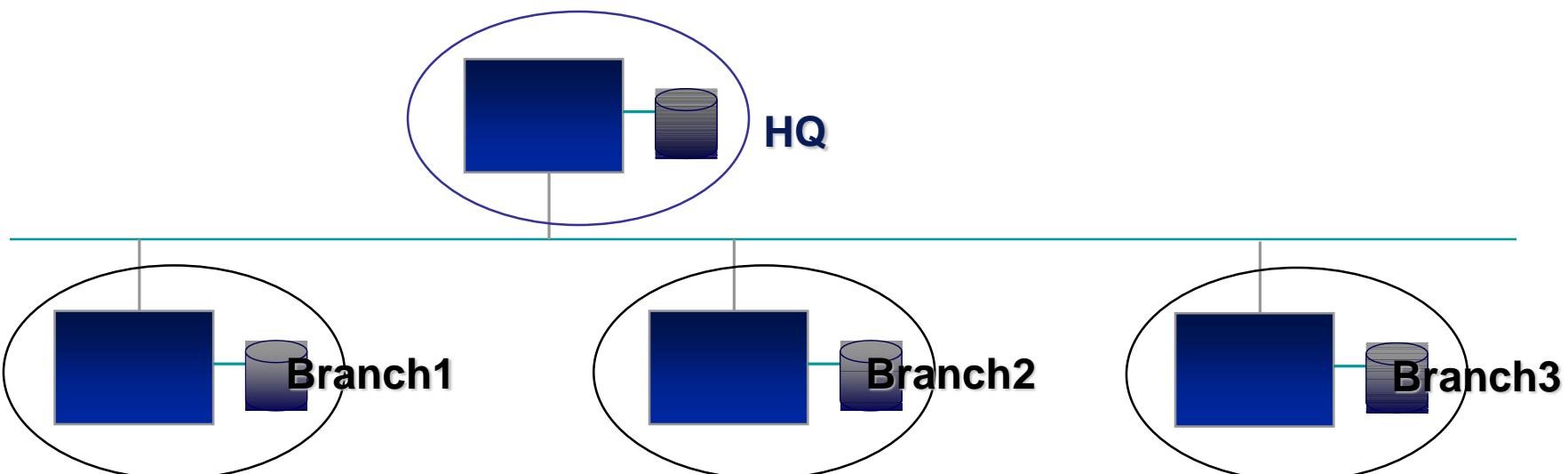
- Fragments:
 - Sets of attributes
- Completeness:
 - availability of all the attributes
- Restorability:
 - JOIN on the key



Example: bank accounts

ACCOUNT (Number, Name, Branch, Balance)

TRANSACTION (AccountNumber, Date, Incremental, Amount, Description)



(Primary) Horizontal Fragmentation

$$R_i = \sigma_{P_i} R$$

Example:

$$\text{Account1} = \sigma_{\text{Branch}=1} \text{ACCOUNT}$$

$$\text{Account2} = \sigma_{\text{Branch}=2} \text{ACCOUNT}$$

$$\text{Account3} = \sigma_{\text{Branch}=3} \text{ACCOUNT}$$

Derived Horizontal Fragmentation

$S_i = S \bowtie R_i$

Example:

Transaction1 = TRANSACTION \bowtie ACCOUNT1

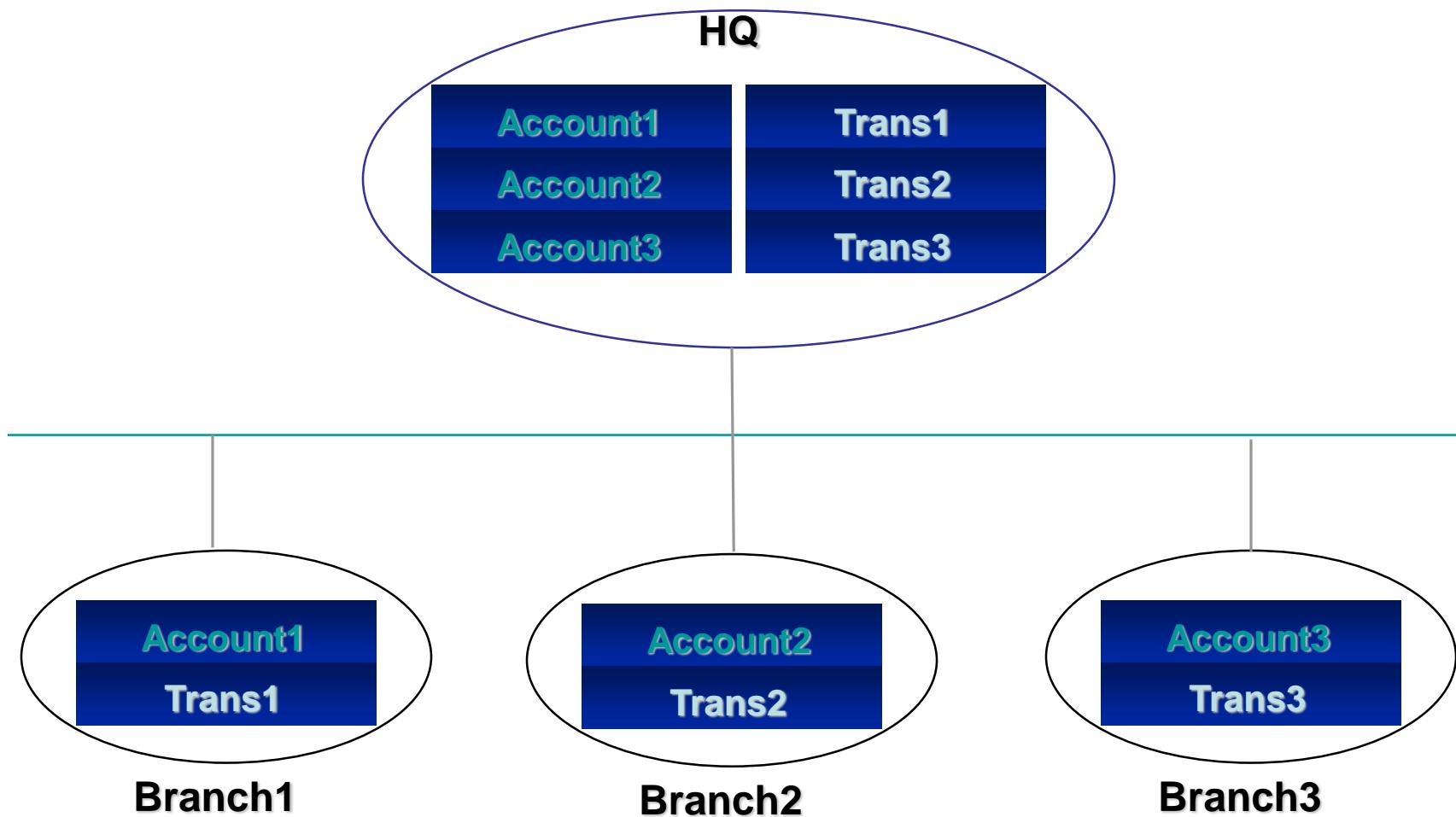
Transaction2 = TRANSACTION \bowtie ACCOUNT2

Transaction3 = TRANSACTION \bowtie ACCOUNT3

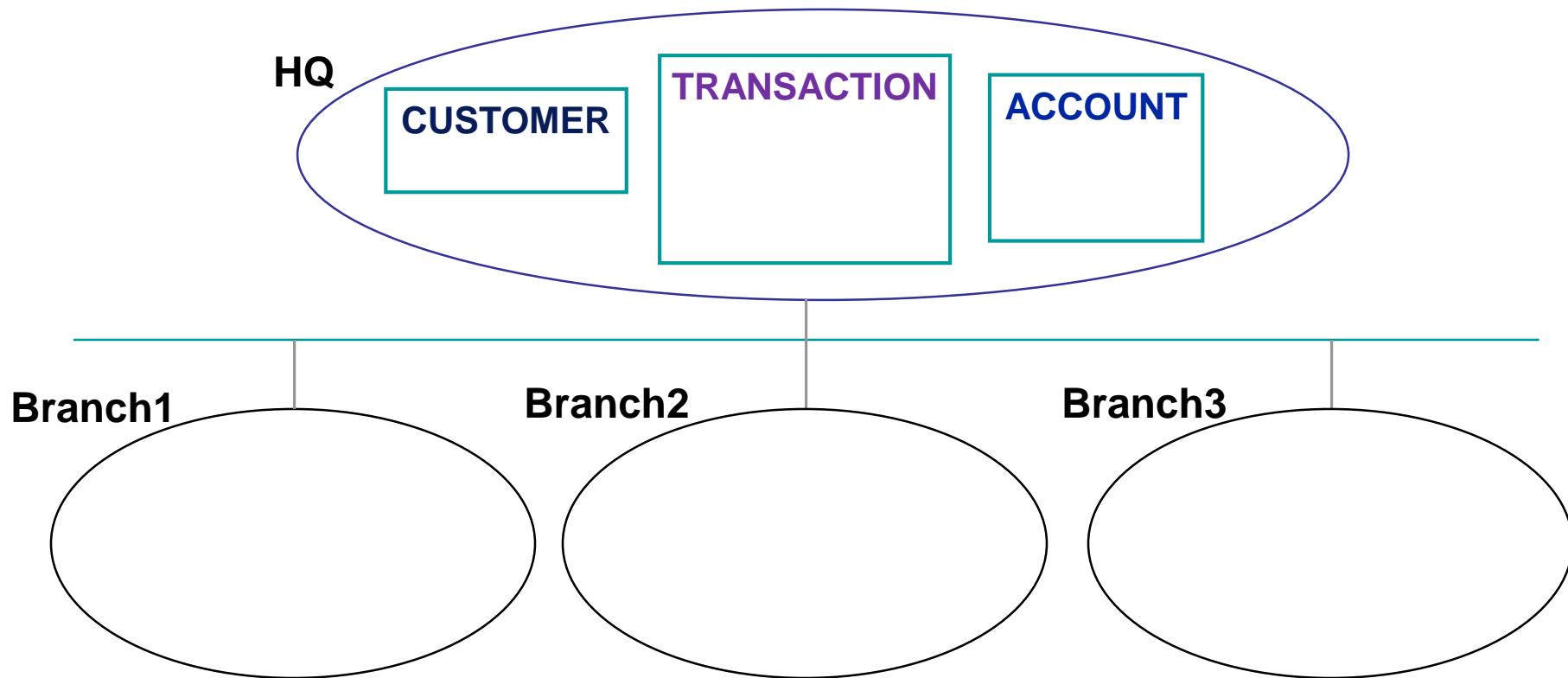
Fragment Allocation

- Network:
 - 3 peripheral sites, 1 central site
- Allocation:
 - Local
 - centralized

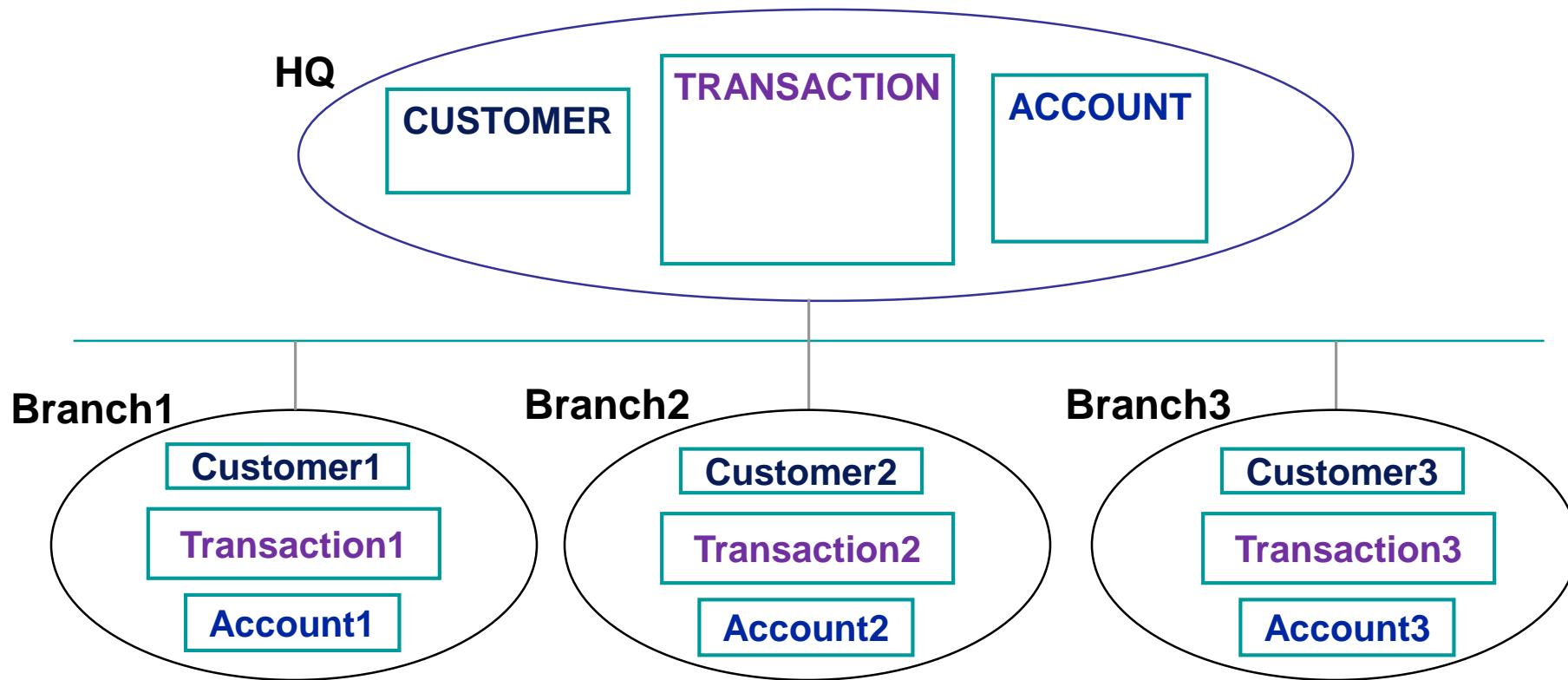
Fragment allocation



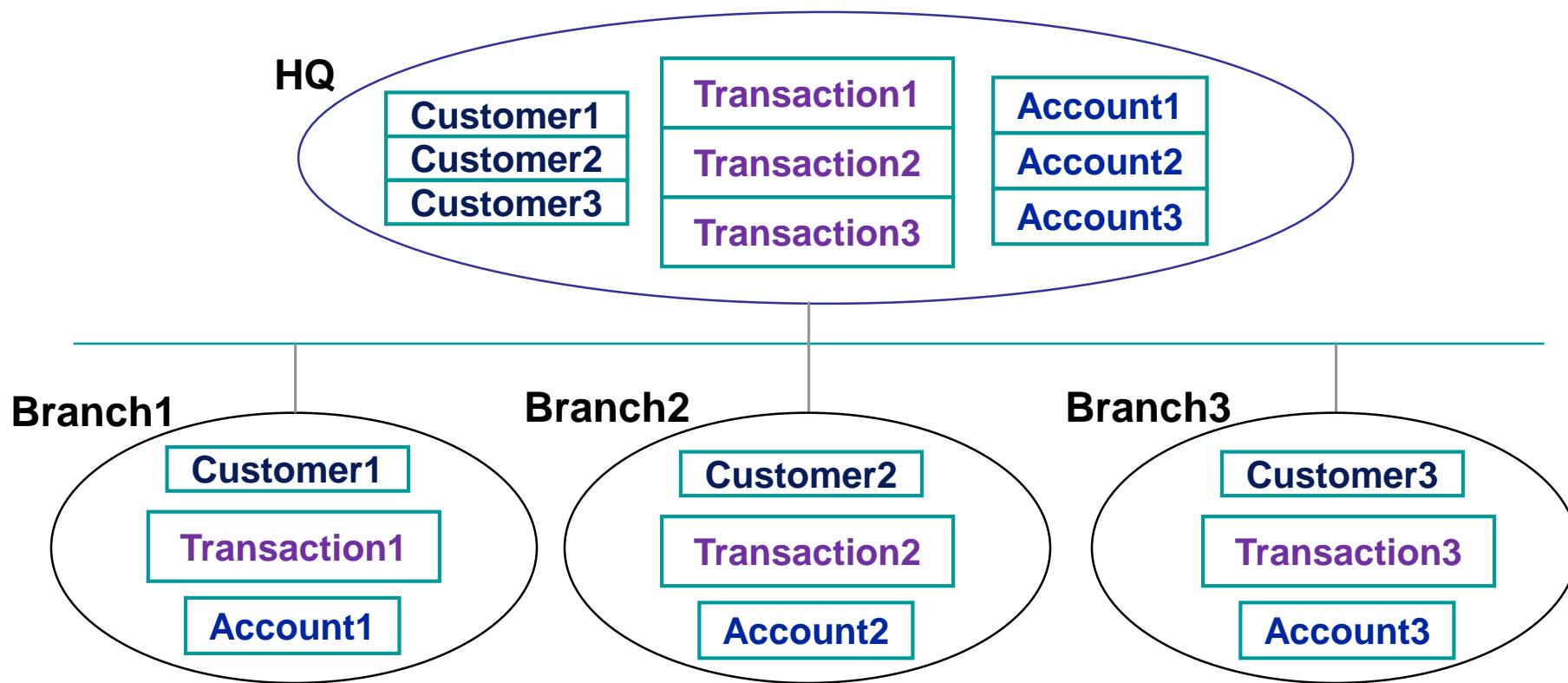
Fully centralized [A]



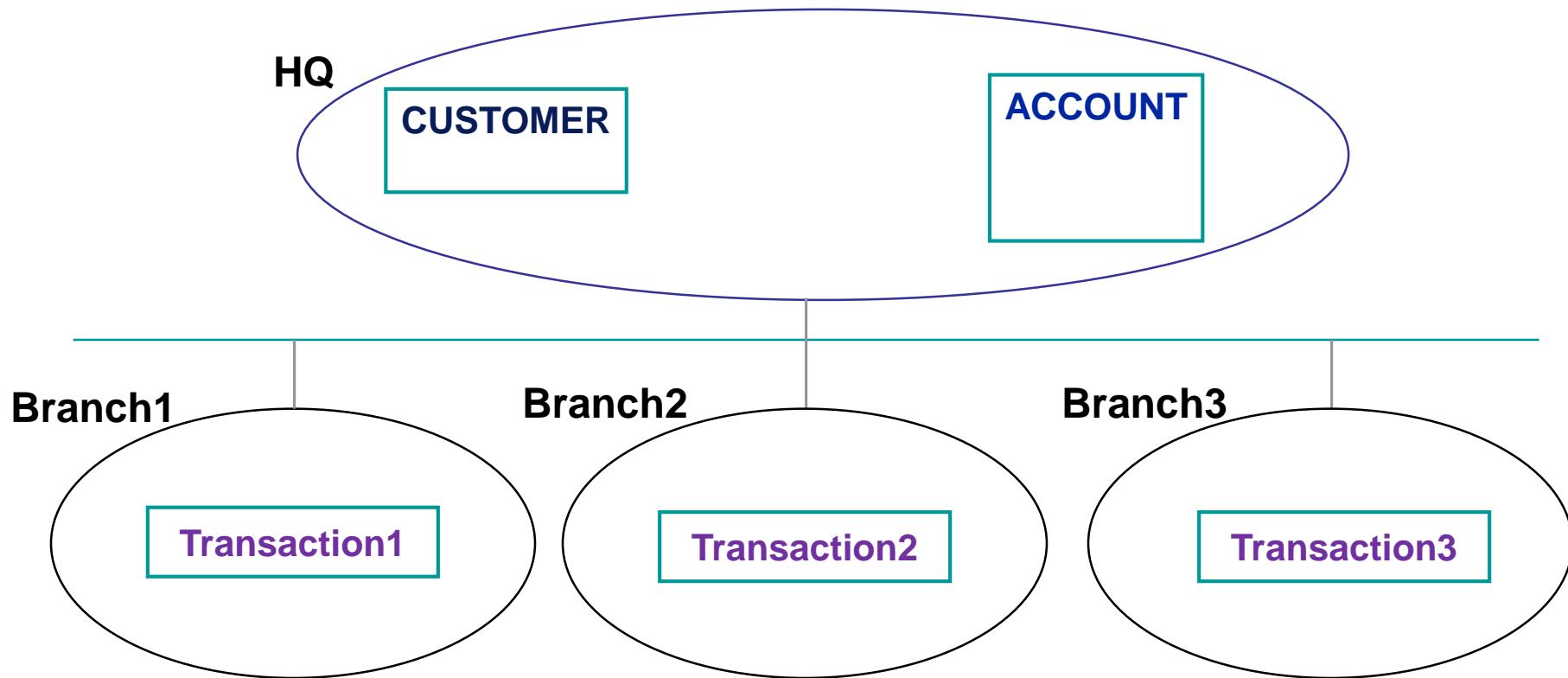
Centralized and distributed (fully replicated) [B]



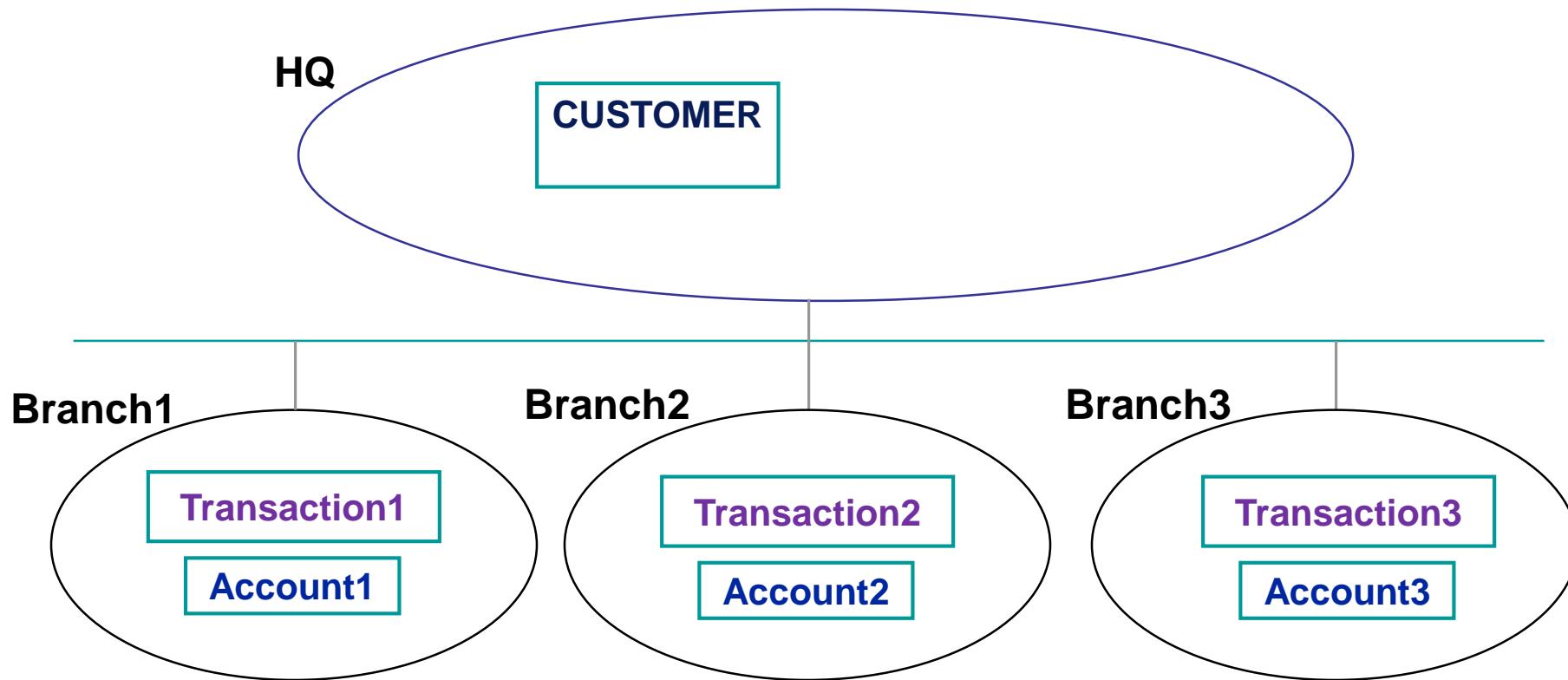
Centralized and distributed (fully replicated) [Bbis]



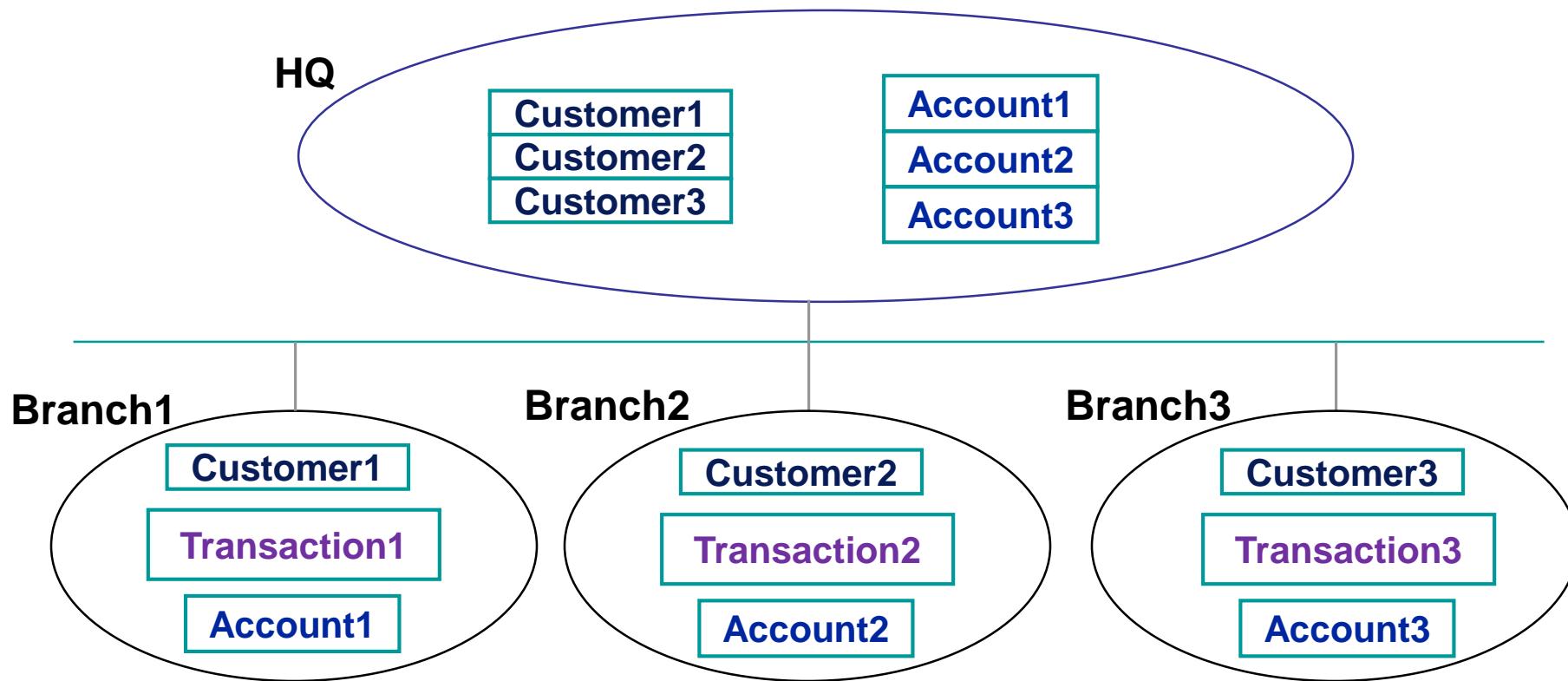
Partially distributed, no replication [C]



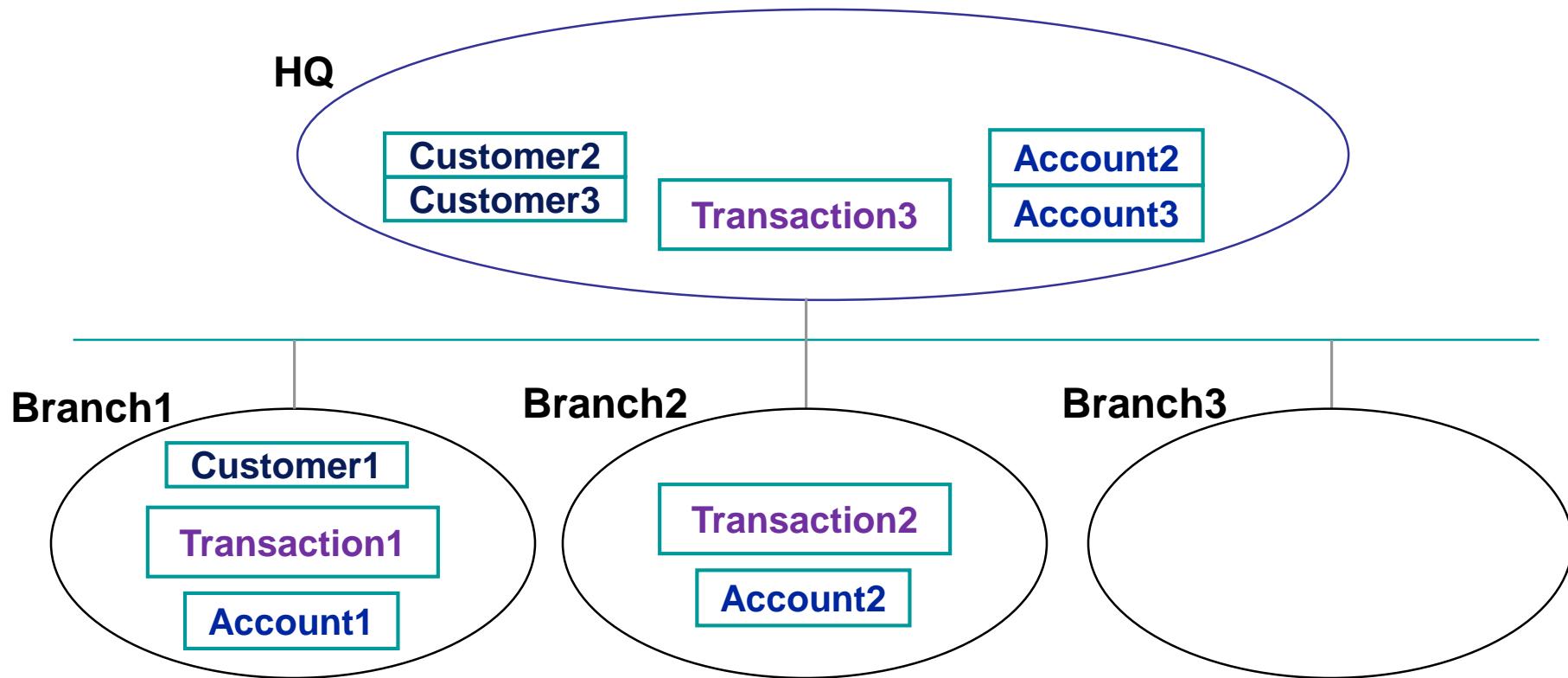
Partially distributed, no replication [D]



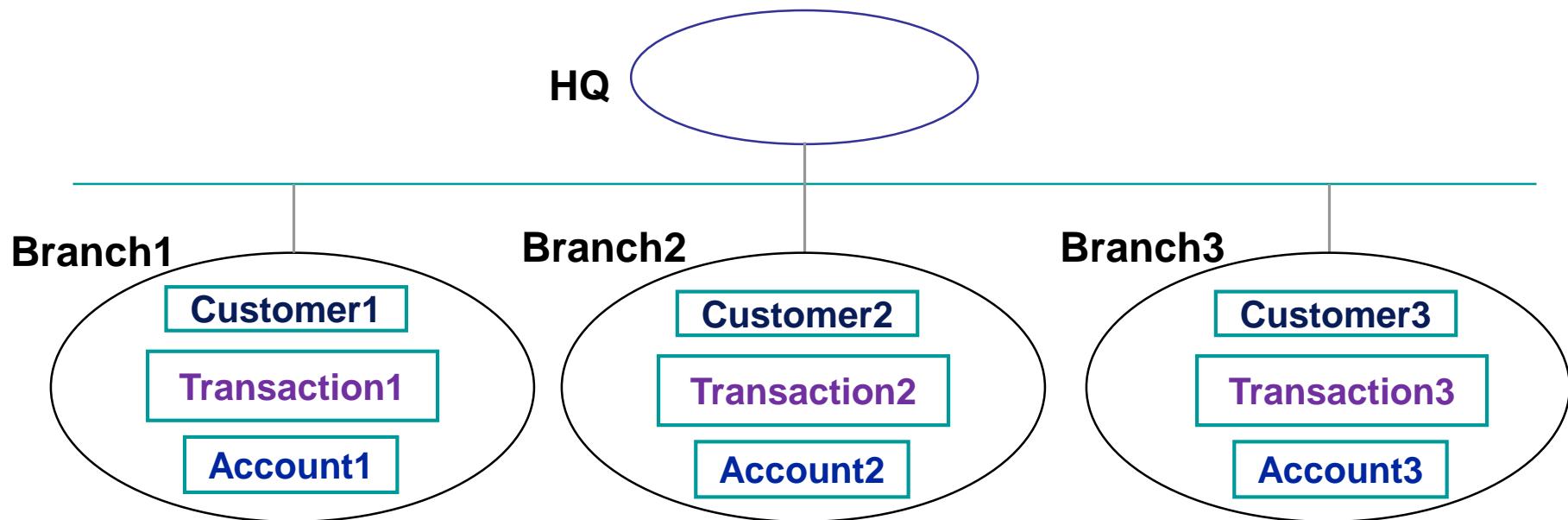
Partially centralized and fully distributed [E]



Asymmetric allocation [F]



Fully distributed (little replication) [G]



Customers owning more than one account are replicated on all branches on which they own at least one account

Distributed Join

- The most expensive distributed data analysis operation
- Consider a natural and frequent join operation:



Distributable Join

UNION

Branch1

Branch2

Branch3

Account1

join

Transaction1

Account2

join

Transaction2

Account3

join

Transaction3

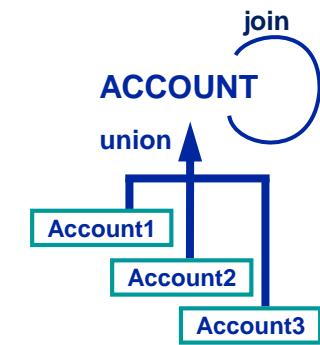
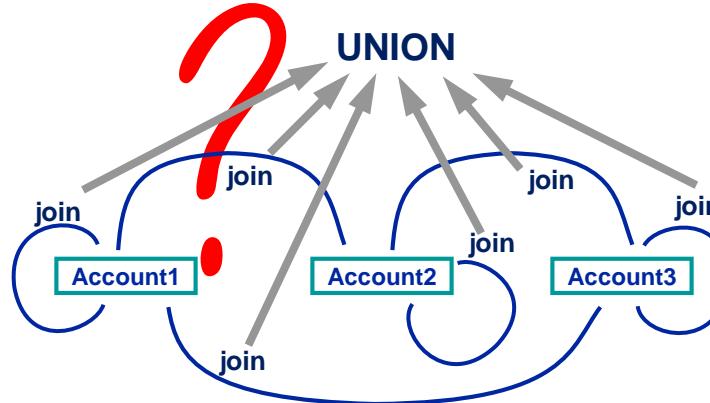
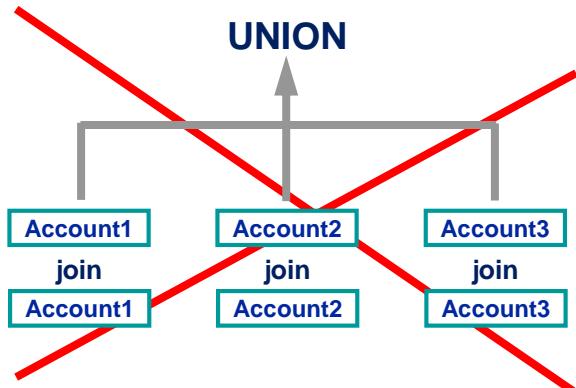
The join is along the join path used to build the derived fragmentation

Requirements for Distributed Join

- The domains of the **join attributes** must be **partitioned** and each partition must be assigned to a couple of fragments
- Example: for numeric values between 1 and 30,000:
 - Partition 1 to 10,000
 - Partition 10,001 to 20,000
 - Partition 20,001 to 30,000
- Some parallel systems distribute the data on the disks at the beginning, to obtain this distribution

Problematic examples: a non-distributable join

- Problematic Query: **customers with more than one account**
 - A self join on ACCOUNT
- The self join is not the one used to guide the fragmentation
 - Couples of matching accounts can be on any node



Problematic examples: a problematic fragmentation

- Problematic fragmentation
 - We extend the database with the following table, tracing couples of transactions that are *internal* money transfers (both the sender and receiver are customers of the bank)

INTERNALTRANSFER(Date, AccNoFrom, IncFrom, AccNoTo, IncTo)

- How to derive a fragmentation from ACCOUNT?
 - Based on the sending account? Or the receiving one?
 - What if we base it on both?
 - Both accounts may be on the same node, or different nodes...

Transparency Levels

- Different ways for composing queries, offered by commercial databases
- Three significant levels of transparency:
 - Transparency of fragmentation
 - Transparency of allocation
 - Transparency of language
- In *absence of transparency*, each DBMS accepts its own SQL 'dialect'
 - The system is heterogeneous and the DBMSs do not support a common interoperability standard

Transparency of Fragmentation

- Query:
 - Extract the balance of the account 45

```
SELECT Balance  
FROM Account  
WHERE Number=45
```

Transparency of Allocation

- Hyp.:
 - Account 45 is subscribed at Branch 1 (local)

```
SELECT Balance  
FROM Account1  
WHERE Number=45
```

Transparency of Allocation

- Hyp.:
 - Allocation of Account 45 is unknown, but possibly it is located at Branch 1

```
SELECT Balance FROM Account1  
WHERE Number=45  
IF (NOT FOUND) THEN  
( SELECT Balance FROM Account2  
WHERE Number=45  
UNION  
SELECT Balance FROM Account3  
WHERE Number=45 )
```

Transparency of Language

```
SELECT Balance FROM Account1@1  
WHERE Number=45  
IF (NOT FOUND) THEN  
( SELECT Balance FROM Account2@c  
WHERE Number=45  
UNION  
SELECT Balance FROM Account3@c  
WHERE Number=45 )
```

Transparency of Fragmentation

- Query:
 - Extract the transactions of the accounts with negative balance

```
SELECT Number, Incremental, Amount  
FROM Account AS C  
JOIN Transaction AS T  
ON C.Number=T.AccountNumber  
WHERE Balance < 0
```

Transparency of Allocation (distributed join)

```
SELECT Number, Incremental, Amount  
      FROM Account1 JOIN Trans1 ON .....  
      WHERE Balance < 0  
  
UNION  
  
SELECT Number, Incremental, Amount  
      FROM Account2 JOIN Trans2 ON .....  
      WHERE Balance < 0  
  
UNION  
  
SELECT Number, Incremental, Amount  
      FROM Account3 JOIN Trans3 ON .....  
      WHERE Balance < 0
```

Transparency of Language

```
SELECT Number, Incremental, Amount  
      FROM Account1@1 JOIN Trans1@1 ON .....  
     WHERE Balance < 0
```

UNION

```
SELECT Number, Incremental, Amount  
      FROM Account2@C JOIN Trans2@C ON .....  
     WHERE Balance < 0
```

UNION

```
SELECT Number, Incremental, Amount  
      FROM Account3@C JOIN Trans3@C ON .....  
     WHERE Balance < 0
```

Transparency of Fragmentation

- Update:
 - Move Account 45 from Branch 1 to Branch 2

UPDATE Account

SET Branch = 2

WHERE Number = 45

AND Branch = 1

Transparency of Allocation (and Replication)

INSERT INTO Account2

SELECT Number, Name, 2, Balance FROM Account1 WHERE Number = 45

INSERT INTO Trans2

SELECT * FROM Trans1 WHERE AccountNumber = 45

DELETE FROM Trans1 WHERE AccountNumber = 45

DELETE FROM Account1 WHERE Number = 45

Transparency of Language

INSERT INTO Account2@2

SELECT Number, Name, 2, Balance FROM Account1@c WHERE Number=45

INSERT INTO Account2@c

SELECT Number, Name, 2, Balance FROM Account1@c WHERE Number=45

INSERT INTO Trans2@2

SELECT * FROM Trans1@c WHERE Number=45

INSERT INTO Trans2@c

SELECT * FROM Trans1@c WHERE Number=45

(similarly: 2 pairs of DELETE commands)

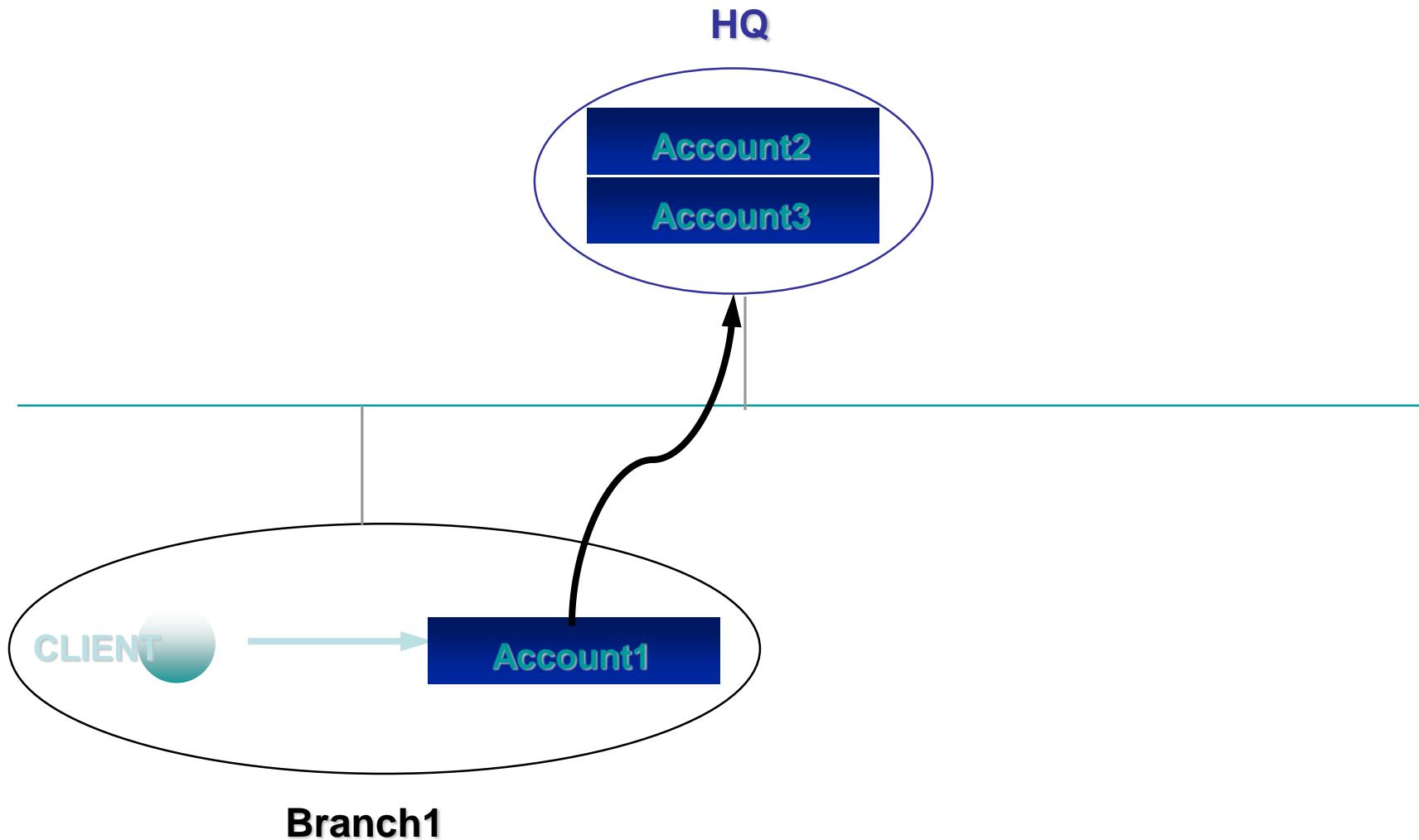
Distribution Design Problem

- Determining the best fragmentation and allocation of given tables
- Fragmentation should match locality characteristics, but there are trade offs.
 - With a university database, STUDENT allocated at the central admission office, COURSE distributed at the departments.
 - How should STUDY-PLAN & EXAM be fragmented?
 - Depending on the choice, ONE of the two joins with either STUDENT or COURSE is a distributed join, the other one is not.
- Allocation should give the ideal degree of redundancy
 - Redundancy speeds up retrieval and slows down updates
 - Redundancy increases availability and robustness

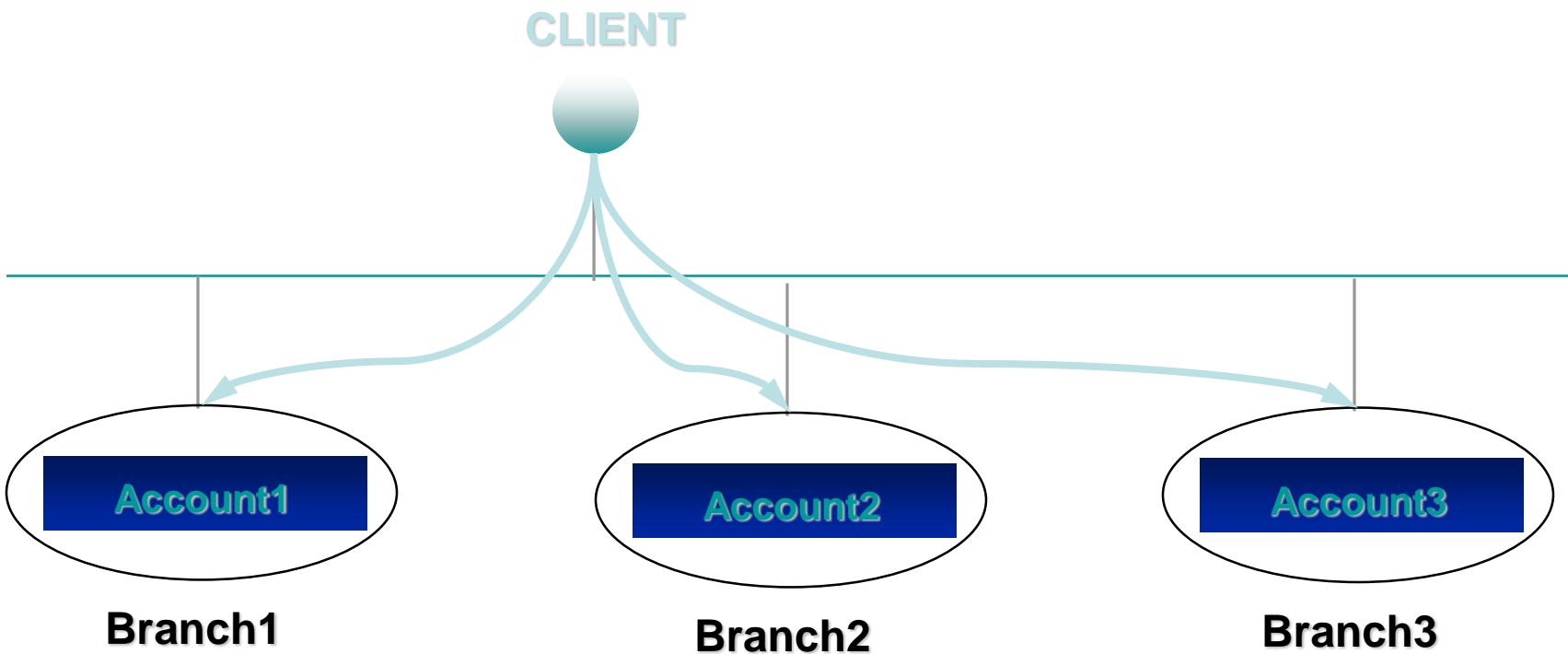
Efficiency

- **Query optimization**
- **Execution time**
 - **Serial execution**
 - **Parallel execution**

Serial Execution



Parallel Execution

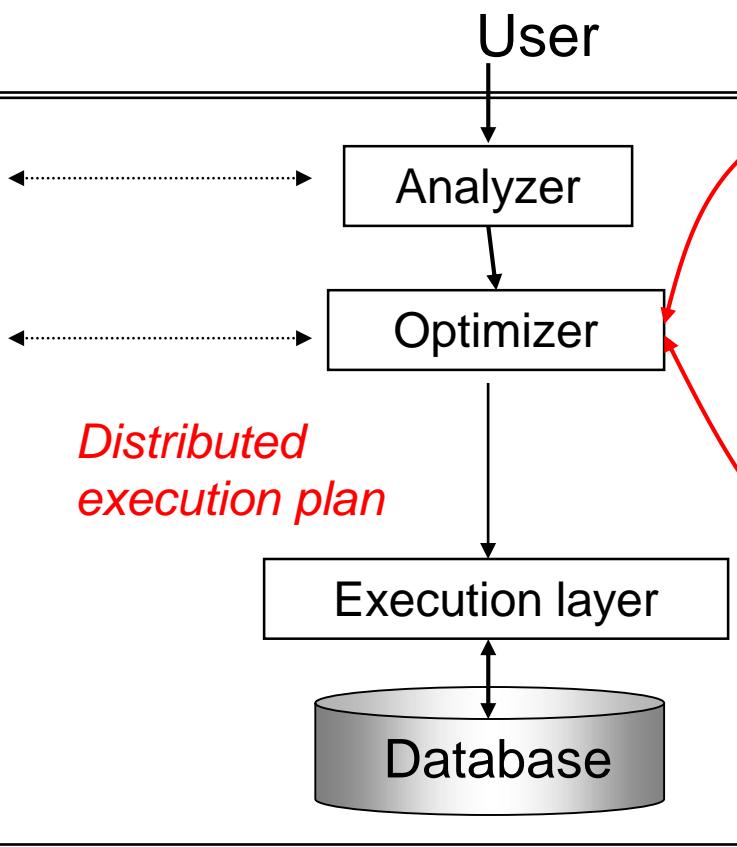


Distributed optimization with negotiation

Site S1

Distributed catalog

Distributed statistics



Site S2

Optimizer

Execution layer

Database

Site S3

Optimizer

Execution layer

Database

Distributed database with master-slave optimization

Site S1

Distributed catalogs

Distributed profiles

User

Analyzer

Optimizer

Distributed execution plan

Execution layer

Database

Site S2

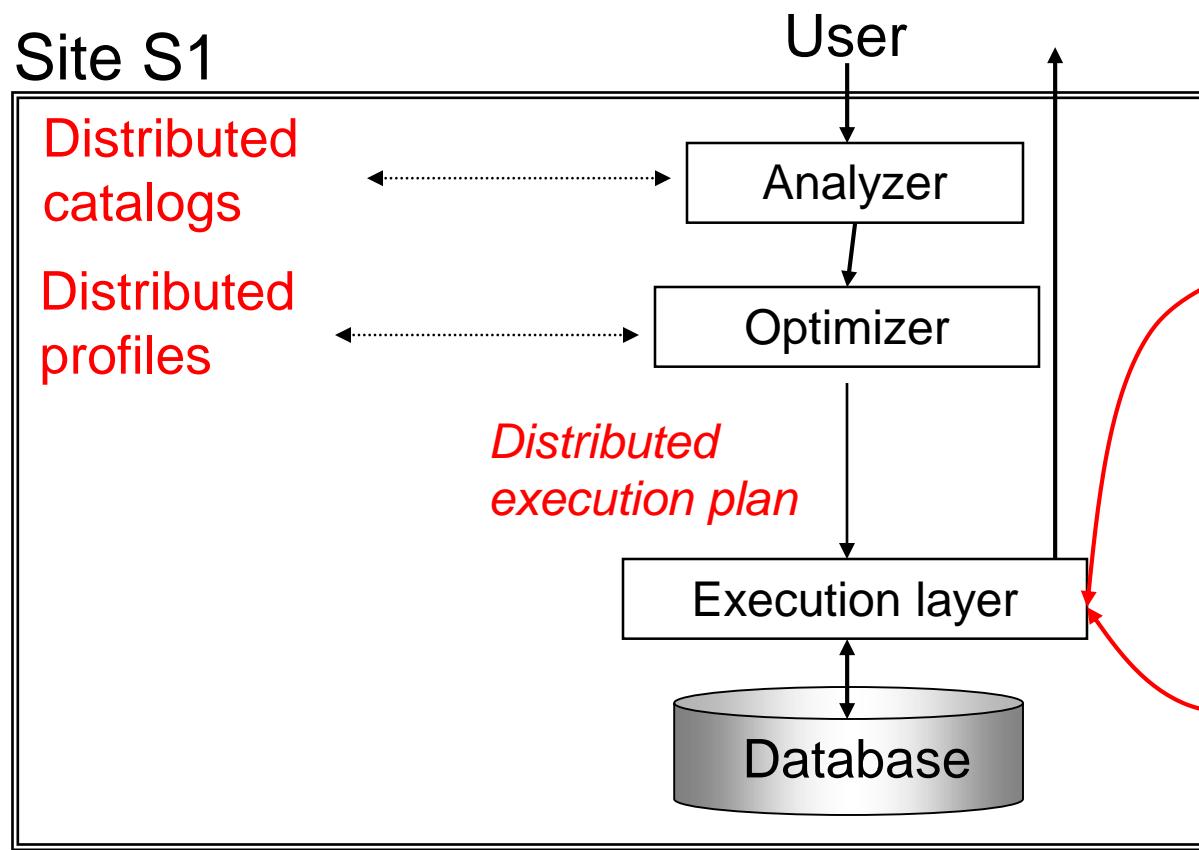
Execution layer

Database

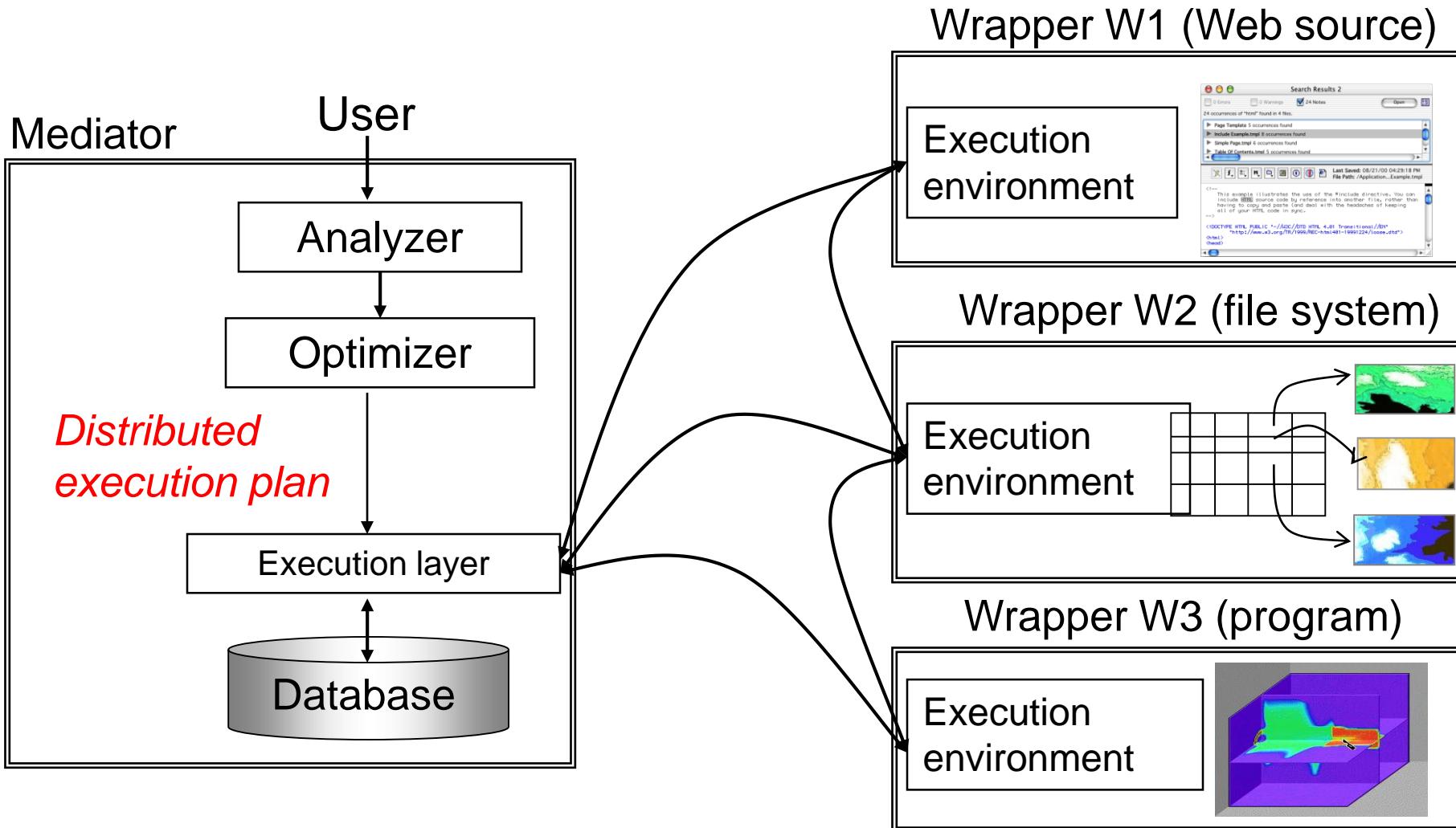
Site S3

Execution layer

Database



Distributed system with mediator and wrappers



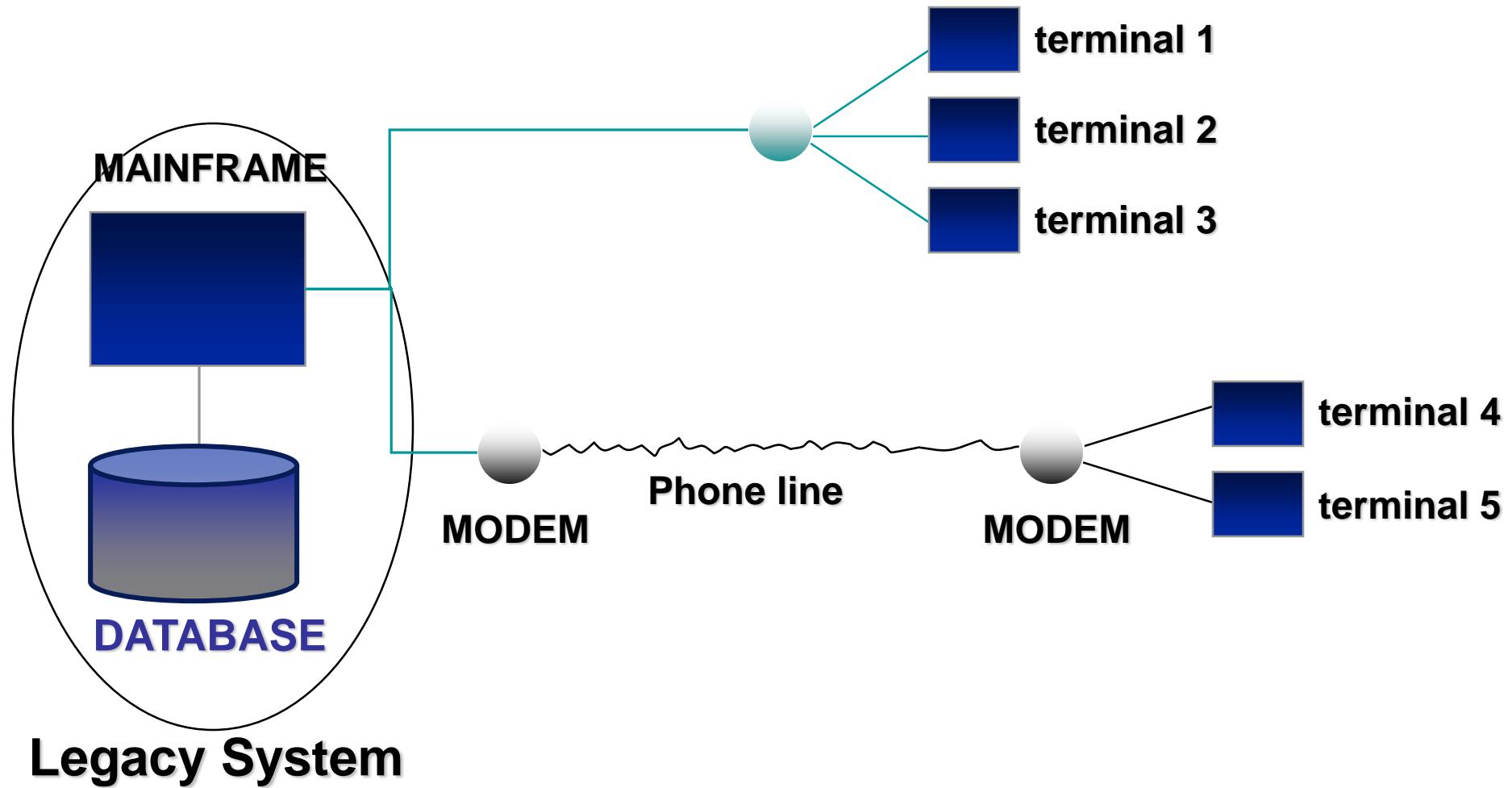
A pragmatic classification of distributed SQL applications

- **Remote request:** ability to support read-only SQL queries to a single remote database
- **Remote transaction:** ability to support SQL update queries to a single remote database
 - These are easily managed by ODBC/JDBC interfaces
- **Distributed transaction:** ability to support update transactions to many distributed database, every SQL query routed to a single database
 - Requires 2 phase commit
- **Distributed request:** ability to support update transactions to many distributed database, every SQL query possibly over many databases
 - Requires 2 phase commit and distributed query optimization

Legacy systems

- System architectures based on **mainframes** (powerful centralized computer), clients consist in very simple terminals (with textual interface)
- In many cases there is no source code or no documentation, thus they are very hard to change
- However they provide adequate performance and availability, that was obtained thanks to huge efforts on solid technologies
- Typically they are obsolete systems, but they still manage important applications: large banking applications, financial applications, flight booking systems

Legacy Systems



Obsolete Technologies of Legacy Systems

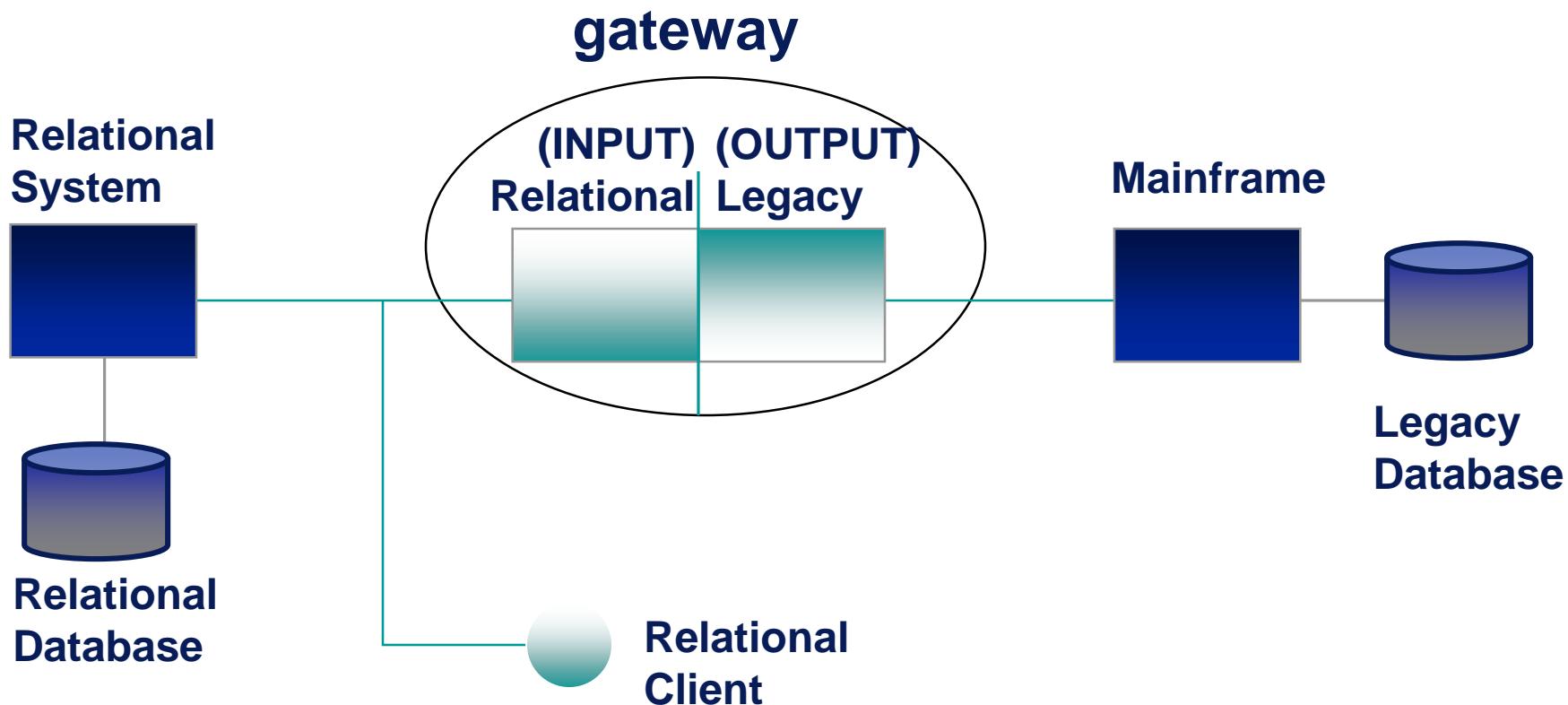
- Hardware (high \$ cost for slow but very reliable performance)
- Software (Cobol, DL/1: 1960-1980)
- On separated archives (even without DBMS)
- However, legacy systems are still reliable for operations that require continuous 7days 24h availability

Gateway (Wrapper)

- Software system that:
 - Is able to transfer requests from a (input) context to another (output) context
 - Provides server capabilities to the input system and client capabilities to the output system
 - Performs the needed conversion for the different formats and languages of the two contexts

Usage of Gateway Systems

- Between transactional systems
- Towards legacy systems



ACID vs BASE

- For large-scale systems, a new approach has been proposed as an alternative to classical ACID transactions

Basic Availability Soft-state Eventual consistency – BASE

- Consistency represents here the fact that the same query can get different results if issued at the same time in different network locations

Advanced Databases

6

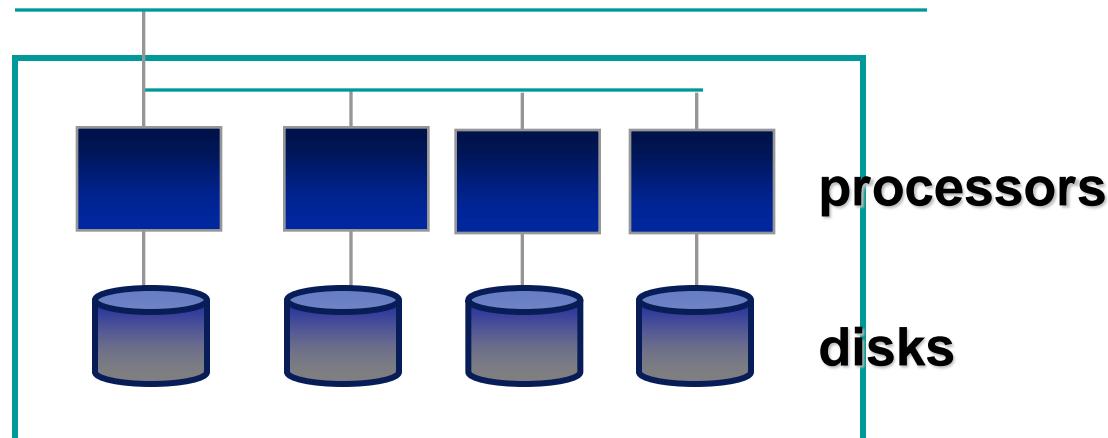
Parallel & Replicated Databases

Using Parallelism within the Servers

- Multiprocessor machines
- Identical computation on each processor
- Goal: increasing performances

→ PARALLEL DATABASE

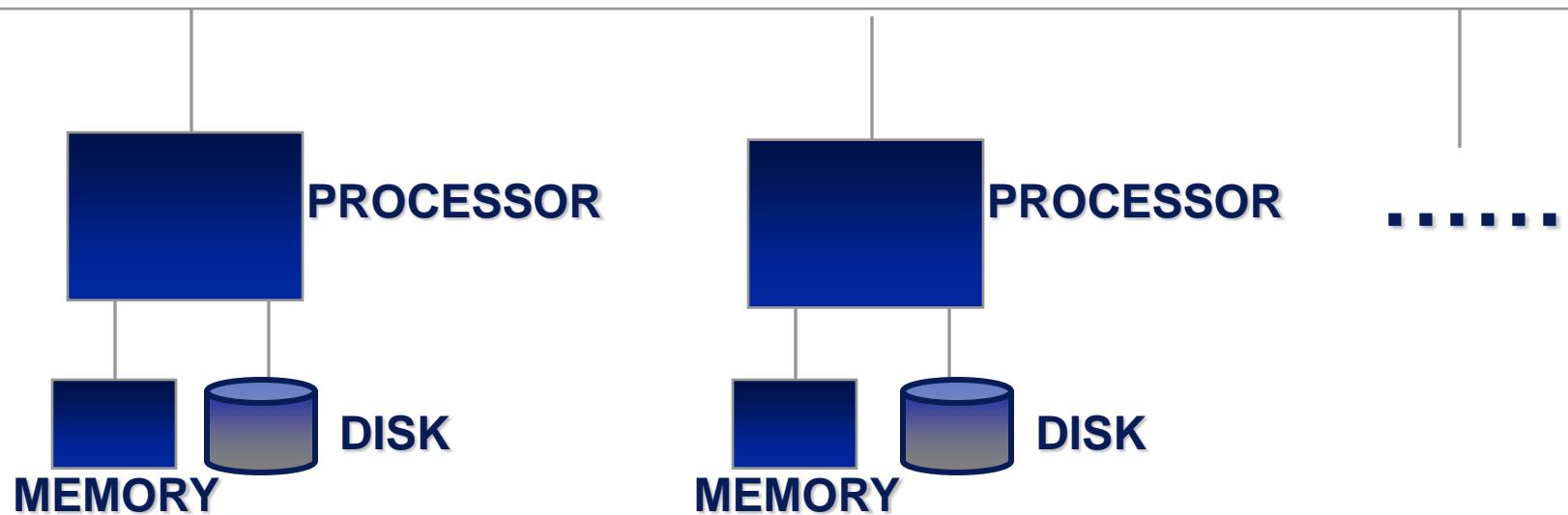
Database server
With parallelism



Architecture comparison

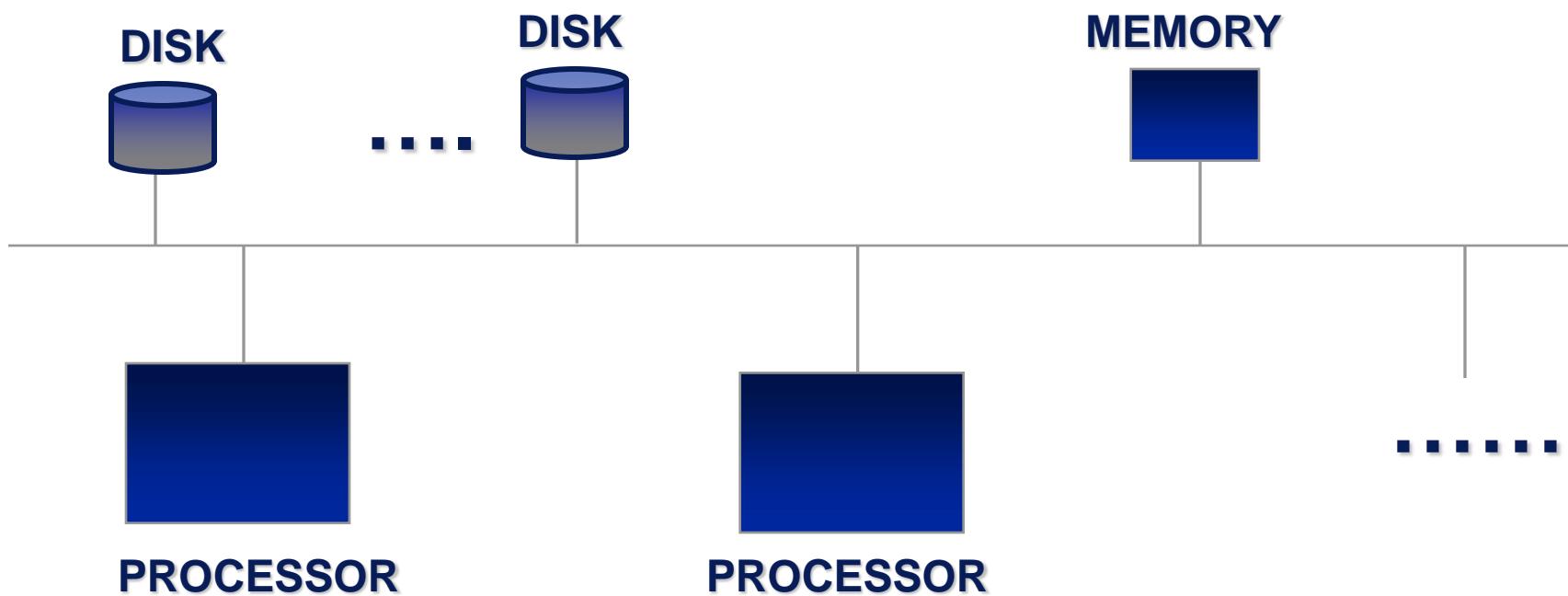
SHARED-NOTHING

Fast network



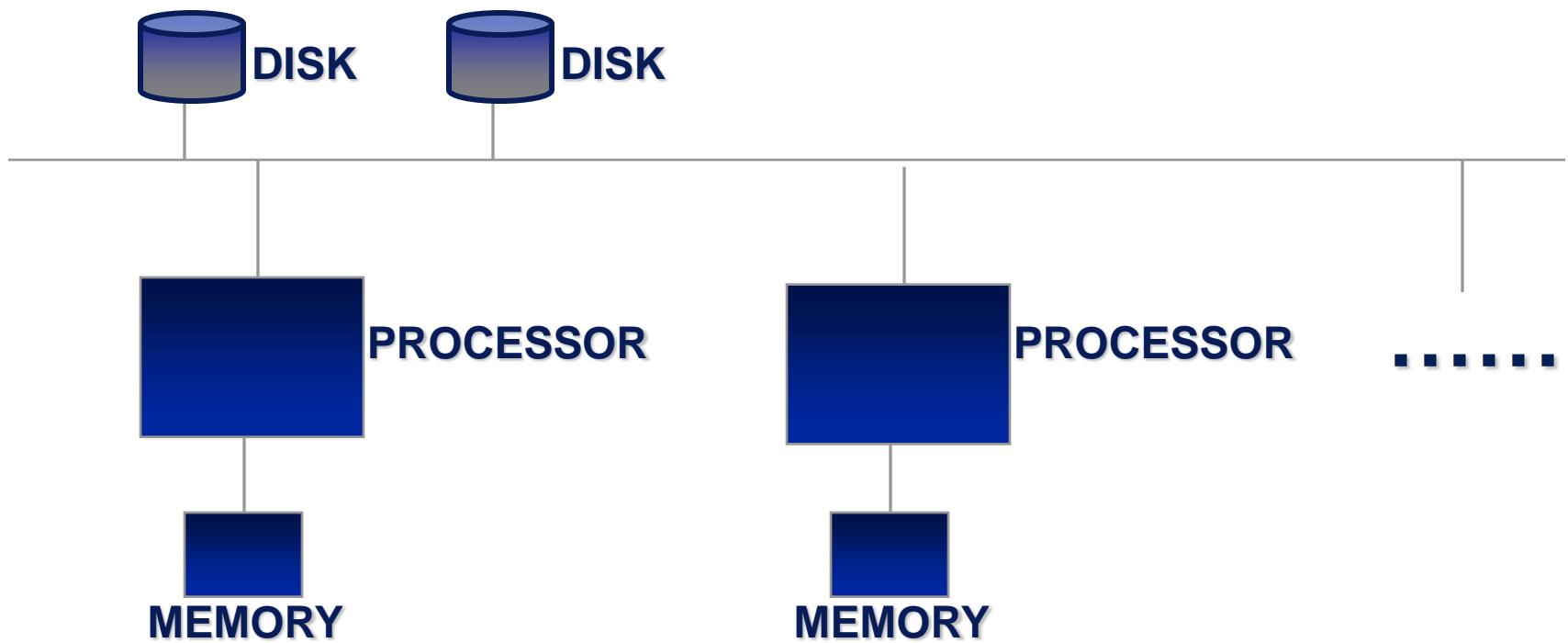
Architecture comparison

SHARED-MEMORY



Architecture comparison

SHARED-DISKS



Application scalability

- Load: set of all the applications (queries)
- Scalability: capability of a system to increase performance under an increased load
- Load growth dimensions:
 - Number of queries
 - Complexity of queries

Two Load Types

- Transactional
 - Load: short transactions
 - Measure: tps (transactions per second)
 - Response time: few seconds
- Data analysis
 - Load: complex SQL query
 - Response time: variable

Parallelism

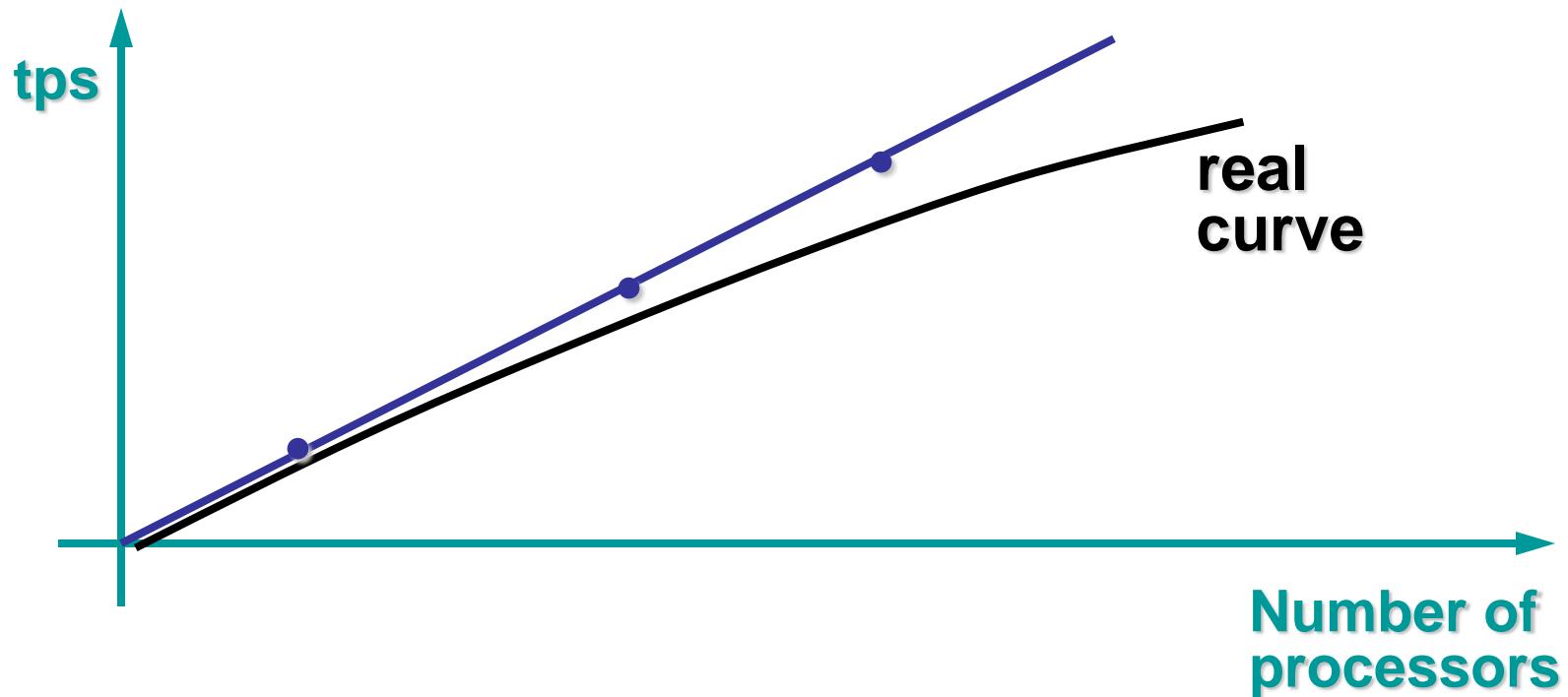
- Obtained through several cooperating processors, installed in a single system architecture
- Two types of parallelism:
 - **Inter-query:** each query is performed by a single processor (for transactional loads)
 - **Intra-query:** each query is performed by several processors (for data analysis loads)

Benchmark

- Methods for comparing performances of different (competing) systems
- Standardization
 - Of the Database
 - Of the load
 - Code of the transactions
 - Transmission
 - Frequency
 - Of the measuring conditions
- Different load types
 - Tpc-a: transactional
 - Tpc-b: mixed
 - Tpc-c: data analysis

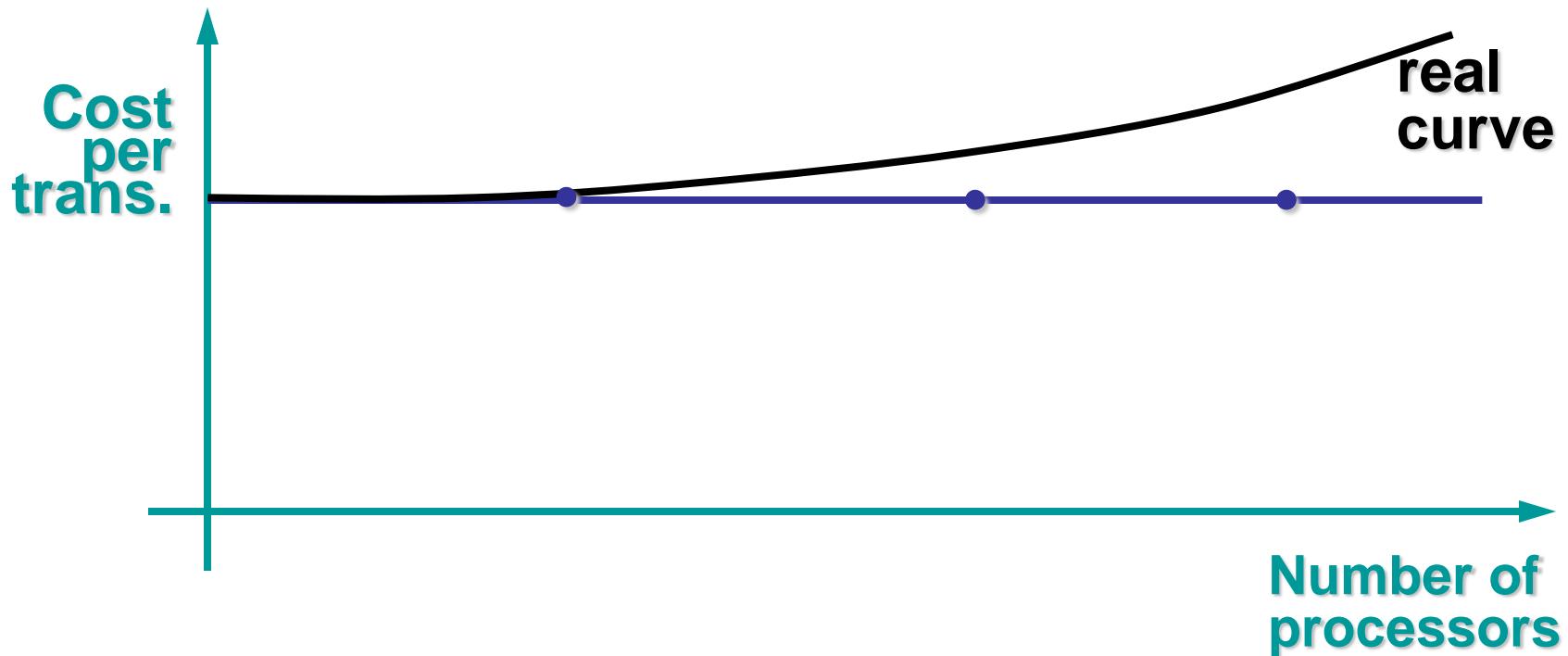
Speed-up curve

- Measures the increase of efficiency wrt the increasing number of processors



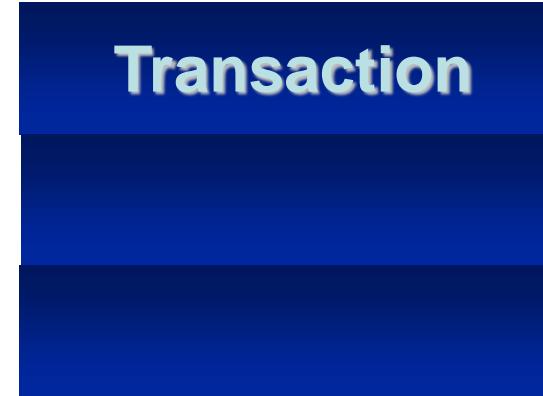
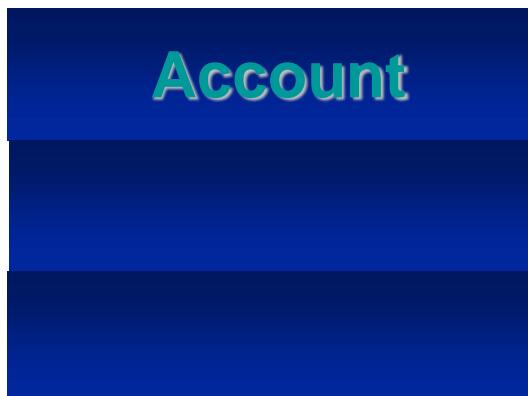
Scale-up curve

- Measures the total cost per transaction wrt the increasing number of processors

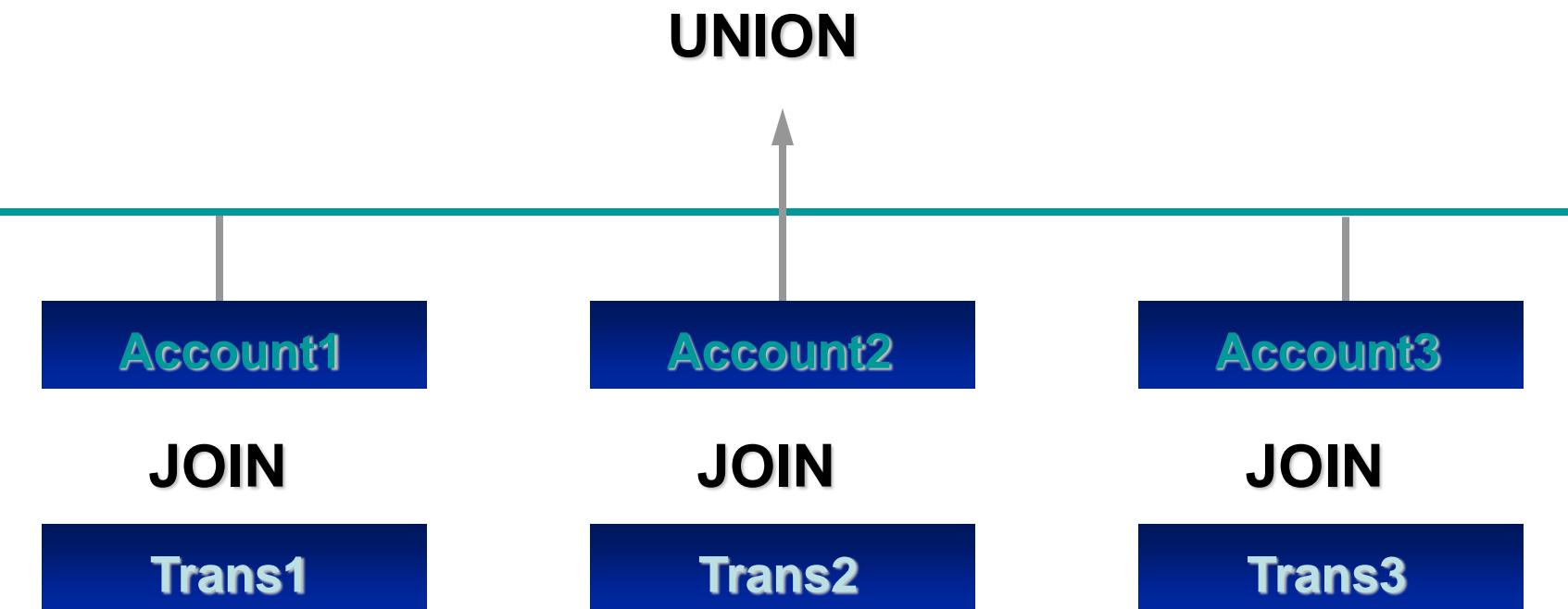


Distributed Join

- The most expensive distributed data analysis operation
- Let's consider:



Distributed Join



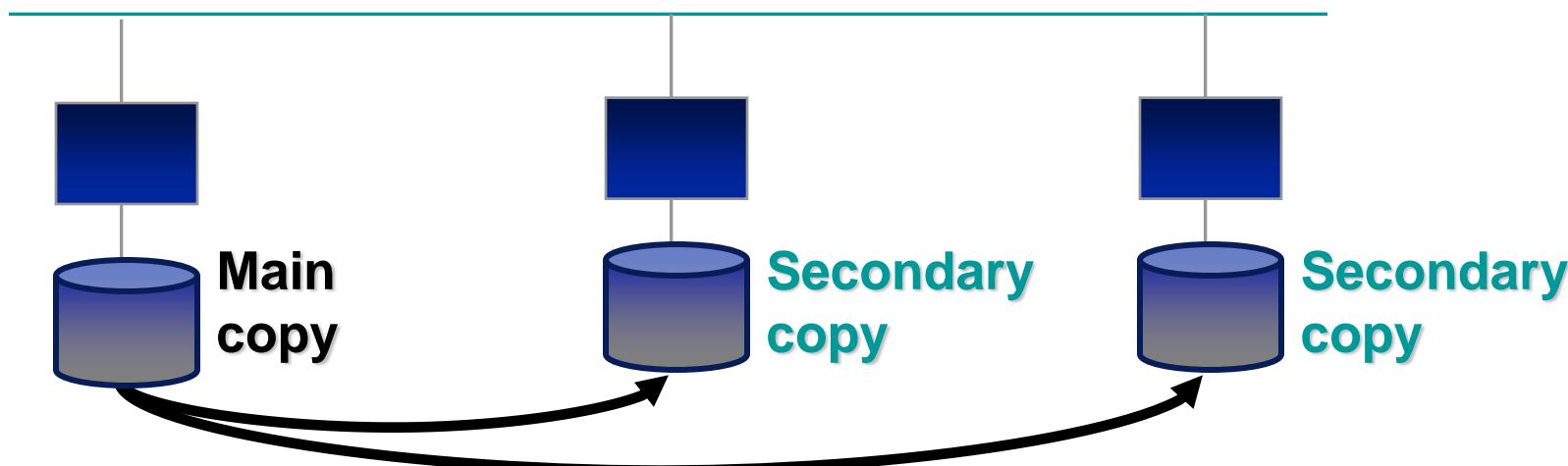
Requirements for Distributed Join

- The domains of the join attributes must be partitioned and each partition must be assigned to a couple of fragments
- Example: for numeric values between 1 and 30,000:
 - Partition 1 to 10,000
 - Partition 10,001 to 20,000
 - Partition 20,001 to 30,000
- Some parallel systems distribute the data on the disks at the beginning, to obtain this distribution

Data Replication

- Motivation:
 - Higher availability, efficiency, reliability,
 - Different data management

→ Replicated Databases

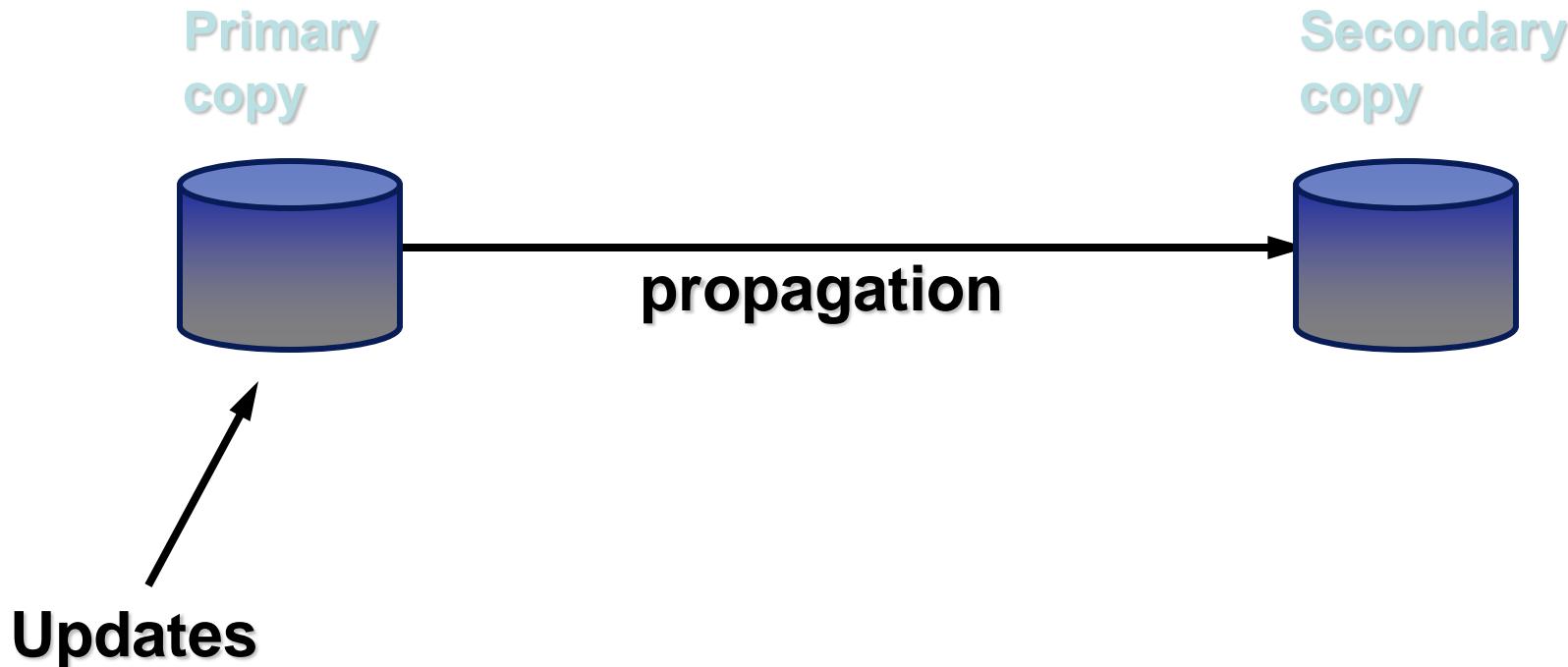


Data Replication

- A fundamental ingredient in information systems
- Motivations:
 - Efficiency
 - Reliability
 - Autonomy

Replication Methods

- Asymmetric Replication



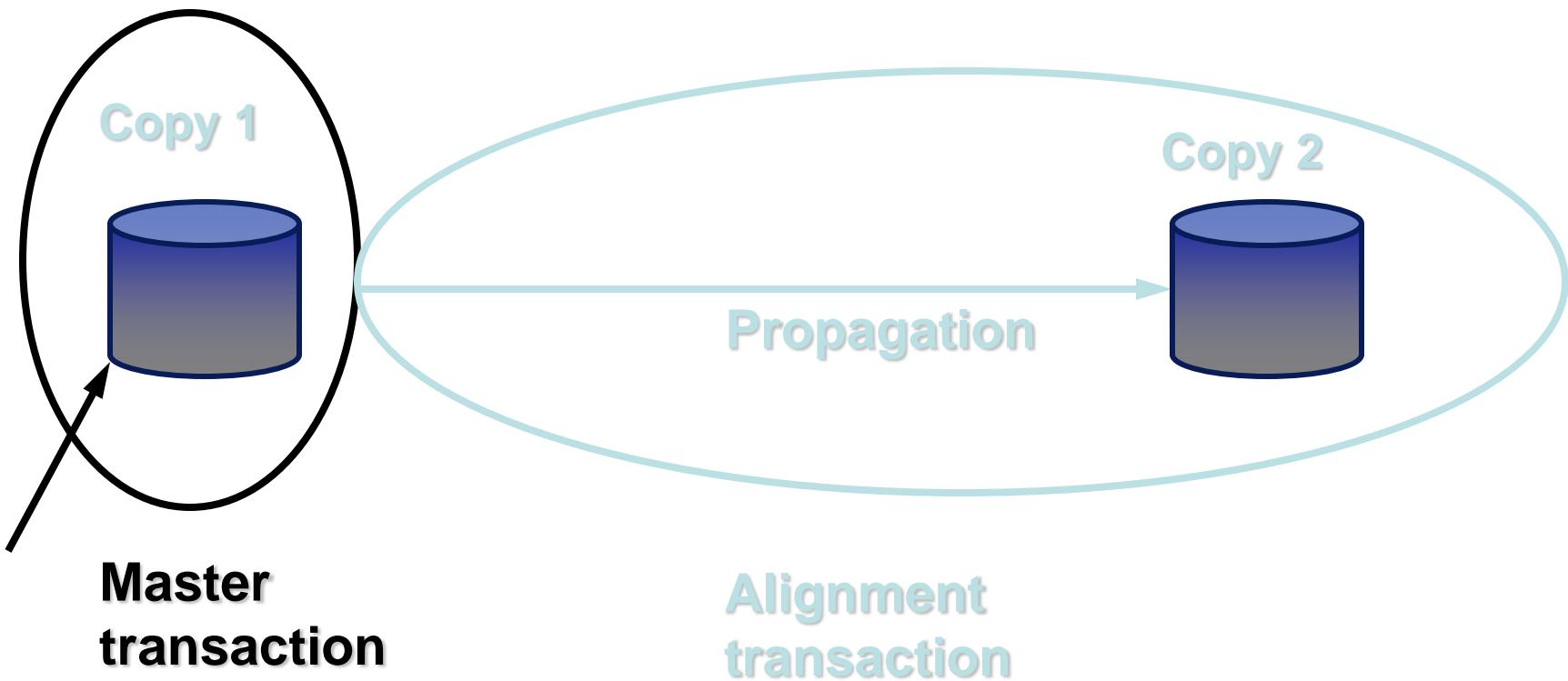
Replication Methods

- Symmetric replication



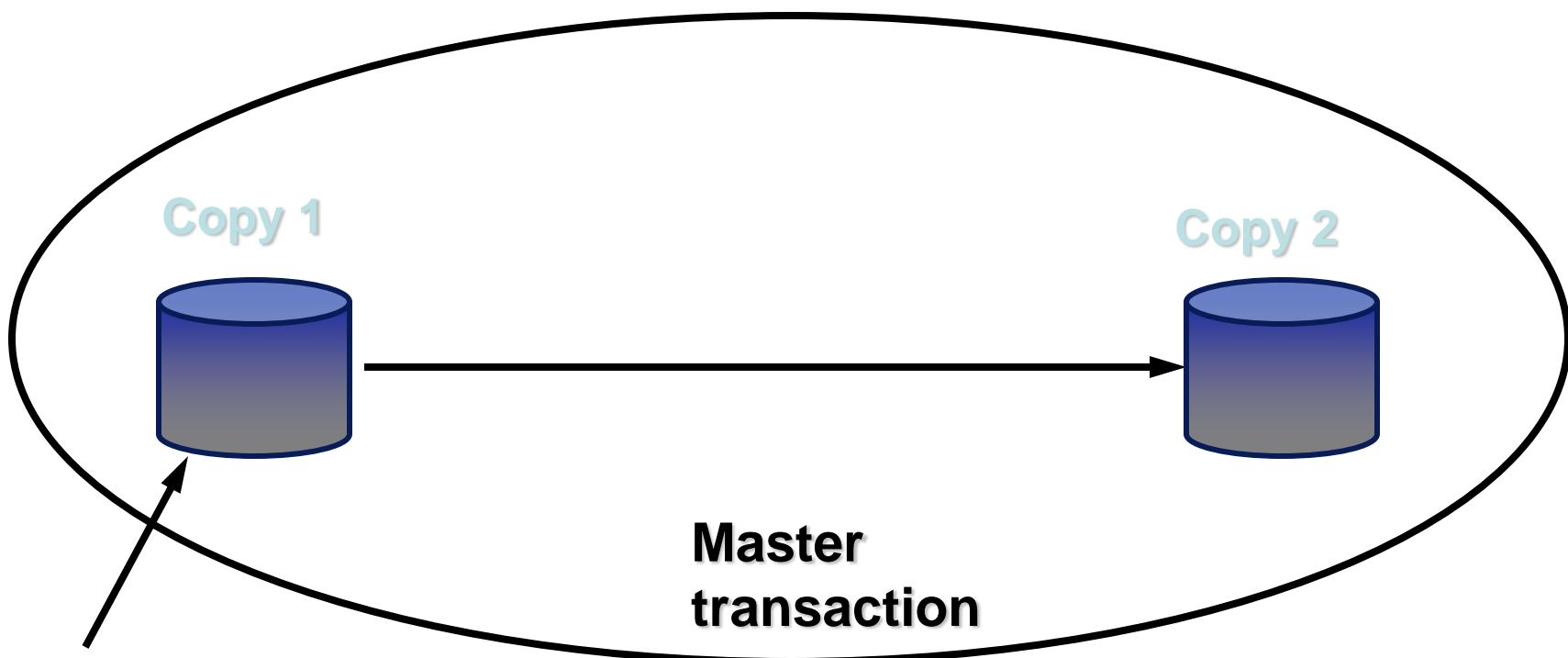
Transmission of Updates

- Asynchronous transmission



Transmission of Updates

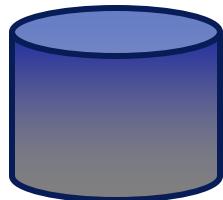
- Synchronous transmission



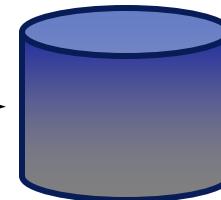
Alignment Techniques

- Refresh

Copy 1



Copy 2



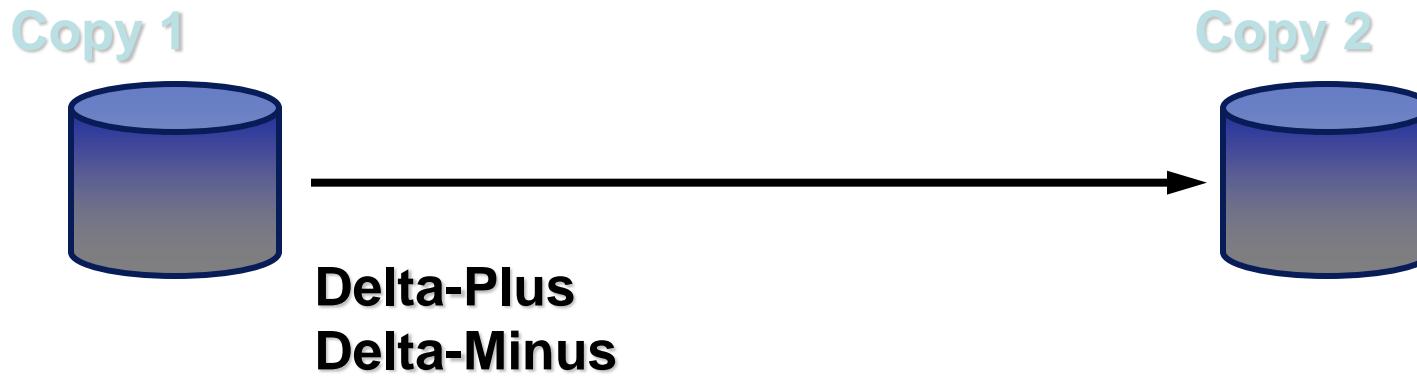
Whole content
of Copy 1



- Alignment can be:
 - Periodic
 - On command
 - On update accumulation

Alignment Techniques

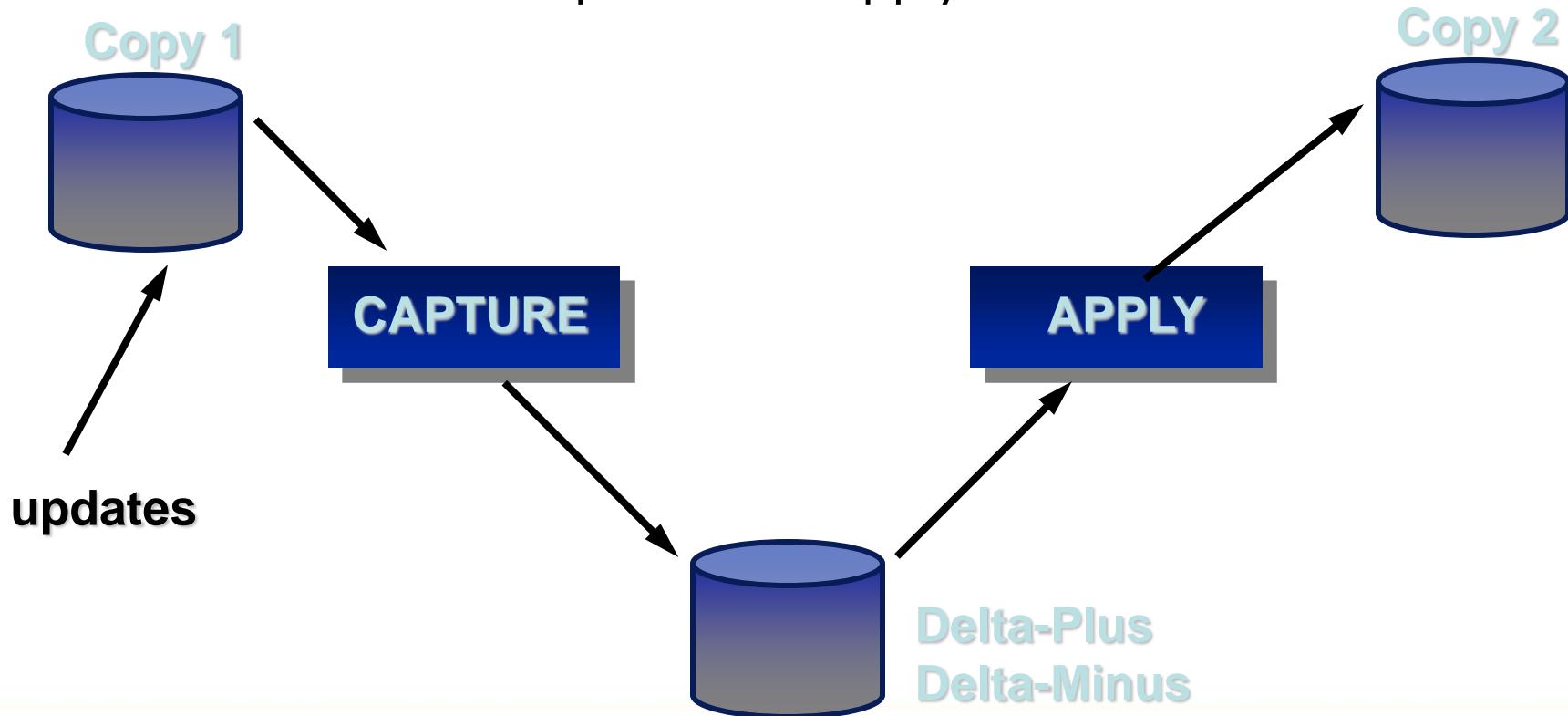
- Incremental



- Alignment can be:
 - Periodic
 - On command
 - On update accumulation

Replication Mechanisms

- asymmetric, asynchronous, incremental
- Product: Replication Manager
- Two modules: Capture and Apply



Replication Triggers

- Capture data variations within the tables Delta-Plus and Delta-Minus, transparently wrt the applications
- This technique was initially used to support replication, currently most systems prefer extracting deltas from logs and not from the database

Replication Triggers

```
CREATE TRIGGER Capture-Ins  
AFTER INSERT ON PRIMARY  
FOR EACH ROW  
INSERT INTO Delta-Plus VALUES (NEW.*)
```

```
CREATE TRIGGER Capture-Del  
AFTER DELETE ON PRIMARY  
FOR EACH ROW  
INSERT INTO Delta-Minus VALUES (OLD.*)
```

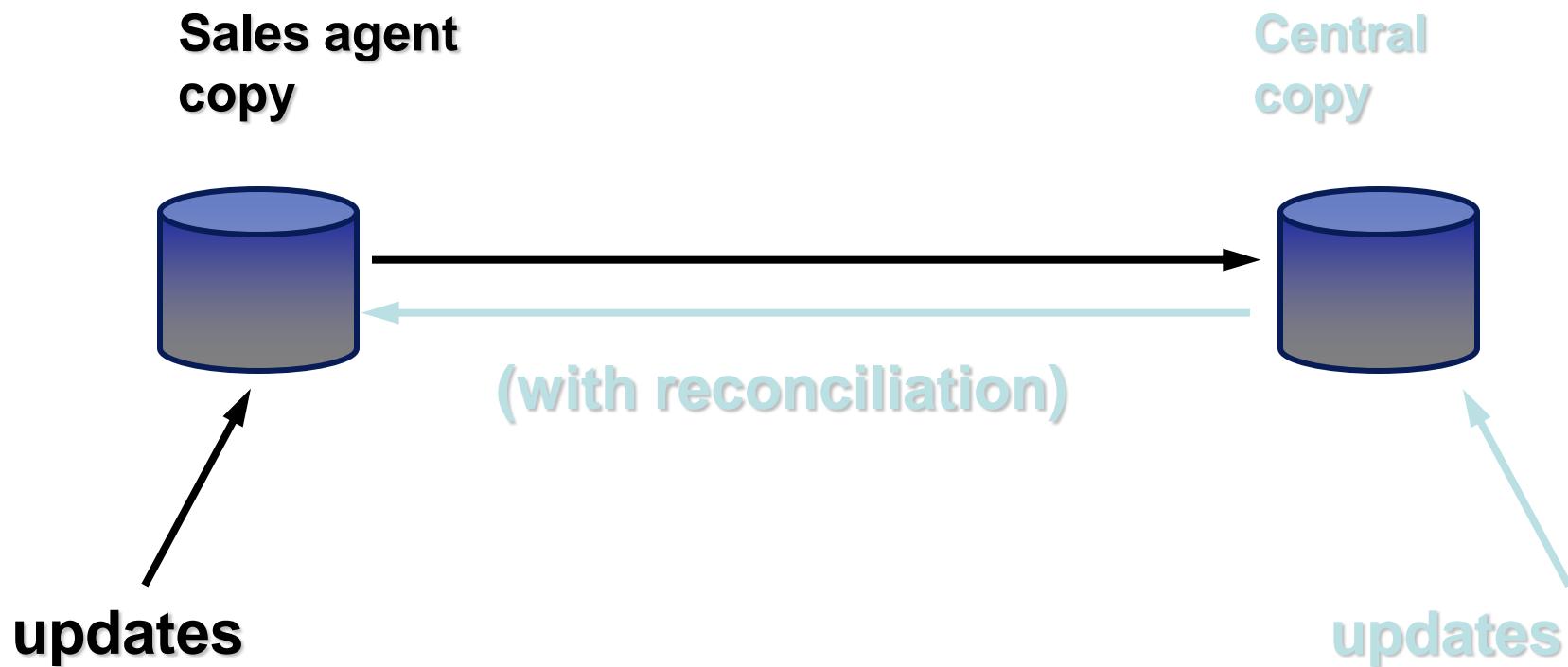
```
CREATE TRIGGER Capture-Upd  
AFTER UPDATE ON PRIMARY  
FOR EACH ROW  
BEGIN  
    INSERT INTO Delta-Plus VALUES (NEW.*)  
    INSERT INTO Delta-Minus VALUES (OLD.*)  
END
```

A Special Case: Replication in Mobile Computers

- Mobile computers: occasionally connected to the network
- Copies can be disconnected for hours or even days, then reconnected (reconciliation)
- Example Application: mobile sales agents

Disconnected Copies Re-alignment

- Often requires symmetric replication



Advanced Databases

7

Data Warehouses

An Environment for Data Analysis

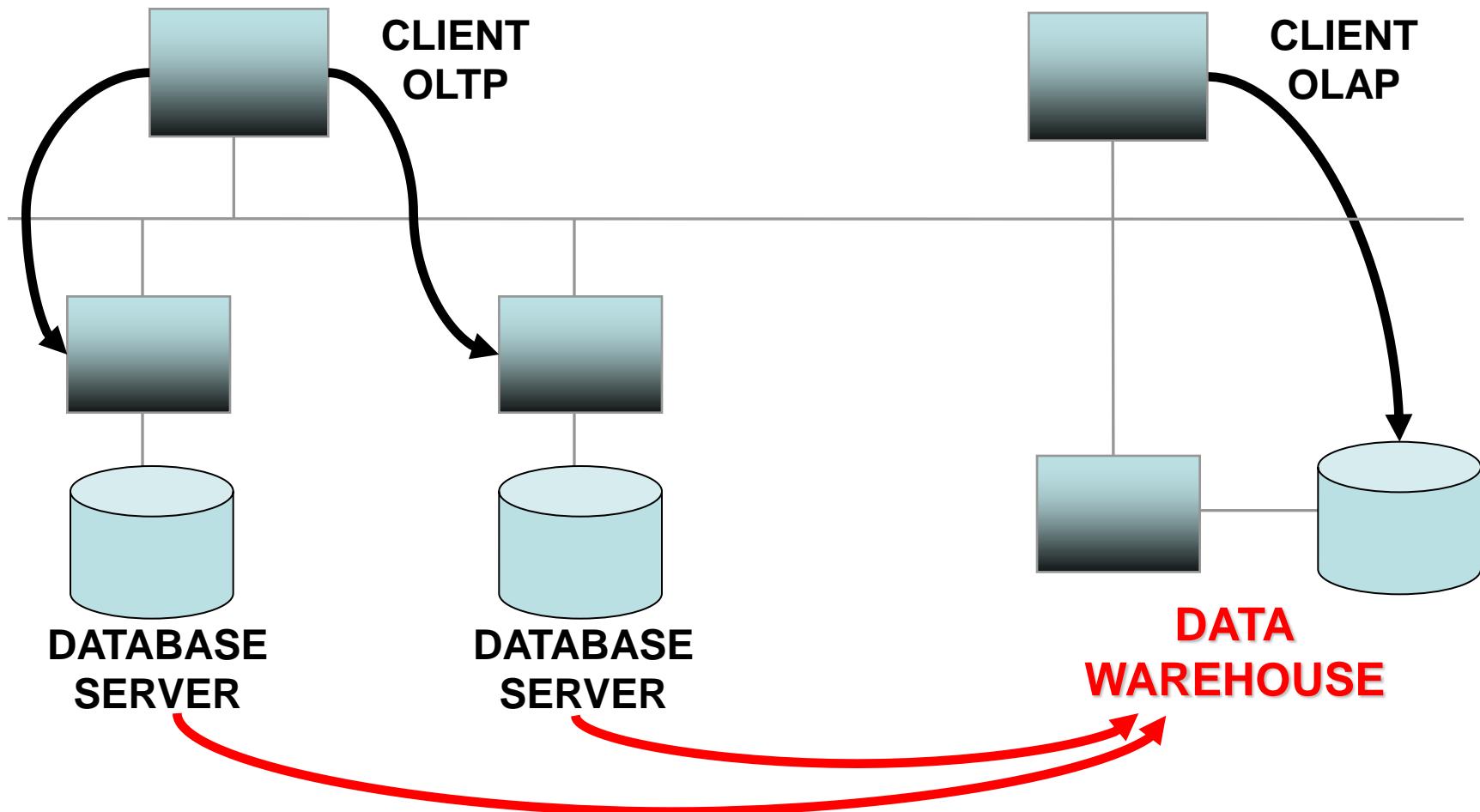
- **DATA WAREHOUSE**

- A structured description of all those data that are necessary for a strategic analysis of the trends and the behaviour of a firm

- **ON-LINE ANALYTICAL PROCESSING (OLAP)**

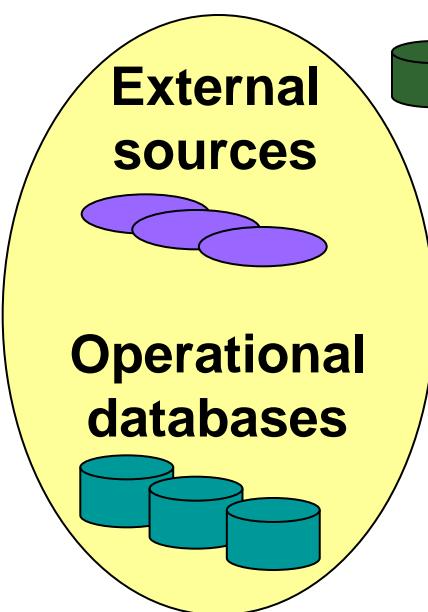
- The name given to analysis activities (it is contrasted to On Line Transaction Processing, OLTP)

Interaction between OLTP and OLAP

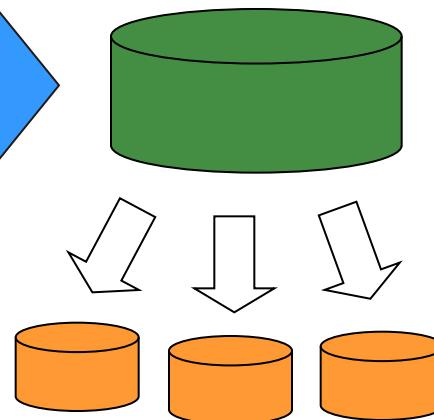
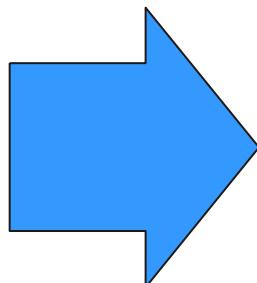


An Architecture for Data Warehousing

Monitoring & Administration



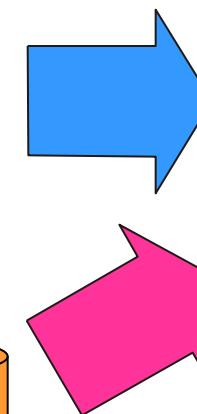
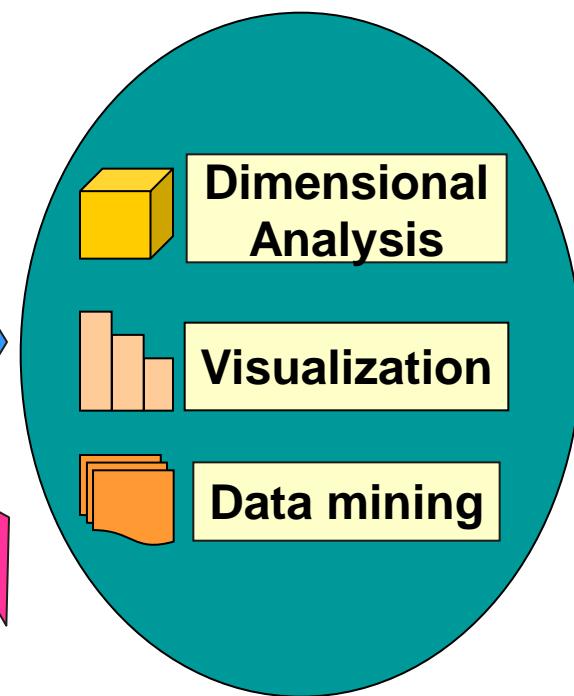
Enterprise Data Warehouse



Data sources

Data Mart

Analysis tools



Data Warehouse (DW) and Data Mart (DM)

- A Data Warehouse often integrates several Data Marts
- Users typically address one specific Data Mart
- Data Marts share common data
- Each Data Mart is responsible for one specific aspect of the firm business

Star model

- The ***star model*** is used for each Data Mart
 - Also known as ***multi-dimensional schema***
- It is a conceptual model which poses some restrictions
- Advantages:
 - Availability of suitable specific query interfaces
 - Good performance
 - Straightforward design of the relational schema

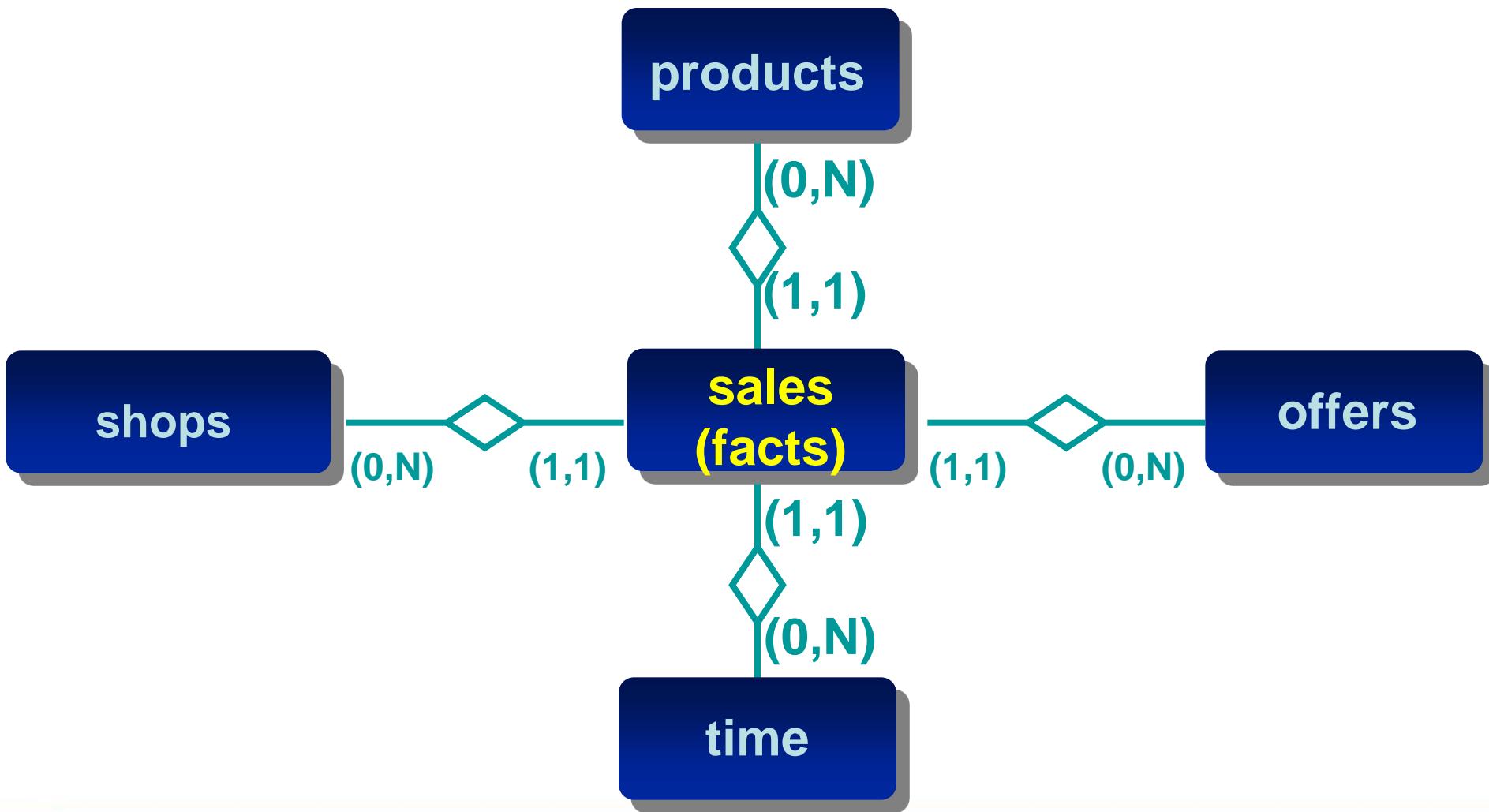
Multi-dimensional representation

- Relevant concepts:
 - **fact** — an aspect which is crucial for the analysis
 - **measure** — an atomic property of a fact
 - **dimension** — a specific perspective for the analysis

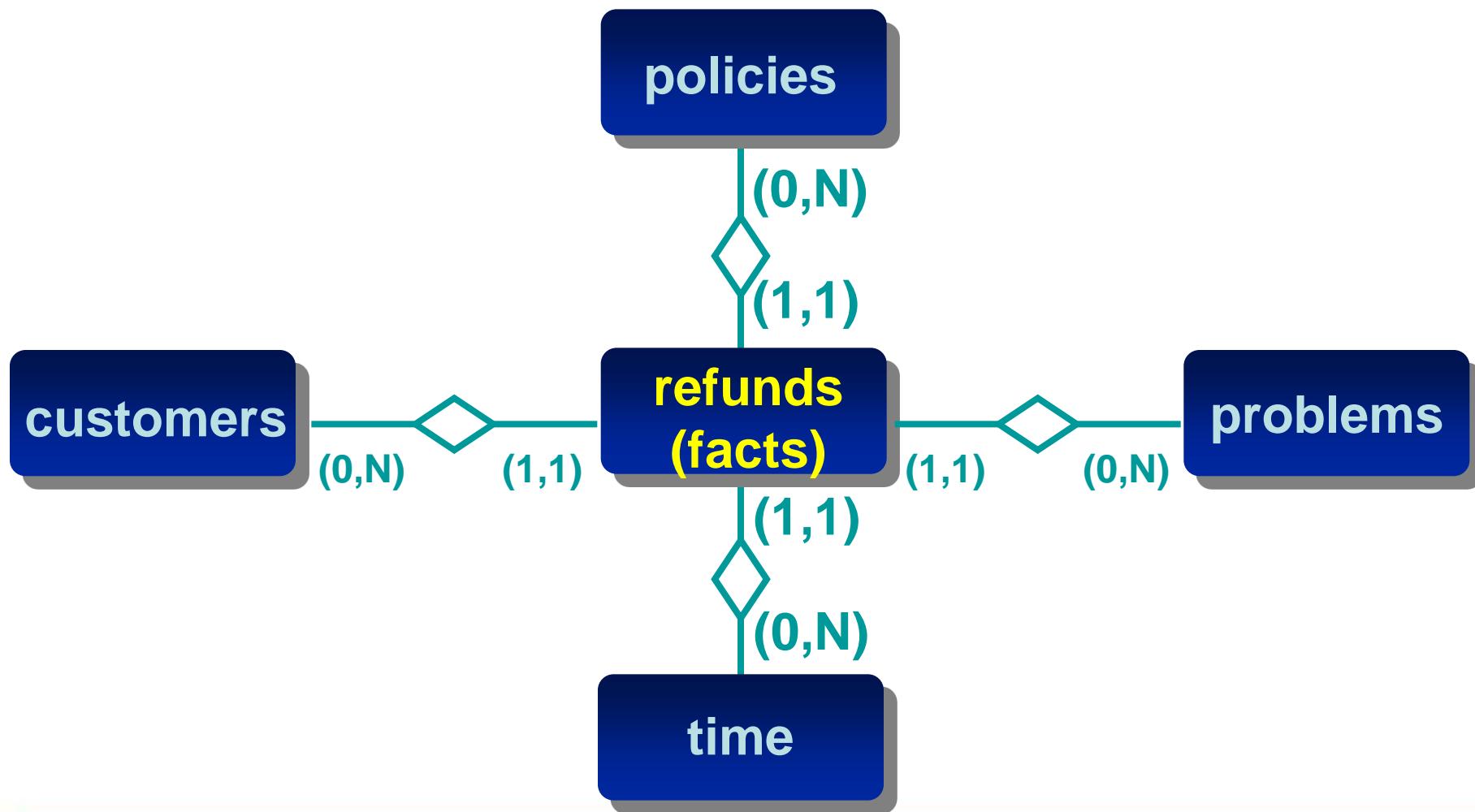
Examples of facts/measures/dimensions

- Retail shops:
 - Sales
 - Quantity, price
 - Product, time, zone
- Telephone service:
 - Phone call
 - Cost, duration
 - Caller, answerer, time

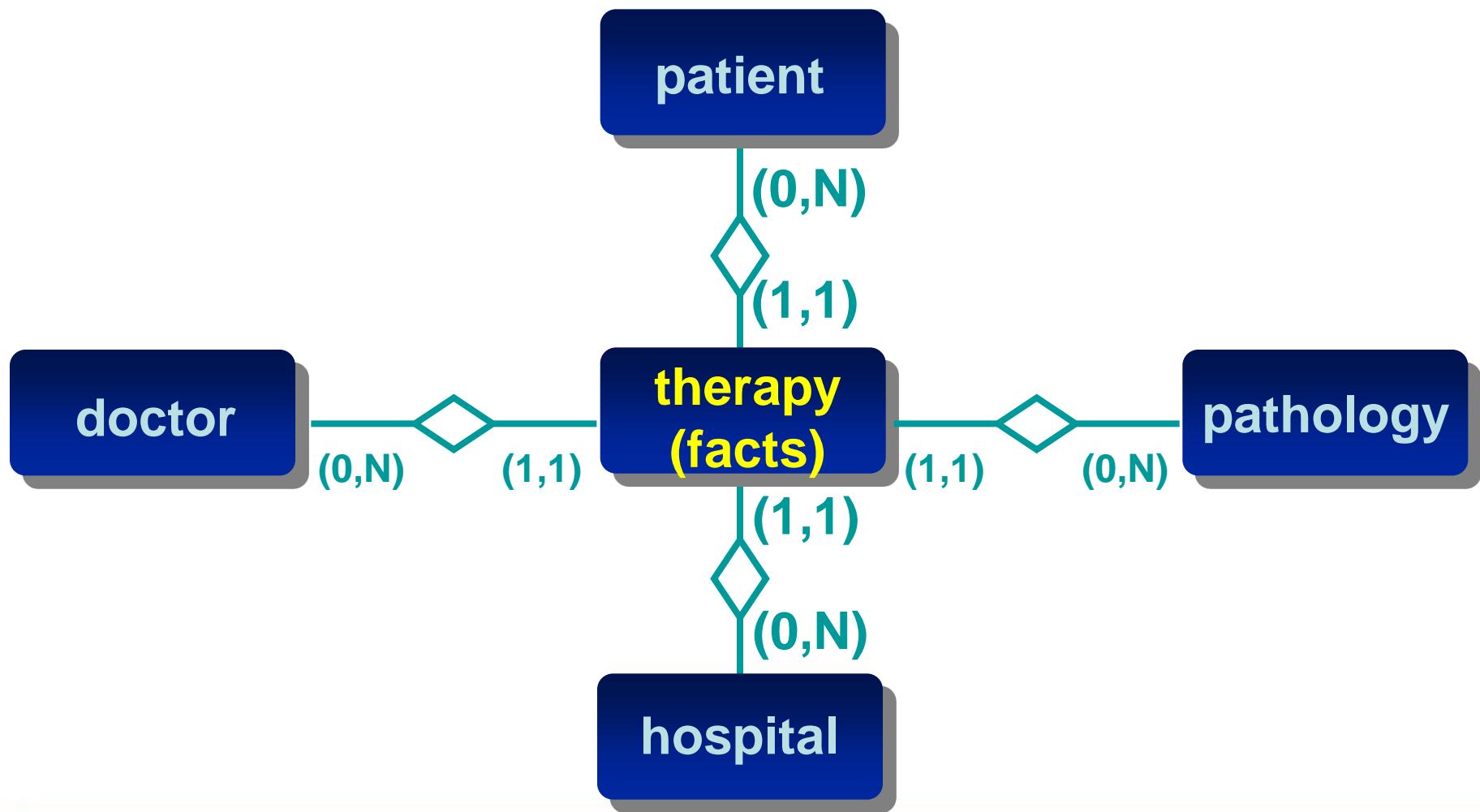
An example: sales management



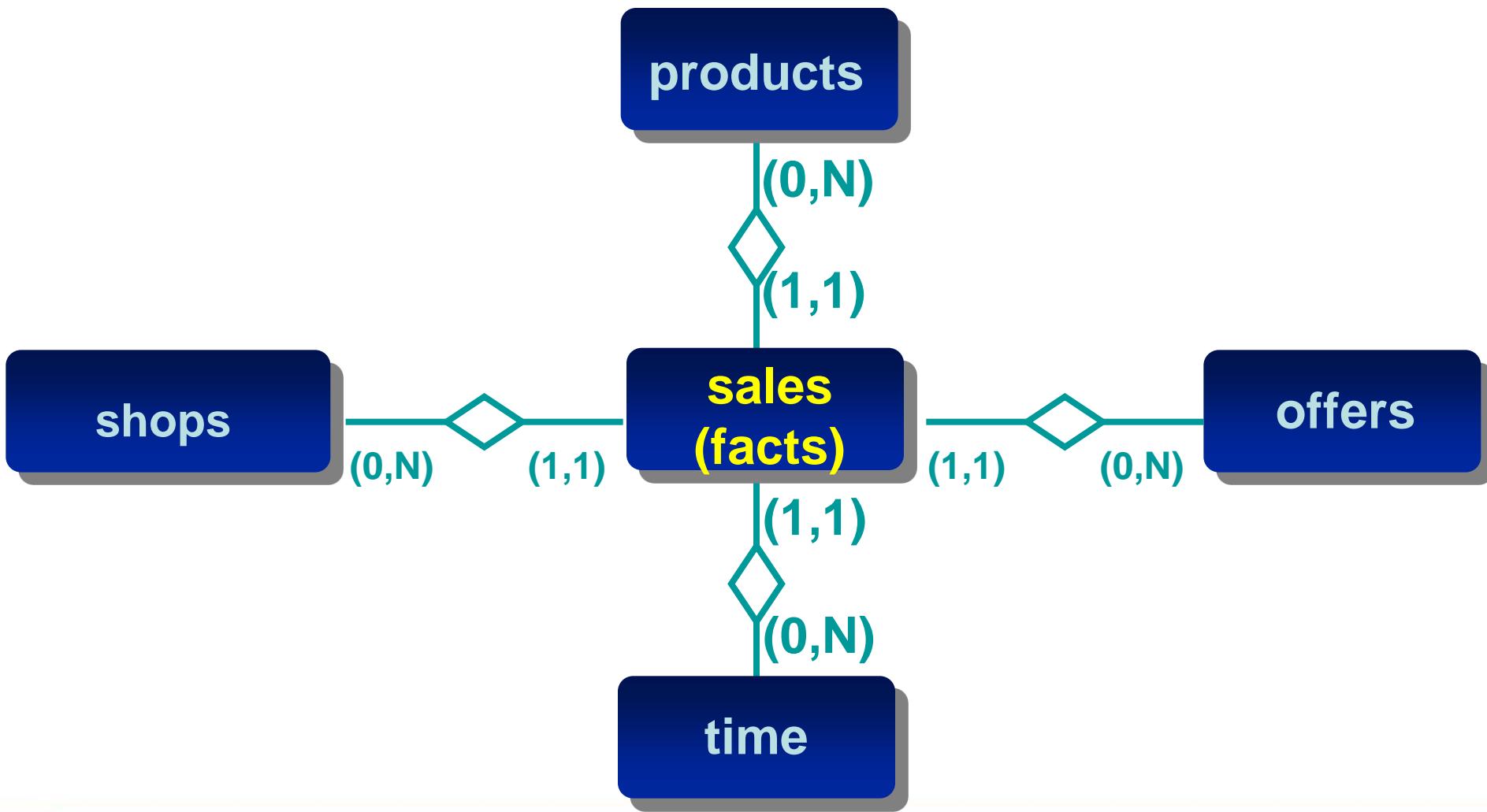
Another example: reimbursements



Another example: therapies



Consider again the sales management schema



Facts: sales

Product-ID

Shop-ID

Time-ID

Offer-ID

Total-proceeds

Quantity

Unit-proceeds

First dimension: products

Product-ID

Category

Sub-Category

Brand

Packing

Weight

Size

Provider

Second dimension: shops

Shop-ID

Name

Address

City

Sales-District

Phone

Manager-Name

Size

Logistics

Third dimension: time

Time-ID

Day-in-Week

Day-in-Month

Day-in-Year

Week-in-Month

Week-in-Year

Month-in-Year

Season

Flag-WorkingDay

Flag-Sunday

Fourth dimension: offers

Offer-ID

Offer-name

Discount-Type

Discount-Percentage

Advertisement

Flag-Coupon

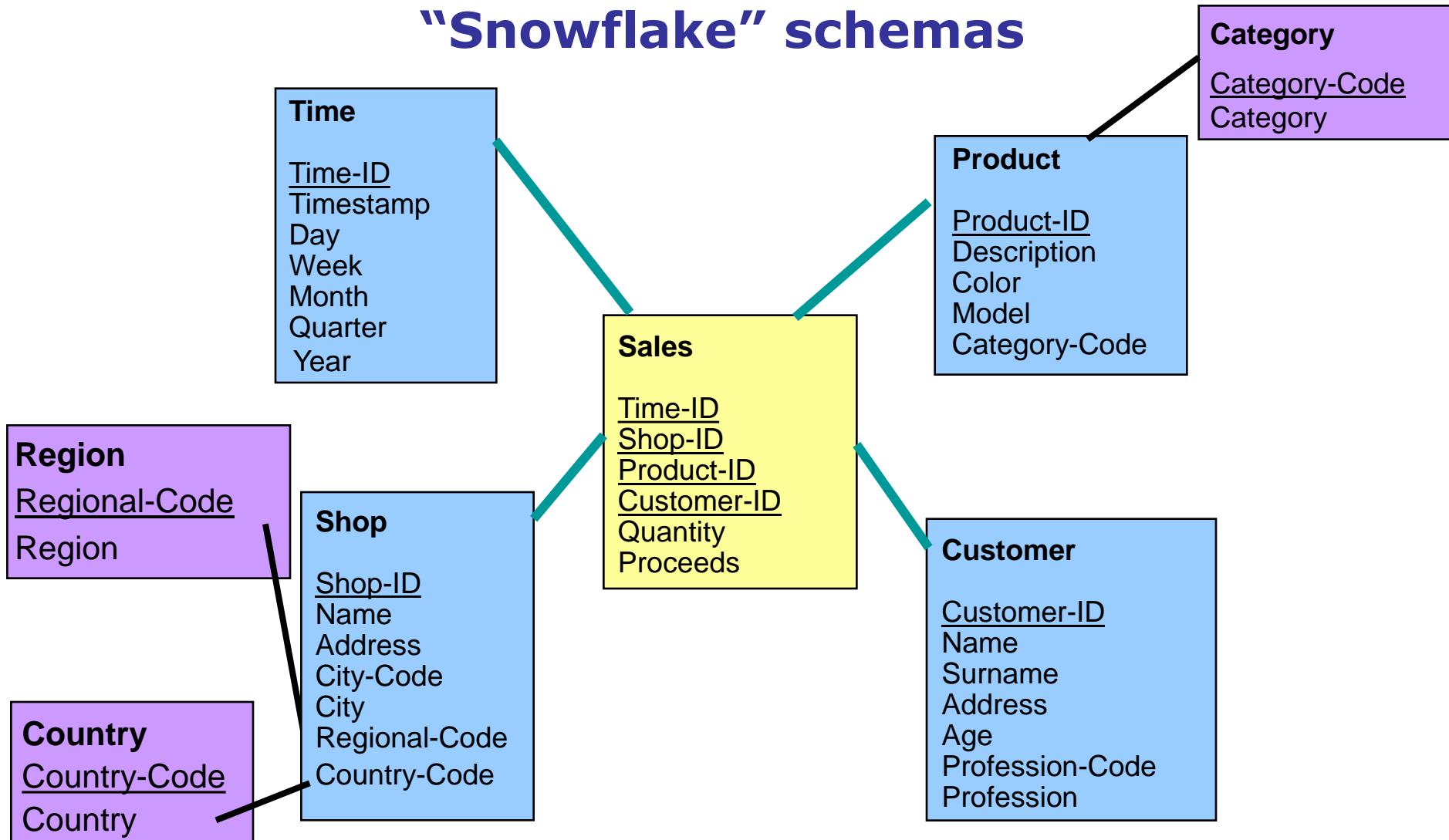
Start-Date

End-Date

Cost

Agency

“Snowflake” schemas



Multi-dimensional data representation

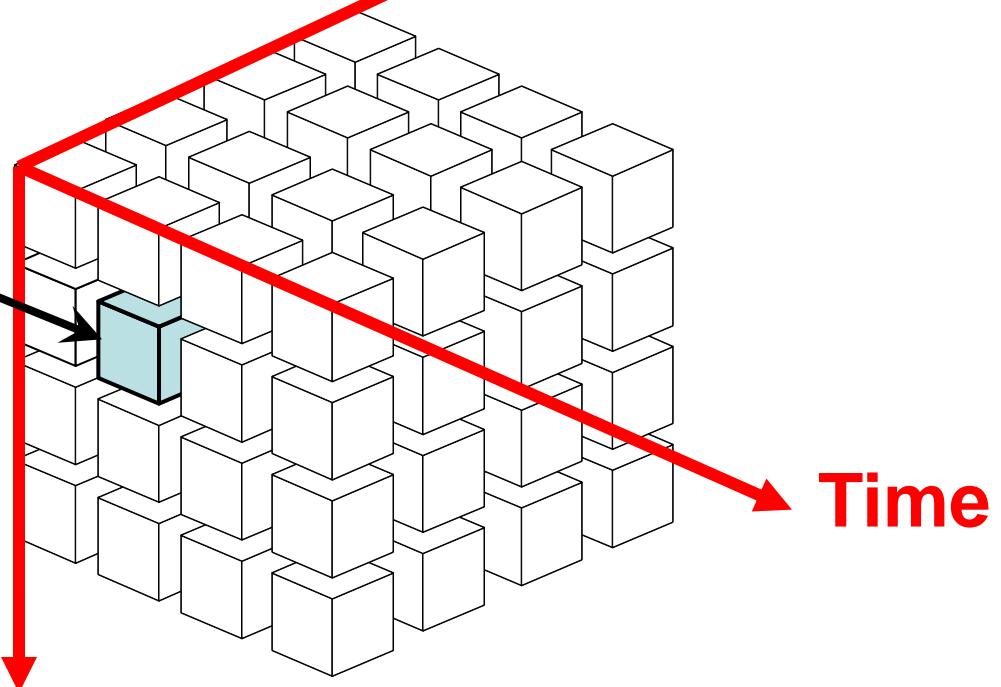
SALES

Quantity

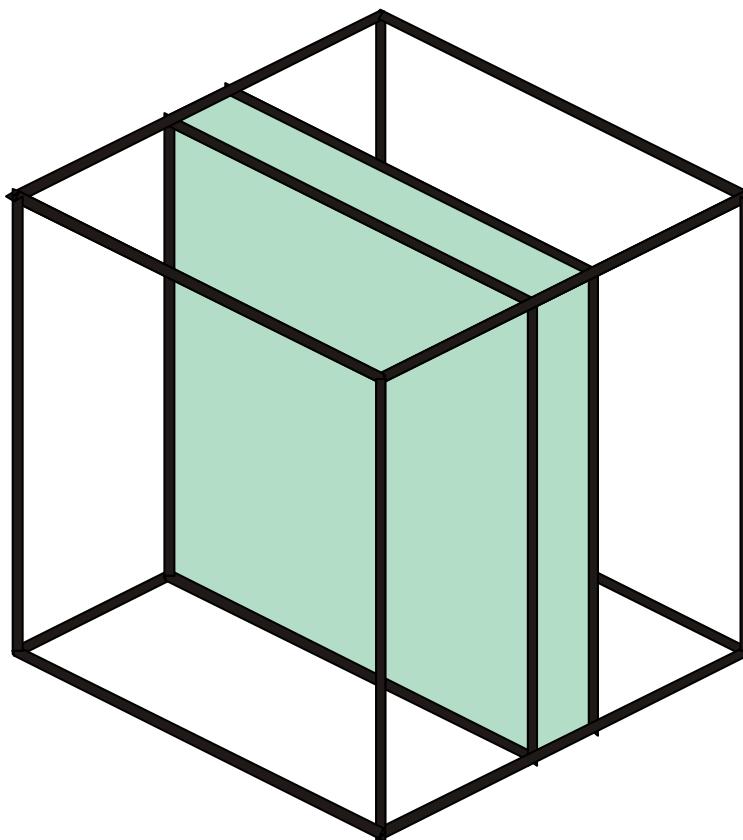
Products

Shops

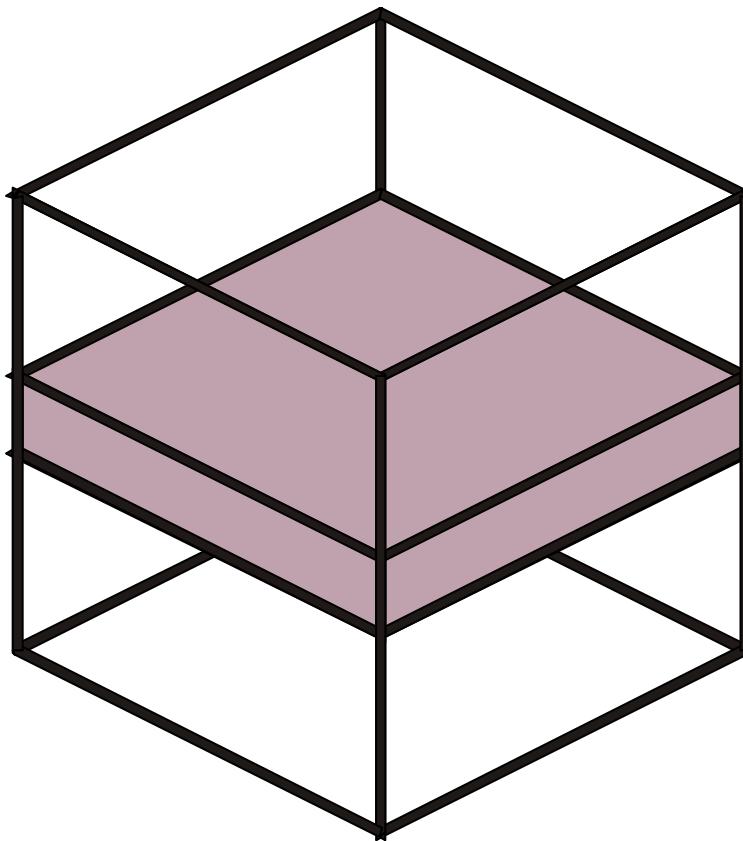
Time



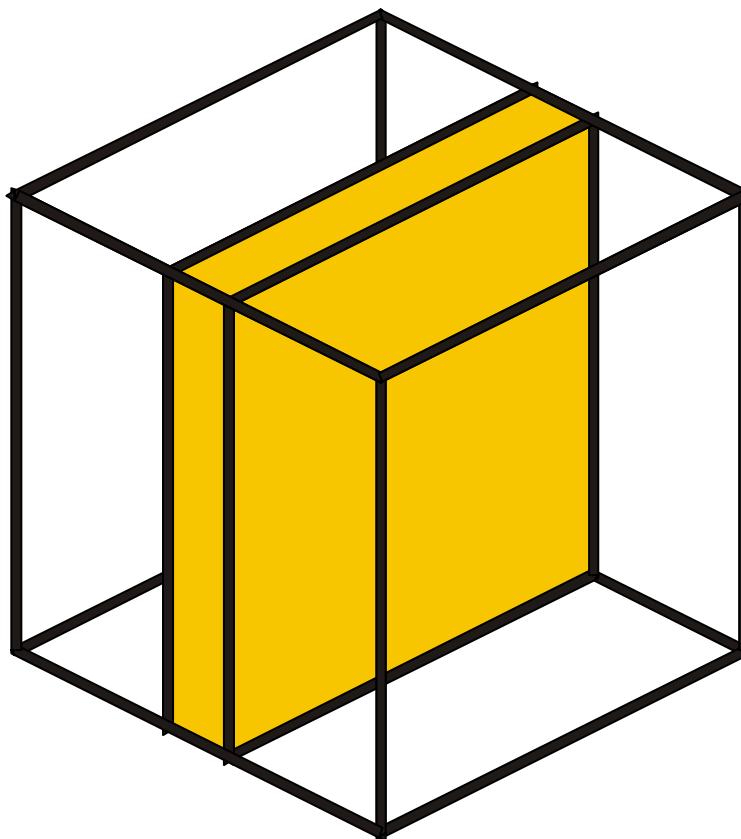
- The regional manager studies the sales of all products in all periods w.r.t. the shops of his region



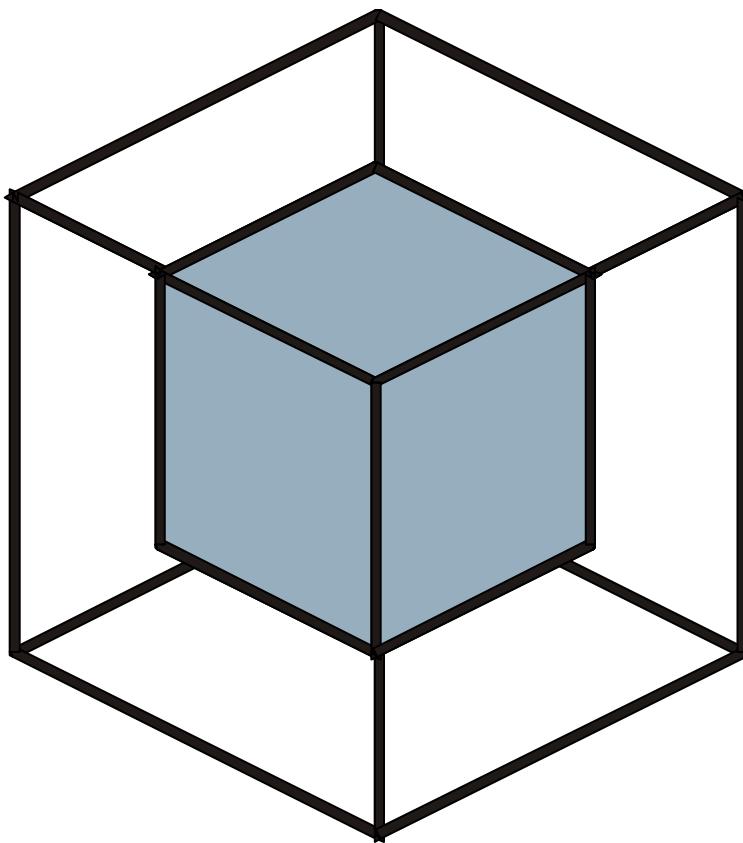
- The product manager studies the sales of a product in all periods and in all shops



- The financial manager studies the sales of all products in all shops, comparing the current period with the previous one



- The strategic manager focuses on one category of products, one area and a limited period of time



Data Visualization

- Data are visualized and rendered graphically, so as to be easy to understand
- Common means of visualization:
 - Tables
 - Pie/doughnut charts
 - Column/bar histograms
 - Line charts
 - 3D surfaces
 - Bubble charts
 - Area blocks
 - Cylinders/cones/pyramids
 - ...

Example of query with a browser

Offer	period	zone	product	dimension of analysis
Pay 2 & buy 3	March	north	milk	total-proceeds
40% discount	April	east	bread	quantity
20% discount	May	west	pasta	unit-proceeds
1-free-mug (...)	
.....				
	February/ April		pasta	sum(proceeds) sum(quantity)

The “same” query in SQL

```
select  c1,  c2,  aggr(c3),  aggr(c4)
from facts, dim1, dim2
where join-predicate(facts, dim1)
      and join-predicate(facts, dim2)
      and selection-predicate(dim1)
      and selection-predicate(dim2)
group by c1, c2
order by c1, c2
```

Result

Month	product	Sum of proceeds	Sum of quantity
February	pasta	110.000	45.000
March	pasta	95.000	50.000
April	pasta	105.000	51.000

Operations over multi-dimensional data

- ***Roll up*** — aggregates data
 - Sums up the sale quantity over last year per each region and product category
- ***Drill down*** — disaggregates data
 - For one particular product category in a region, “unrolls” and shows in detail the sale quantities of each day in each shop
- ***Slice & dice*** — selection and projection
- ***Pivot*** — change the orientation of the data cube

Drill Down: adding one dimension (**Zone**)

Month	Product	Zone	Sum of quantity
February	pasta	north	15.000
February	pasta	east	17.000
February	pasta	west	13.000
March	pasta	north	18.000
March	pasta	east	18.000
March	pasta	west	14.000
April	pasta	north	18.000
April	pasta	east	17.000
April	pasta	west	16.000

Roll-up: removing one dimension (**Month**)

Product	Zone	Sum of quantity
pasta	north	51.000
pasta	east	52.000
pasta	west	43.000

Aggregate queries

- **Examples:**

- Total proceeds for each product category in each shop in each day
- Total monthly proceeds in each shop
- Total monthly proceeds for each product category in each shop
- Average monthly proceeds for each category (calculated over all shops)

Aggregates in SQL: **data cube**

- Expresses all possible aggregations of the tuples of a table
- Uses a new purpose-specific *polymorphic* value: **ALL**

Data cube in SQL

```
select Model, Year,  
       Color, sum( Quantity )  
from Sales  
where Model in ('Fiat', 'Ford')  
      and Color = 'Red'  
      and Year between 1994 and 1995  
group by Model, Year, Color  
with cube
```

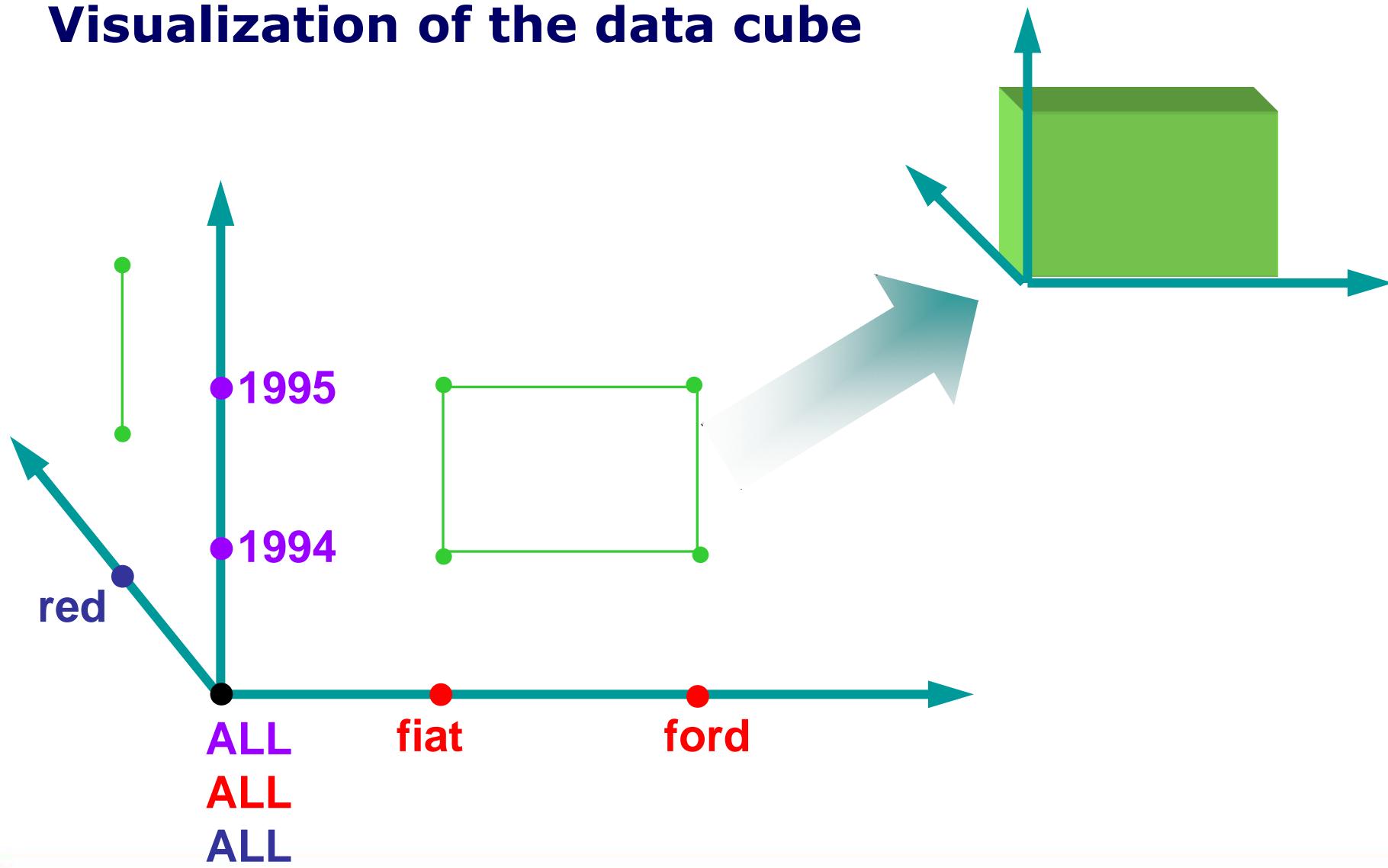
Relevant facts

Model	Year	Color	Quantity
fiat	1994	red	50
fiat	1995	red	85
ford	1994	red	80

Data cube results:

model	year	color	sum (quantity)
fiat	1994	red	50
fiat	1995	red	85
fiat	1994	ALL	50
fiat	1995	ALL	85
fiat	ALL	red	135
fiat	ALL	ALL	135
ford	1994	red	80
ford	1994	ALL	80
ford	ALL	red	80
ford	ALL	ALL	80
ALL	1994	red	130
ALL	1995	red	85
ALL	ALL	red	215
ALL	1994	ALL	130
ALL	1995	ALL	85
ALL	ALL	ALL	215

Visualization of the data cube



Roll up in SQL

```
select Model, Year,  
       Color, sum( Quantity )  
from Sales  
where Model in ('Fiat', 'Ford')  
    and Color = 'Red'  
    and Year between 1994 and 1995  
group by Model, Year, Color  
with roll up
```

Roll up results:

Model	Year	Color	sum(Quantity)
fiat	1994	red	50
fiat	1995	red	85
ford	1994	red	80
fiat	1994	ALL	50
fiat	1995	ALL	85
ford	1994	ALL	80
fiat	ALL	ALL	135
ford	ALL	ALL	80
ALL	ALL	ALL	215

Typical Size of a Data Warehouse

time: 730 days

shops: 300

products: 30.000

daily sales: 3.000

offers: at most one per product on sale

sales: $730 * 300 * 3000 * 1 = 657 \text{ millions}$

Size: $657 \text{ millions} * 8 \text{ attributes} * 4 \text{ Byte} = 21 \text{ GB}$

Classification of OLAP System

- MOLAP (Multi-dimensional OLAP)
 - as alternative to
- ROLAP (Relational OLAP)
 - MOLAP: the internal data storage is not relational, so as to guarantee better performance
 - ROLAP: the relational storage guarantees the capability of managing large volumes of data

Specific OLAP Technologies

- **Bitmap Indexes**

- Allow for efficient evaluation of OR and AND combinations of simple comparison predicates

- **Join Indexes**

- Pre-computed joins between the table of facts and the tables representing the dimensions

- **Materialized views**

- Those views are pre-computed, which can be used to answer most frequently asked queries

Advanced Databases

7

Data Warehouses

Data Mining

Data mining

- Objective
 - Extract information *hidden* into data so as to support strategic decisions
- An inter-disciplinary task (and subject)
 - Statistics
 - Algorithmics
 - Neural networks
 - Fractals
 - ...

Applications of Data Mining

- Market analysis
 - Which products are purchased together or one before another? (basket analysis)
- Analysis of behaviours
 - Identify fraudulent credit card usage
- Forecasts
 - Foreseeing the cost of medical treatments
- Control
 - Industrial production errors

An example: sales analysis

Transaction	Date	Item	Qty	Price
1	12/17/95	ski-pants	1	140 €
1	12/17/95	ski-boots	1	180 €
2	12/18/95	T-shirt	1	25 €
2	12/18/95	jacket	1	300 €
2	12/18/95	ski-boots	1	70 €
3	12/18/95	jacket	1	300 €
4	12/19/95	T-shirt	3	25 €
4	12/19/95	jacket	1	300 €

Association Rules

- Association rules look for *regularity* within data
 - When a customer buys ski-boots, she also buys skis
- Structure of association rules:

Body* \Rightarrow *Head

- *Body*: premise of the rule
- *Head*: consequence of the rule

An example of Association Rule

Diaper \Rightarrow Beer

- 2% of all transactions contain both items
- 30% of transactions containing Diaper also contain Beer

Characteristics of Association Rules

- **Support**
 - Probability that both Head and Body are in the same transaction [$P(H,B)$]
- **Confidence**
 - Probability that the Head is in a transaction t , given that the Body **is** in t [$P(H|B)$, *conditional probability*]
- **Problem statement**
 - Extract from a dataset all association rules with support and confidence over given thresholds

Examples of association rules

Body	Head	Support	Confidence
ski-pants	ski-boots	0.25	1
ski-boots	ski-pants	0.25	1
T-shirt	ski-boots	0.25	0.5
T-shirt	jacket	0.25	1
ski-boots	T-shirt	0.25	0.5
ski-boots	jacket	0.25	1
jacket	T-shirt	0.5	0.66
jacket	ski-boots	0.25	0.33
{ T-shirt, ski-boots }	jacket	0.25	1
{ T-shirt, jacket }	ski-boots	0.25	0.5
{ ski-boots, jacket }	T-shirt	0.25	1

Other Examples

- Items sold in the same special offer
- Items frequently purchased together in summer but not in winter
- Items frequently purchased together as in the shop they are arranged in a particular layout (adjacent, near, ...)
- Items purchased in consecutive transactions by the same customer

Sequential Patterns

- Input dataset:
 - All the transactions of a given customer
- Objective:
 - Find those sequences of items which are frequently contained into corresponding sequences of transactions, such that the frequency is over a given threshold

Examples

- “5% of customers bought a CD player in a transaction and some CDs in the following two transactions”
- “10% of the purchases of a television set is followed by the purchase of a video-recorder”
- Applications
 - Measure of the customer satisfaction
 - Special offers tailored for specific customer classes
 - Medicine (sequences of symptoms \Rightarrow disease)

Discretization

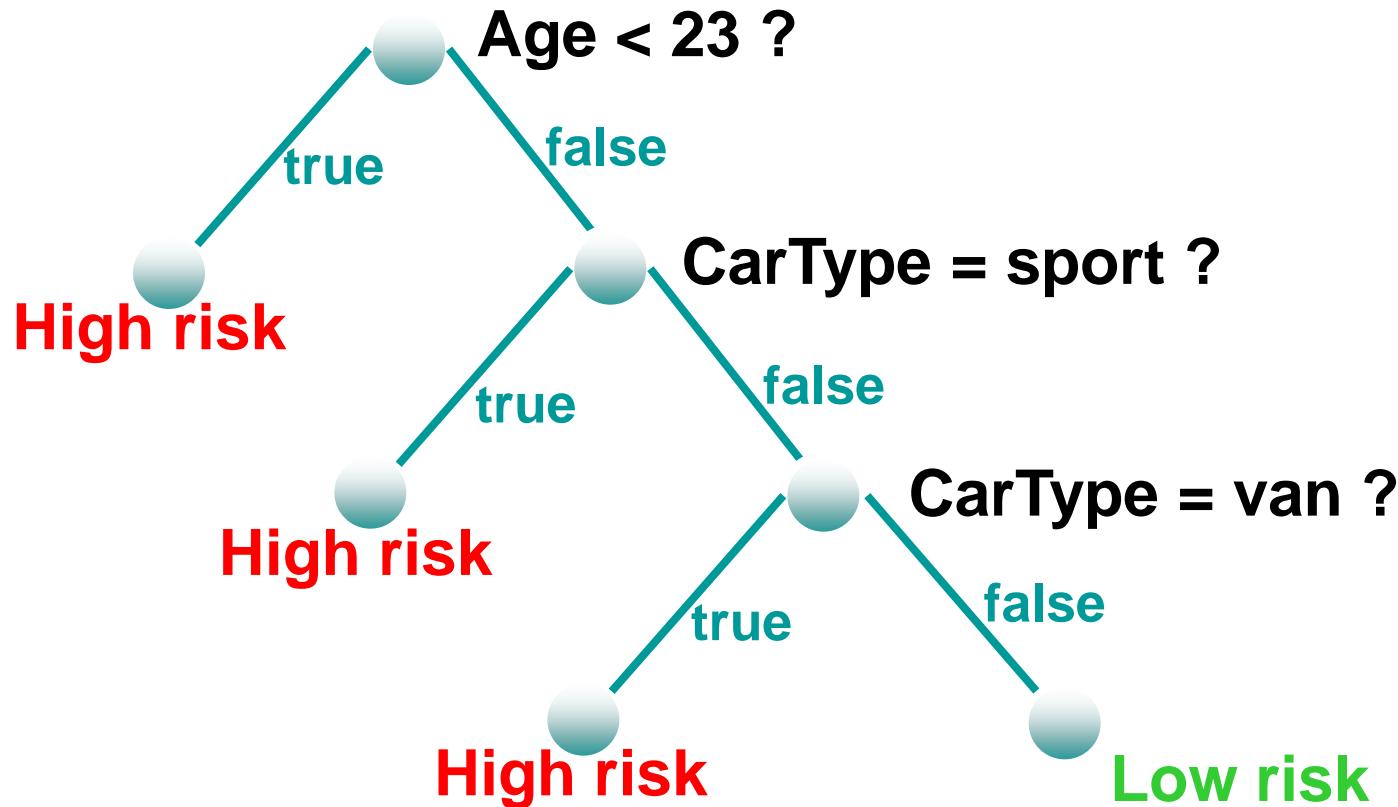
- A continuous domain can be represented by means of a sequence of suitable intervals
 - Example: blood-pressure
 - High: >250
 - Medium: $>130, <250$
 - Low: <130
- *Objective*: find the correlation between the risk of infarction and blood-pressure with a given statistical significance
- *Advantages*:
 - Compact value representation
 - Determination of critical values
 - Facilitation of future data analysis

Classification

- Cataloguing a fact, concept, or phenomenon into a pre-defined class
- The phenomenon is described by elementary facts (atomic data, within **tuples**)
- The classifier is constructed and trained over a set of training data (**training set**)
- Classifiers are represented as **decision-trees**

Example: identify risky policies

POLICY (DrivingLicense, Age, CarType)



Advanced Databases

8

Active Databases

Active Databases

- A database that supports active rules (also called *triggers*)
- Outline:
 - Trigger definition in SQL:1999
 - Trigger definition in DB2 and Oracle
 - Design methods for trigger-based systems
 - Advanced features of triggers
 - Several examples

The Trigger Concept

- Paradigm: Event-Condition-Action
 - When an event occurs
 - If the condition is true
 - The action is executed
- The rule model offers an effective way for representing reactive computations
- Other examples of rules in the DBMS world:
 - Integrity constraints
 - Datalog rules
 - Business rules
- Problem: difficult to realize complex database systems using triggers well

Event-Condition-Action

- **Event**
 - Normally a modification of the state of the database: insert, delete, update
 - When the event occurs, the trigger is *activated*
- **Condition**
 - A predicate identifying the situations in which the application of a trigger is necessary
 - When the condition is evaluated, the trigger is *considered*
- **Action**
 - A generic update statement or a stored procedure
 - When the action is elaborated, the trigger is *executed*
- **DBMSs already provide all the required components. One only needs to integrate them**

Triggers in SQL:1999, Syntax

- SQL:1999 (SQL-3) was strongly influenced by DB2 (IBM); the other systems do not fully comply (as they exist since the mid eighties).
- Each trigger is characterized by:
 - name
 - name of the target (monitored) table
 - execution mode (**before** or **after**)
 - monitored event (**insert**, **delete** or **update**)
 - granularity (statement-level or row-level)
 - names and aliases for transition values and transition tables
 - action
 - creation timestamp

Triggers in SQL:1999, Syntax

```
create trigger TriggerName
  < before | after >
  < insert | delete | update [of Column] > on Table
  [referencing
    <[old table [as] OldTableAlias]
     [new table [as] NewTableAlias] > |
    <[old [row] [as] OldTupleName]
     [new [row] [as] NewTupleName] >]
  [for each < row | statement >]
  [when Condition]
SQLStatements
```

Kinds of events

- BEFORE
 - The trigger is considered and possibly executed before the event (i.e., the database change)
 - Before triggers cannot change the database state; at most they can change (“condition”) the transition variables in row-level mode (set t.new=expr)
 - Normally this mode is used when one wants to check a modification before it takes place, and possibly make a change to the modification itself.
- AFTER
 - The trigger is considered and possibly executed after the event
 - It is the most common mode, suitable for most applications.

Granularity of events

- Statement-level mode (default mode, **for each statement** option)
 - The trigger is considered and possibly executed only once for each statement that activated it, independently of the number of modified tuples
 - Closer to the traditional approach of SQL statements, which normally are set-oriented
- Row-level mode (**for each row** option)
 - The trigger is considered and possibly executed once for each tuple modified by the statement
 - Writing of row-level triggers is simpler

The referencing clause

- The format depends on the granularity
 - For row-level mode, there are two *transition variables* (`old` and `new`) that represent the value either prior to or following the modification of the tuple under consideration
 - For statement-level mode, there are two *transition tables* (`old table` and `new table`) that contain either the old or the new value of all modified tuples
- Variables `old` and `old table` can't be used with triggers whose event is `insert`
- Variables `new` and `new table` can't be used with triggers whose event is `delete`
- Transition variables and transition tables enable tracking the changes that activate a trigger

Example of a Row-level Trigger

```
create trigger AccountMonitor
  after update on Account
  for each row
  when new.Total > old.Total
  insert values
    (new.AccNumber,new.Total-old.Total)
  into Payments
```

Example of a Statement-level Trigger

```
create trigger FileDeletedInvoices  
after delete on Invoice  
referencing old_table as OldInvoiceSet  
insert into DeletedInvoices  
(select *  
from OldInvoiceSet)
```

Execution of Multiple Triggers: Conflicts between Triggers

- If several triggers are associated to the same event, SQL:1999 prescribes the following policy
 - BEFORE triggers (statement-level and row-level) are executed
 - The modification is applied and the integrity constraints defined on the DB are checked
 - AFTER triggers (row-level and statement level) are executed
- If there are several triggers belonging to the same category, the order of execution chosen by the system is based upon their definition timestamp (older triggers have higher priority).

Recursive Execution Model

- SQL:1999 states that triggers are handled within a Trigger Execution Context (TEC)
- The execution of a trigger action may produce events that activate other triggers, which will have to be evaluated within a new, internal TEC.
 - At this point, the state of the enclosing TEC is saved, and the enclosed TEC is executed; this is a recursive process.
 - At the end of the enclosed TEC's execution, the state of the enclosing TEC is recovered and its execution is resumed.
- Trigger execution halts after a given recursion depth by rising a "nontermination exception"
- Any failure during a chain of trigger activated by a given statement S causes the partial rollback of S and of all changes due to the triggers of the chain.

Example: Salary Management

Employee

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	56.000	1	10
52	Jones	50.000	1	20

Department

DeptNum	MGRRegNum
1	50

Project

ProjNum	Objective
10	NO
20	NO

Example Trigger T1: Bonus

Event: update of Objective in Project
Condition: Objective = 'YES'
Action: Increase by 10% the salary of the employees involved in the project

```
CREATE TRIGGER Bonus
AFTER UPDATE OF Objective ON Project
FOR EACH ROW
WHEN NEW.Objective = 'YES'
BEGIN
    update Employee
        set Salary = Salary*1.10
        where ProjN = NEW.ProjNum;
END;
```

Example Trigger T2: CheckIncrement

- Event:** update of Salary in Employee
Condition: New salary greater than manager's salary
Action: Decrease salary and make it the same as the manager's

```
CREATE TRIGGER CheckIncrement
AFTER UPDATE OF Salary ON Employee
FOR EACH ROW DECLARE X number;
BEGIN
    SELECT Salary into X
    FROM Employee JOIN Department
    ON Employee.RegNum = Department.MGRRegNum
    WHERE Department.DeptNum = NEW.DeptN;
    IF NEW.Salary > X
        update Employee set Salary = X
        where RegNum = NEW.RegNum;
    ENDIF;
END;
```

Example Trigger T3: CheckDecrement

- Event:** update of Salary in Employee
Condition: Decrement greater than 3%
Action: Decrement salary only by 3%

```
CREATE TRIGGER CheckDecrement
AFTER UPDATE OF Salary ON Employee
FOR EACH ROW
WHEN (NEW.Salary < OLD.Salary * 0.97)
BEGIN
    update Employee
    set Salary=OLD.Salary*0.97
    where RegNum = NEW.RegNum;
END ;
```

Activation of T1

```
update Project
set Objective = 'yes' where ProgNum = 10
```

Event: update of the
Objective attr. in Project

Cond.: true

Project	
ProjNum	Objective
10	yes
20	no

Action: increase Black's salary by 10%

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	61.600	1	10
52	Jones	50.000	1	20

Activation of T2

Event: update of `Salary` in `Employee`

Condition: true (employee Black's salary is greater than manager Smith's salary)

Action: Black's salary is modified and set to Smith's salary

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	59.000	1	10
52	Jones	50.000	1	20

- T2 is activated again – the condition is false
- T3 is activated

Activation of T3

Event: update of `Salary` in `Employee`

Condition: true (Black's salary was decreased by more than 3%)

Action: Black's salary is decreased by only 3%

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	59.752	1	10
52	Jones	50.000	1	20

- T2 is activated again – the condition is true

Activation of T2

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	59.000	1	10
52	Jones	50.000	1	20

Activation of T3

- The trigger condition is false
 - Salary was decreased by less than 3%
- Trigger activation has reached termination

Triggers in DB2

- They follow the SQL:1999 syntax and semantics
- Some examples:
- 1. “Conditioner” (acts before the update and integrity checking)

```
create trigger LimitaAumenti  
before update of Salario on Impiegato  
for each row  
when (New.Salario > Old.Salario * 1.2)  
set New.Salario = Old.Salario * 1.2
```

- 2. “Re-installer” (acts after the update)

```
create trigger LimitaAumenti  
after update of Salario on Impiegato  
for each row  
when (New.Salario > Old.Salario * 1.2)  
set New.Salario = Old.Salario * 1.2
```

Triggers in Oracle

- They follow a different syntax (multiple events allowed, no table variables, where clause only legal with row-level triggers)

```
create trigger TriggerName
  { before | after } event [, event [,event ]]
  [[referencing
    [old [row] [as] OldTupleName]
    [new [row] [as] NewTupleName] ]
  for each { row | statement } [when Condition]]
  SQLStatements
```

Event ::= { **insert** | **delete** | **update** [**of Column**] } on *Table*

- They have also a rather different conflict semantics and no limitation on expressive power of before trigger's action.

Conflicts between Triggers in Oracle

- If several triggers are associated to the same event, ORACLE has the following policy:
 - BEFORE statement-level triggers are executed
 - BEFORE row-level triggers are executed
 - The modification is applied and the integrity constraints defined on the DB are checked
 - AFTER row-level triggers are executed
 - AFTER statement-level triggers are executed
- If there are several triggers belonging to the same category, the order of execution depends on the creation time of the trigger.
- “Mutating table exception”: occurs when the chain of triggers activated by a before trigger T tries to change the state of T’s target table. Forces a statement rollback.

Example of Trigger in Oracle

Event: update of AvailableQty in Warehouse
Condition: Quantity below threshold and no pending order
Action: Do the order

```
create trigger Reorder
after update of AvailableQty on Warehouse
when (new.AvailableQty < new.ThresholdQty)
for each row
declare
  X number;
  select count(*) into X
  from PendingOrder
  where Part = new.Part;
  if X = 0
  then
    insert into PendingOrder
      values(new.Part,new.ReorderQty,sysdate)
  end if;
end;
```

Design - Trigger Properties

- It's important to ensure that interferences among triggers and chain activations do not produce anomalies in the behavior of the system
- 3 classical properties
 - **Termination**: for any initial state and sequence of modifications, a final state is produced (no infinite activation cycles)
 - **Confluence**: triggers terminate and produce a unique final state, independent of the order in which triggers are executed
 - **Determinism of observable behavior**: triggers are confluent and produce the same sequence of messages
- Termination is by far the most important property

Termination Analysis

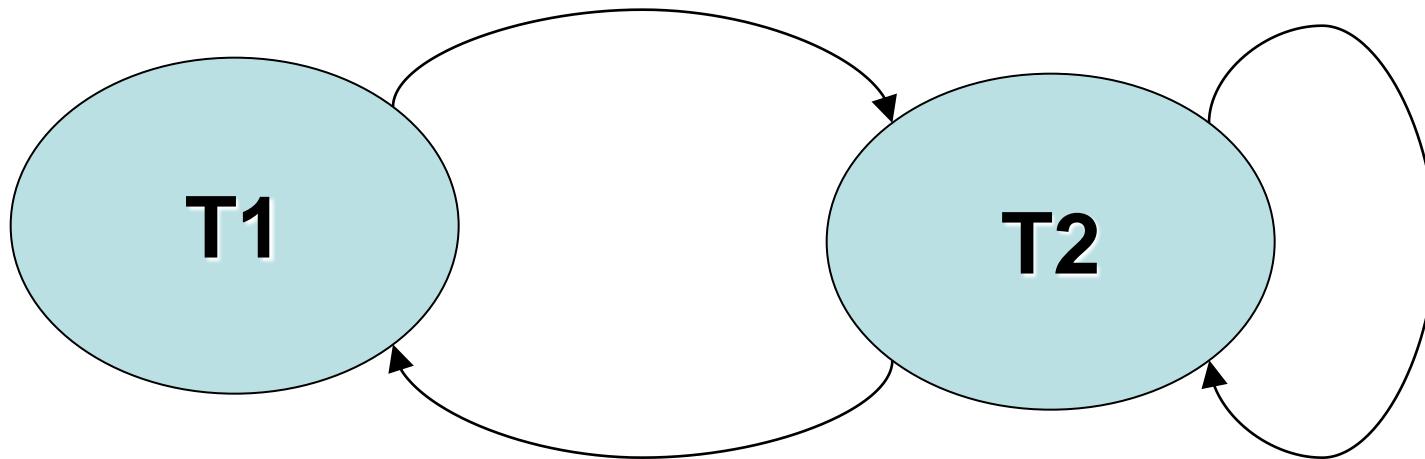
- There are several conceptual tools, most of which graph based
- The simplest is the **triggering graph**
 - A node for each trigger
 - An arc from a node t_i to a node t_j if the execution of t_i 's action can activate trigger t_j (can be done with a simple syntactic analysis)
- If the graph is acyclic, the system is guaranteed to terminate
 - There cannot be infinite trigger sequences
- If the graph has some cycles, it *may* be non-terminating (but it might as well be terminating)

Example with Two Triggers

```
T1:  create trigger AdjustContributions
      after update of Salary on Employee
      referencing new table as NewEmp
      update Employee
      set Contribution = Salary * 0.8
      where RegNum in (    select RegNum
                           from NewEmp)

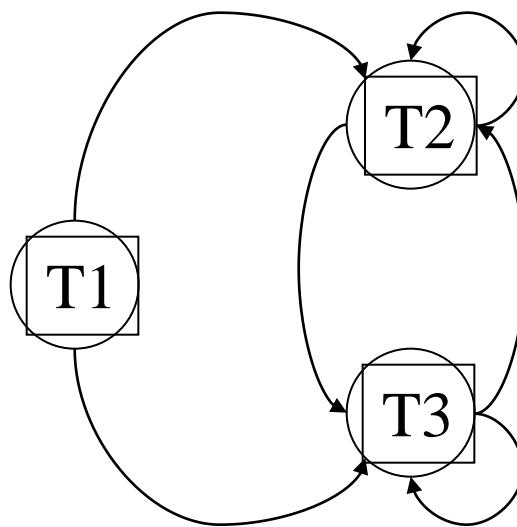
T2:  create trigger CheckBudgetThreshold
      after update on Employee
      when 50000 < (select sum(Salary+Contribution)
                      from Employee)
      update Employee
      set Salary = 0.9*Salary
```

Triggering Graph for Previous Triggers



- There are two cycles, but the system is terminating.
- To make it non-terminating, it suffices to reverse the direction of the comparison in T2's condition or to multiply by a factor greater than 1 in T2's action.

Termination Graph for the Salary Management Triggers



- The graph is cyclic, but the repeated execution of the triggers reaches termination anyway

Problems in Designing Trigger Applications

- Triggers add data management functions in a transparent and reusable manner
 - Such functions would otherwise be distributed over all applications
- But understanding the interactions between triggers is rather complex
- DBMS vendors use triggers to realize internal services, by introducing mechanisms for their automatic generation
 - Examples:
 - Constraint management
 - Data replication
 - View maintenance

Applications of Active Databases

- **Internal** rules (generated by the system and not visible by the user)
 - Integrity constraint management
 - Computation of derived and replicated data
 - Versioning management, privacy, security,
 - Action logging, event recording
- **External** rules (generated by the database administrator, they express application-specific knowledge)
 - Personalization, adaptation
 - Context-awareness
 - Business rules

Referential Integrity Management

- *Repair strategies* for violations of referential integrity constraints
 - The constraint is expressed as a predicate in the condition part

Ex: `CREATE TABLE Employee (`

`FOREIGNKEY (DeptN) REFERENCES Department (DeptNum)`
 `ON DELETE SET NULL,`
 ) ;

- Operations that can violate this constraint:
 - `INSERT` in `Employee`
 - `UPDATE` of `Employee.DeptN`
 - `UPDATE` of `Department.DeptNum`
 - `DELETE` in `Department`

Actions in the Employee Table

- Event:** insert in `Employee`
- Condition:** the new `DeptN` value is not among those in the `Dept` table
- Action:** insertion is inhibited, reporting error

```
CREATE TRIGGER CheckEmpDept
BEFORE INSERT ON Employee
FOR EACH ROW
WHEN (not exists select * from Department
      where DeptNum = NEW.DeptN)
BEGIN raise_application_error(-20000, 'Invalid
Department'); END;
```

- For the update of `DeptN` in `Employee`, the trigger only changes in the event part

Deletion in the Department Table

Event: delete in `Department`

Condition: the deleted `DeptNum` is used in the `Employee` table

Action: Set null policy (the employee's dept is set to null)

```
CREATE TRIGGER CheckDeptDeletion
AFTER DELETE ON Department
FOR EACH ROW
WHEN (exists select * from Employee
      where DeptN = OLD.DeptNum)
BEGIN
    UPDATE Employee
    SET DeptN=NULL
    WHERE DeptN = OLD.DeptNum;
END ;
```

(note: condition could be omitted)

Updates in the Department Table

- Event:** update of `DeptNum` in `Department`
Condition: the new `DeptNum` value is used in the `Employee` table
Action: cascade policy (`DeptN` in `Employee` is also modified)

```
CREATE TRIGGER CheckDeptUpdate
AFTER UPDATE OF DeptNum ON Department
FOR EACH ROW
WHEN (exists select * from Employee
      where DeptN = OLD.DeptNum)
BEGIN
    update Employee set DeptN = NEW.DeptNum
    where DeptN = OLD.DeptNum;
END ;
```

(note: condition could be omitted)

Triggers for Materialized View Maintenance

- Consistency of views wrt tables on which they are defined
 - Base table updates must be propagated to views
- Replication management:

```
CREATE MATERIALIZED VIEW EmployeeReplica  
REFRESH FAST AS  
SELECT * FROM  
DBMaster.Employee@mastersite.world;
```

- Materialized view maintenance is managed via triggers

Recursion Management

- Triggers for recursion management
 - Recursion not yet supported by current DBMSs
- Ex.: representation of a hierarchy of products
 - Each product is characterized by a **super-product** and by a depth **level** in the hierarchy
 - Can be represented by a recursive view (**with recursive** construct in SQL:1999)
 - Alternatively: use triggers to build and maintain the hierarchy

Product (Code, Name, Description, SuperProduct, Level)

- Hierarchy represented by **SuperProduct** and **Level**
- Products not contained in other products:
SuperProduct=NULL and **Level=0**

Deletion of a Product

- In case of product deletion, all its sub-products must be deleted as well

```
CREATE TRIGGER DeleteProduct
AFTER DELETE ON Product
FOR EACH ROW
BEGIN
    delete from Product
    where SuperProduct = OLD.Code;
END;
```

Insertion of a New Product

- In case of an insertion, the appropriate Level value must be calculated

```
CREATE TRIGGER ProductLevel
AFTER INSERT ON Product FOR EACH ROW
BEGIN
    IF NEW.SuperProduct IS NOT NULL
        UPDATE Product
            SET Level = 1 + (select Level from Product
                                where Code = NEW.SuperProduct)
    ELSE
        UPDATE Product
            SET Level = 0
            WHERE Code = NEW.Code;
    ENDIF;
END;
```

Access Control

- Triggers can be used to strengthen access control
- It is convenient to define only those triggers that correspond to conditions that can't be directly verified by the DBMS
- Using BEFORE gives the following advantages
 - Access control is executed before the trigger event is executed
 - Access control is executed only once and not for each tuple on which the trigger event is verified

ForbidSalaryUpdate Trigger

```
CREATE TRIGGER ForbidSalaryUpdate
BEFORE INSERT ON Employee
DECLARE
    not_weekend EXCEPTION; not_workingHours EXCEPTION;
BEGIN
/*if weekend*/
    IF (to_char(sysdate, 'dy') = 'SAT'
        OR to_char(sysdate, 'dy') = 'SUN')
    THEN RAISE not_weekend;
    END IF;
/*if outside working hours (8-18) */
    IF (to_char(sysdate, 'HH24') < 8
        OR to_char(sysdate, 'HH24') > 18)
    THEN RAISE not_workingHours;
    END IF;
```

ForbidSalaryUpdate Trigger (cont'd)

EXCEPTION

WHEN not_weekend

THEN raise_application_error(-20324, 'cannot
modify Employee table during week-end') ;

WHEN not_workingHours

THEN raise_application_error(-20325, 'cannot
modify Employee table outside working
hours') ;

END ;

Evolution of active databases

- Execution modes (immediate, deferred, detached)
- Rule administration: priorities, grouping, dynamic activation and deactivation
- Instead-of clause
- New rule events (system-defined, temporal, user-defined)
- Complex events and event calculus
- The big news: streams.

Execution Modes

- The execution mode describes the connection between the activation (event) and the consideration and execution phases (condition and action)
- Condition and action are always evaluated together
- Normally the trigger is Immediate: considered and executed with the activating event
- Alternative execution modes:
 - Deferred: the trigger is handled at the end of the transaction
 - Example: triggers that check satisfaction of integrity constraints that require the execution of several operations
 - Detached: the trigger is handled in a separate transaction
 - Example: efficient management of variations of stock indices values after several exchanges

Priorities, Activations, and Groups

- Definition of priority
 - Allows specifying the execution order of triggers when there are several triggers activated at the same time
 - SQL:1999 states an order based on the execution mode and granularity; when these coincide, the choice depends on the implementation
- Activation/deactivation of triggers
 - Not in the standard, but often available
- Organization of triggers in groups
 - Some systems offer trigger grouping mechanisms, so as to activate/deactivate by groups

Instead of clause

- Alternative to BEFORE and AFTER
- Another operation than the one that activated the event is executed
- Very dangerous semantics (the application does one thing, the system does another thing)
- Implemented in several systems, often with strong limitations
 - In Oracle it can only be used for updates on views, so as to solve the view update problem when there is ambiguity

Extended Events/1

- System events and DDL commands
 - System: server-error, shutdown, etc.
 - DDL: authorization updates
 - In both cases some DBMSs already have these services that perform complex monitoring
- Temporal events (also periodical events)
 - Example: on July 23rd 2006 at 12, every day at 4
 - Useful for several applications
 - Difficult to integrate them because they are in an autonomous transactional context
 - They can be simulated via software components outside the DBMS that use time management services from the operating system

Extended Events/2

- “User-defined” events
 - Example: “TemperatureTooHigh”
 - Useful in some applications, but normally not offered
 - They too can be easily simulated
- Queries
 - Example: who reads the salaries
 - Normally too heavy to handle

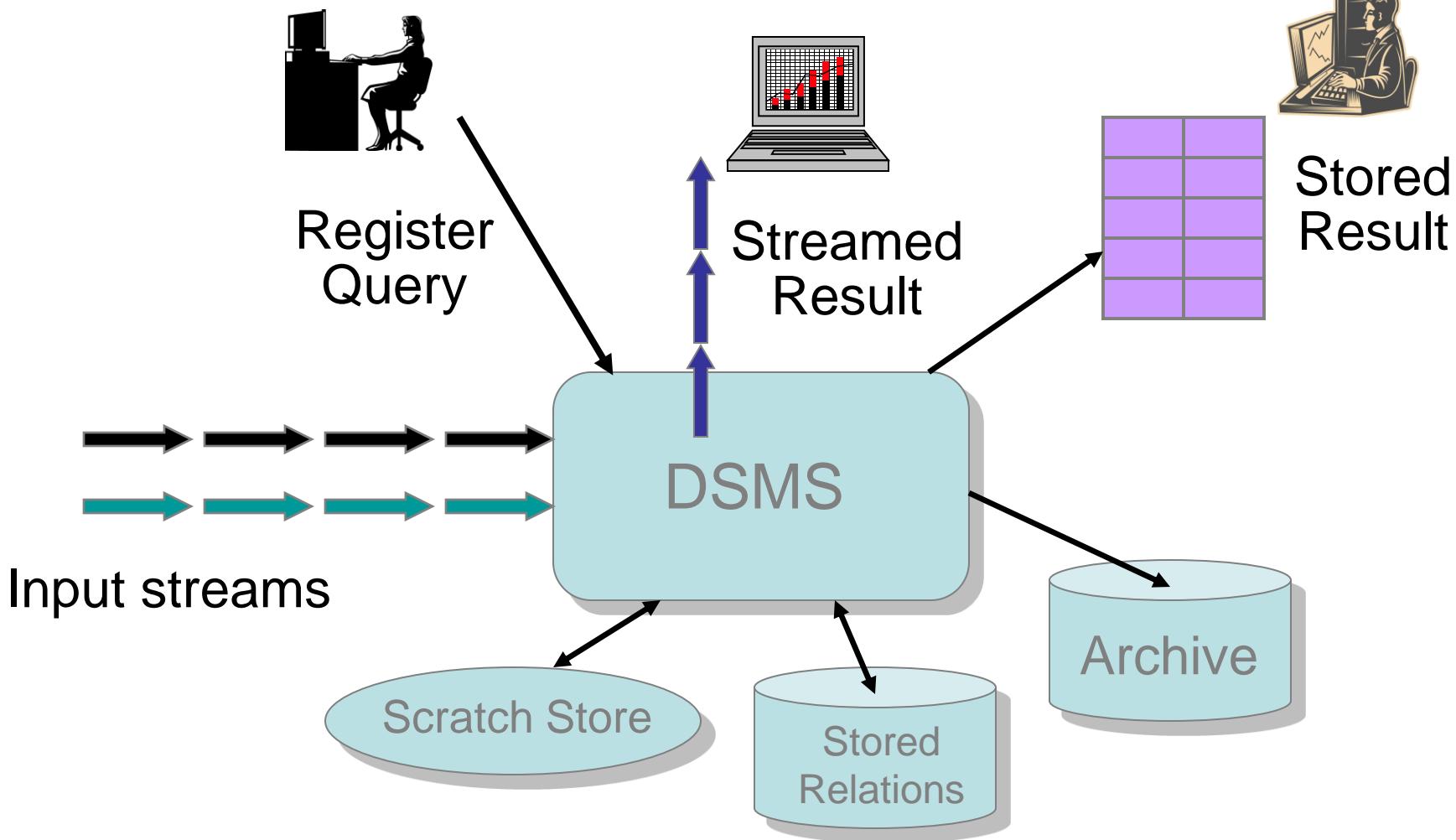
Event expressions

- Boolean combinations of events
 - SQL:1999 allows the specification of several events for a trigger, in disjunction
 - Any event among these is sufficient
 - Some researchers proposed more complex composition models
 - Very complex to handle
 - No strong motivation for introducing them

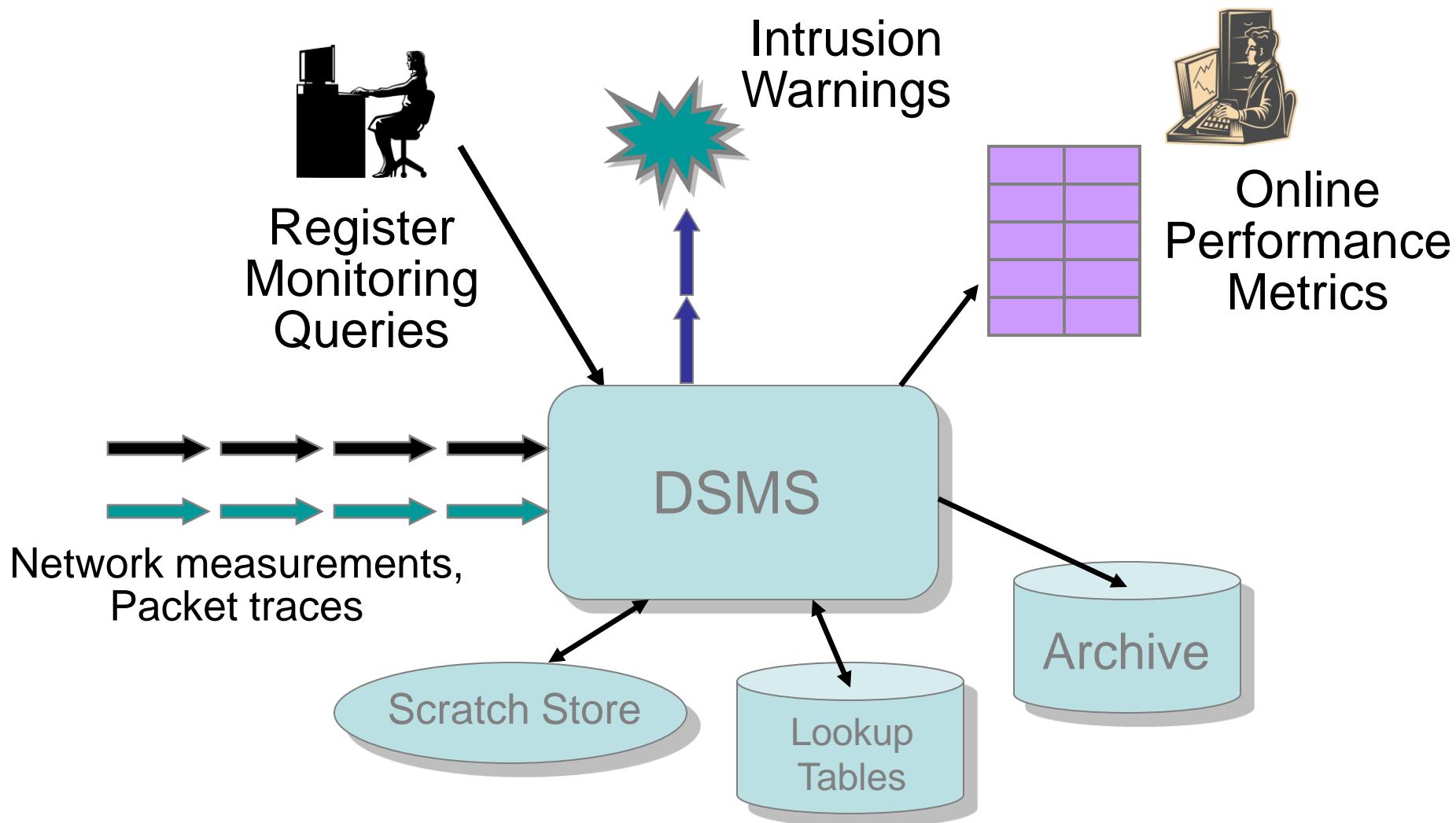
Data Streams

- Traditional DBMS -- data stored in finite, persistent data sets
- New applications -- data as multiple, continuous, rapid, time-varying data streams
 - Network monitoring and traffic engineering
 - Security applications
 - Telecom call records
 - Financial applications
 - Web logs and click-streams
 - Sensor networks
 - Manufacturing processes

The (Simplified) Big Picture



(Simplified) Network Monitoring



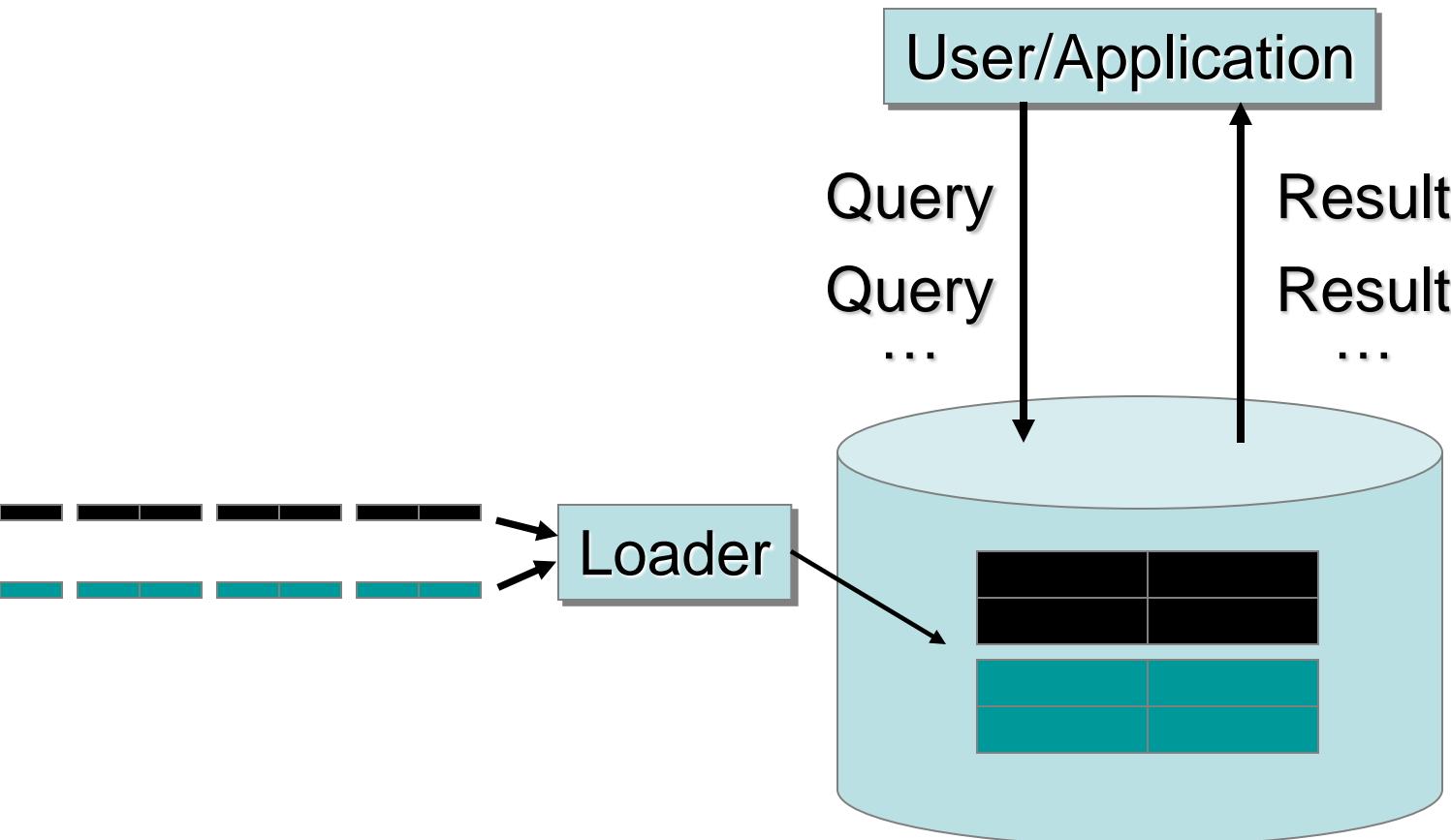
Why they are related to active databases?

- In an active database: the event is a query, the trigger system starts a reaction
- In a stream database: the event is a flow of data, the data stream system starts a reaction
 - Data streams enable the massive control of multiple data-driven events

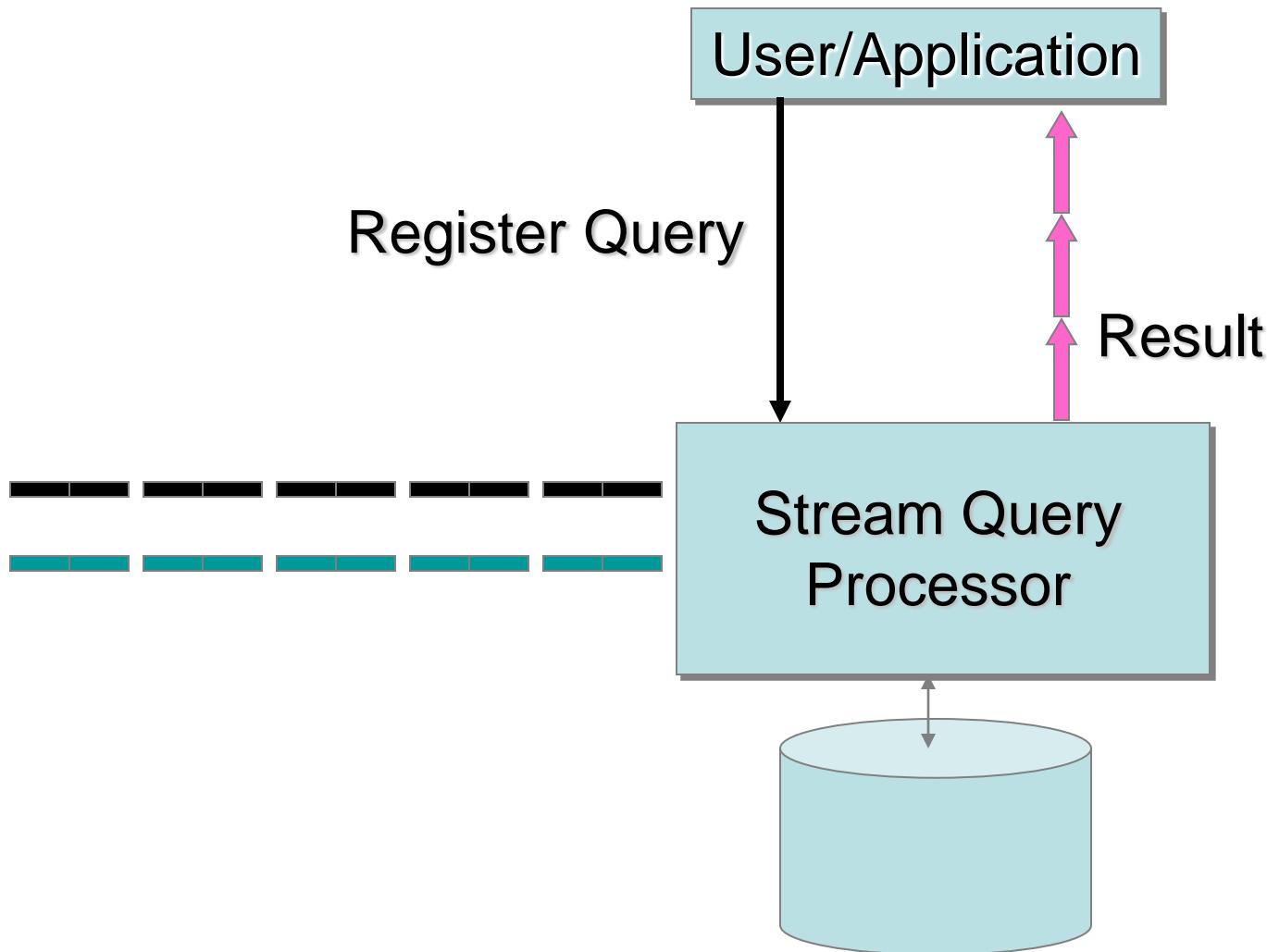
Challenges

- Multiple, continuous, rapid, time-varying streams of data
- Queries may be continuous (not just one-time)
 - Evaluated continuously as stream data arrives
 - Answer updated over time
- Queries may be complex
 - Beyond element-at-a-time processing
 - Beyond stream-at-a-time processing

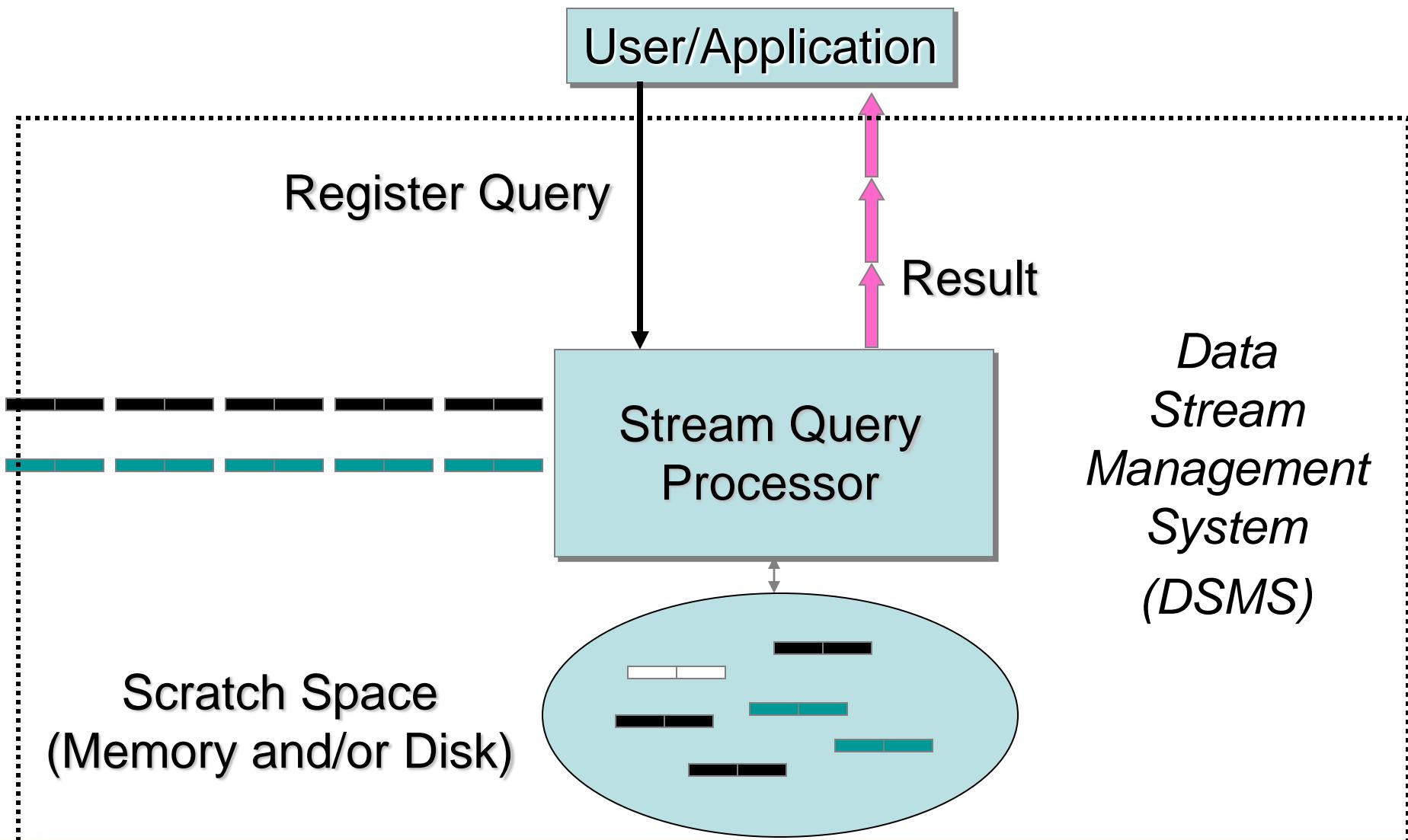
Using Traditional Databases



New Approach for Data Streams



New Approach for Data Streams



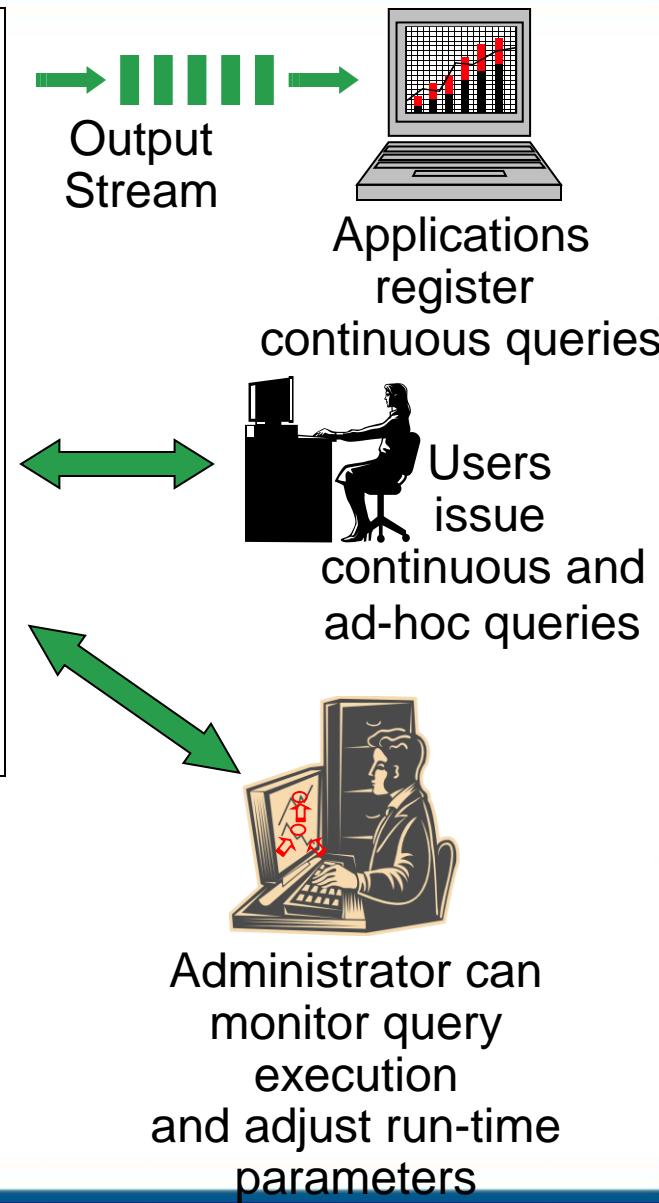
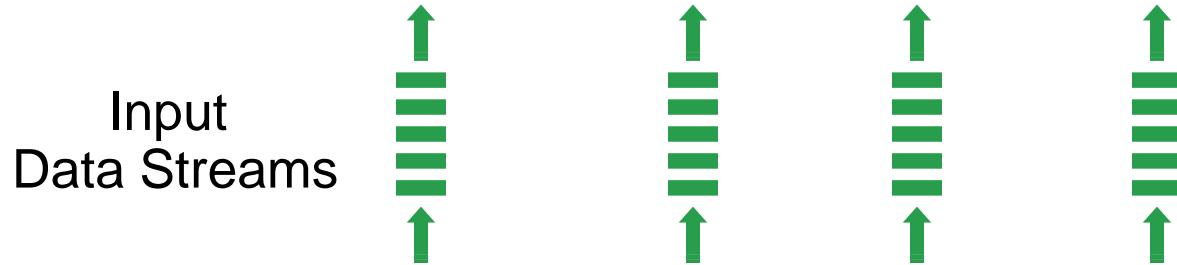
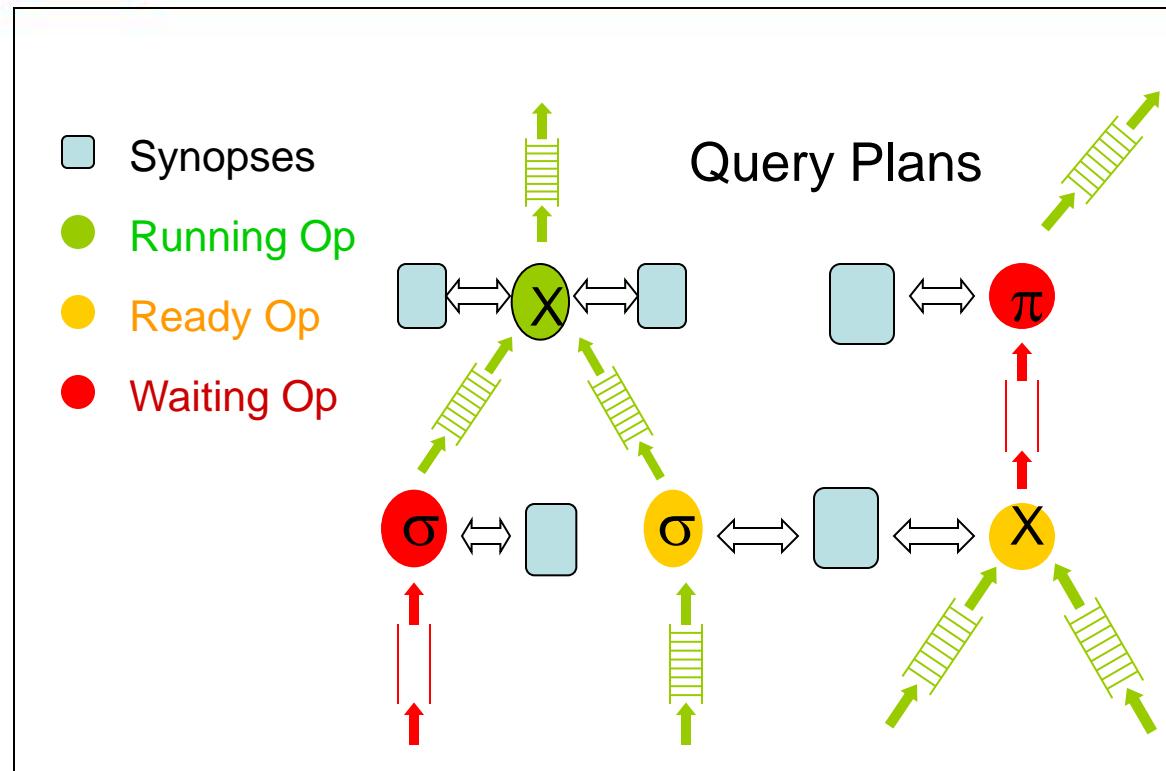
DBMS versus DSMS

- Persistent relations
- One-time queries
- Random access
- Access plan determined by query processor and physical DB design
- “Unbounded” disk store
- Transient streams (and persistent relations)
- Continuous queries
- Sequential access
- Unpredictable data arrival and characteristics
- Bounded main memory

Data Stream Queries -- Basic Issues

- Specifying queries over streams
 - SQL-like versus dataflow network of operators
- Answer availability
 - One-time
 - Multiple-time
 - Continuous ("standing"), stored or streamed
- Registration time
 - Predefined
 - Ad-hoc
- Semantic issues
 - Blocking operators, e.g., *aggregation, order-by*
 - Streams as sets versus lists

Query Processing Architecture



Multiple Queries

- An engine can possibly support a large number of continuous queries
- Multi-query optimization: detecting common operations on common streams and factoring them
 - It is convenient, because the same stream is typically inspected according to many queries
 - It is possible, because queries are registered and rarely change

Operations on Streams

- Filtering streams (move tuples to different streams, remove certain tuples from the stream)
- Join streams (compare tuples from different streams - based upon matching conditions)
- Merging streams (according to time of arrival or logical conditions)
- Map streams (change tuples “on the fly” based upon value-based mappings)
- Compute stream aggregates (moving average, summation over given time intervals)
- Metronome operations: output data at given times
- Heartbeat operations: ouput data synchronized with given feed beats
- Managing disorder or late/missing data

Data window

- A portion of stream, used to put a boundary to (endless) streams.
- Windows can be defined by defining their starting/ending time (logical windows) or their total number of tuples (physical windows)
- Windows can be overlapping (incremented by a given step, smaller than the window size) or tumbling (the step is equal to the window size).

Query Evaluation -- Approximation

- Why approximate?
 - Streams are coming too fast
 - Exact answer requires unbounded storage or significant computational resources
 - Ad hoc queries reference history
- Issues in approximation
 - How is approximation controlled?
 - How is it understood by user?
- Accuracy-efficiency-storage tradeoff

Query Evaluation -- Adaptivity

- Why adaptivity?
 - Queries are long-running
 - Fluctuating stream arrival & data characteristics
 - Evolving query loads
- Issues in adaptivity
 - Adaptive resource allocation (memory, computation)
 - Adaptive query execution plans

StreamBase

- Product designed specifically for managing streams
- Proposes StreamSQL, an SQL extension with streams, windows, and tables as ingredients, and the major stream operations
 - Integrates stream processing with table processing
 - The company pushes the language standardization
- Has a powerful graphic environment for specifying dataflow of tuples and operations upon them
 - Claims very fast development of applications
- Provides partitioning by splitting to guarantee scalability of applications

Coral8

- Product designed specifically for managing streams
- Proposes CCL (Continuous Computation Language), also an SQL extension with streams, windows, and tables as ingredients, and the major stream operations
- Has an engine supporting complex event processing
- Has an integrated engine with IBM DB2, thus supporting native XML processing

Sample Applications

- Network management and traffic engineering
(e.g., Sprint)
 - Streams of measurements and packet traces
 - Queries: detect anomalies, adjust routing
- Telecom call data (e.g., AT&T)
 - Streams of call records
 - Queries: fraud detection, customer call patterns, billing

Sample Applications (cont'd)

- Network security
(e.g., iPolicy, NetForensics/Cisco, Netscreen)
 - Network packet streams, user session information
 - Queries: URL filtering, detecting intrusions & DOS attacks & viruses
- Financial applications
(e.g., Traderbot)
 - Streams of trading data, stock tickers, news feeds
 - Queries: arbitrage opportunities, analytics, patterns

Sample Applications (cont'd)

- Web tracking and personalization (e.g., Yahoo, Google, Akamai)
 - Clickstreams, user query streams, log records
 - Queries: monitoring, analysis, personalization
- Truly massive databases (e.g., Astronomy Archives)
 - Stream the data once (or over and over)
 - Queries do the best they can

Advanced Databases

9

Physical data structures and query optimization

Study of “inside” DB technology: why?

- DBMSs provide “transparent” services:
 - So transparent that, so far, we could ignore many implementation details!
 - So far DBMSs have always been a “black box”
- So... why should we open the box?
 - Knowing [how](#) it works may help to use it better
 - Some services are provided separately

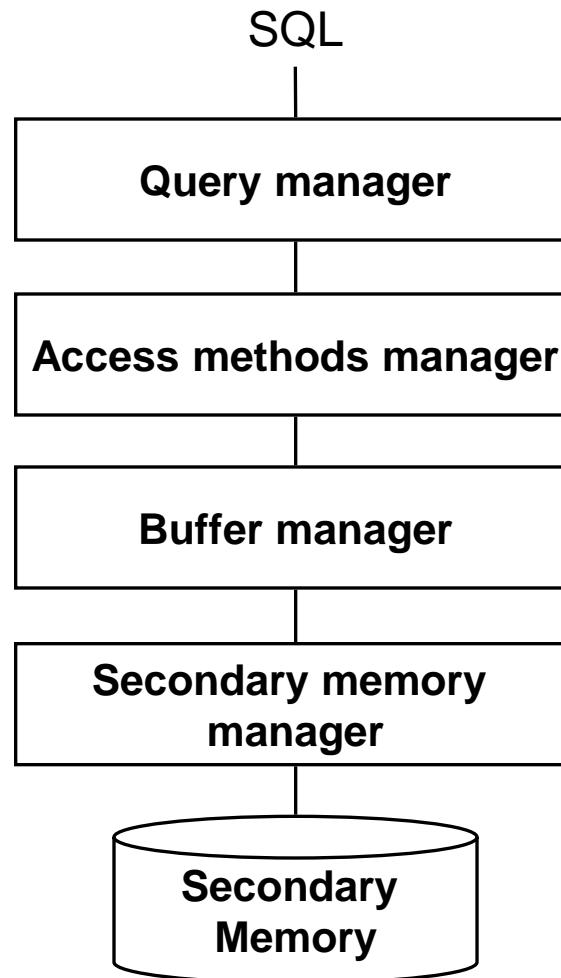
DataBase Management System – DBMS

A system (**software product**) capable of managing **data collections** which are:

- **large** ((much) larger than the central memory available on the computers that run the software)
- **persistent** (with a lifetime which is independent of single executions of the programs that access them)
- **shared** (in use by several applications at a time)

guaranteeing **reliability** (i.e. tolerance to hardware and software failures) and **privacy** (by disciplining and controlling all accesses).

Access and query manager



Technology of DBMSs - topics

- Query management ("optimization")
- Physical data structures and access structures
- Buffer and secondary memory management
- Reliability control
- Concurrency control
- Distributed architectures

Main and Secondary memory (1)

- Programs can only refer to data stored in main memory
- Databases must be stored (mainly) in secondary memory for two reasons:
 - size
 - persistence
- Data stored in secondary memory can only be used if first transferred to main memory
 - (which explains the "main" and "secondary" terminology)

Main and Secondary memory (2)

- Secondary memory devices are organized in **blocks** of (usually) **fixed** length (order of magnitude: a few KBs)
- The only available operations for such devices are reading and writing one **page**, i.e. the byte stream corresponding to a block;
- For convenience and simplicity, we will use **block** and **page** as synonyms

Main and Secondary memory (3)

- Secondary memory access:
 - **seek** time (10-50ms) - *head positioning*
 - **latency** time (5-10ms) - *disc rotation*
 - **transfer** time (1-2ms) - *data transfer*
as an average, hardly less than 10 ms
- The cost of an access to secondary memory is 4 orders of magnitude higher than that to main memory
- In "**I/O bound**" applications the cost **exclusively** depends on the number of accesses to secondary memory

DBMS and file system (1)

- The File System (FS) is the component of the Operating Systems which manages access to secondary memory
- DBMSs make limited use of FS functionalities: to create and delete files and for reading and writing single blocks or sequences of consecutive blocks.
- The DBMS directly manages the file organization, both in terms of the distribution of records within blocks and with respect to the internal structure of each block.

DBMS and file system (2)

- The DBMS manages the blocks of allocated files as if they were a single large space in secondary memory.
- It builds in such space the physical structures with which tables are implemented.
- A file is typically dedicated to a single table, but....
- It may happen that a file contains data belonging to more than one table and that the tuples of one table are split in more than one file.

Blocks and records

- Blocks (the "physical" components of a file) and records (the "logical" components) generally have different size:
 - The size of a block depends on the file system
 - The size of a record depends on the needs of applications and is normally **variable** within a file

Block Factor

- The number of records within a block
 - S_R : Size of a record (assumed constant in the file for simplicity: "fixed length record")
 - S_B : Size of a block
 - if $S_B > S_R$, there may be many records in each block:
$$\lfloor S_B / S_R \rfloor$$
- The rest of the space can be
 - used ("spanned" records (or "hung-up" records))
 - non used ("unspanned" records)

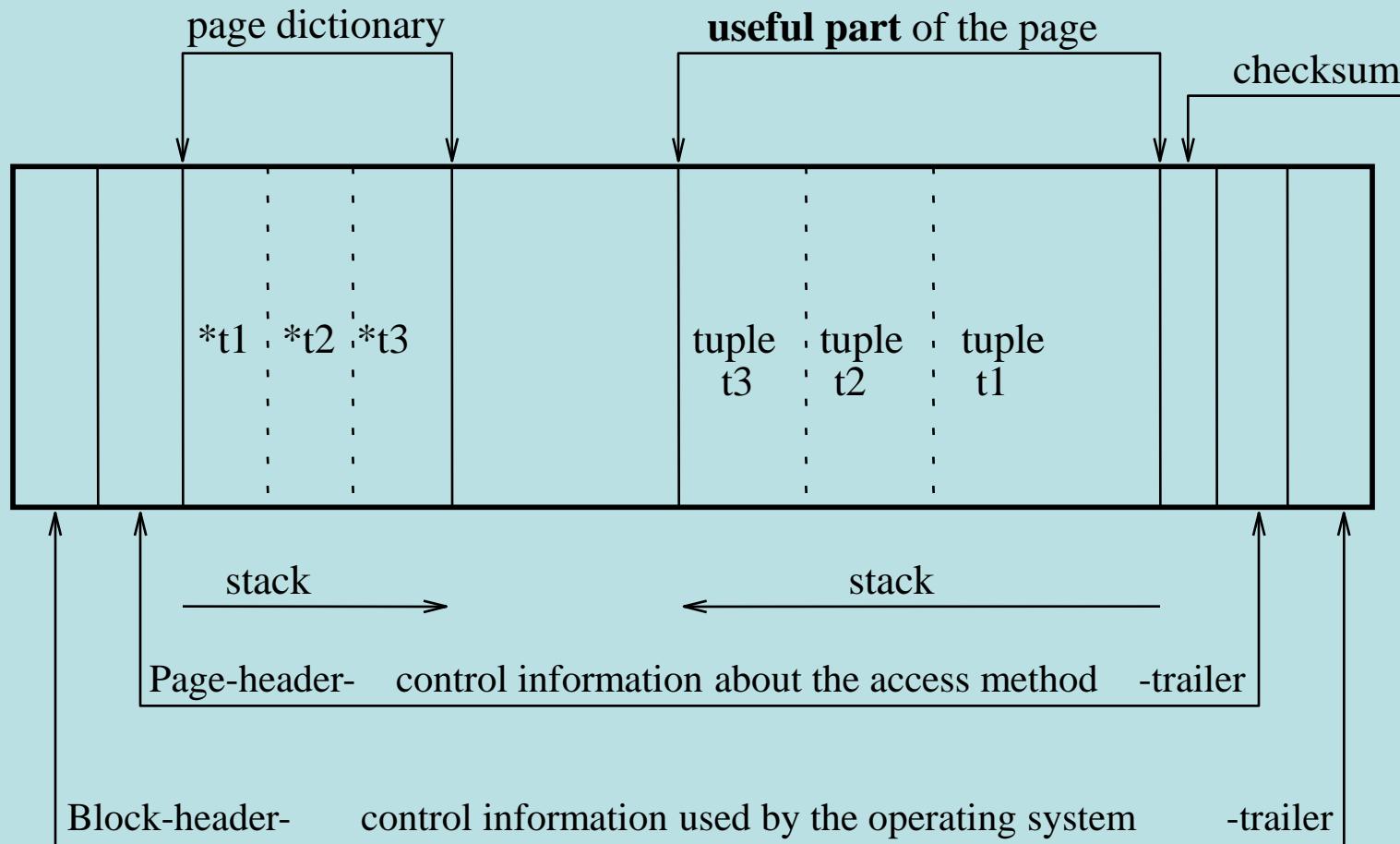
Physical access structures

- Used for the efficient storage and manipulation of data within the DBMS
- Encoded as *access methods*, that is, software modules providing data access and manipulation primitives for each physical access structure
- Each DBMS has a distinctive and limited set of access methods
- We will consider three types of data structures:
 - Sequential
 - Hash-based
 - Tree-based (or index-based)

Organization of tuples within pages

- Each access method has its own page organization
- In the case of sequential and hash-based methods each page has:
 - An initial part (**block header**) and a final part (**block trailer**) containing control information used by the **file system**
 - An initial part (**page header**) and a final part (**page trailer**) containing control information about the **access method**
 - A **page dictionary**, which contains pointers to each item of useful elementary data contained in the page
 - A **useful part**, which contains the data. In general, the page dictionary and the useful data grow as opposing stacks
 - A **checksum**, to verify that the information in it is valid
- Tree structures have a different page organization

Organization of tuples within pages



Page manager primitives

- ***Insertion and update of a tuple***
 - may require a reorganization of the page if there is sufficient space to manage the extra bytes introduced
- ***Deletion of a tuple***
 - often carried out by marking the tuple as 'invalid'
- ***Access to a field of a particular tuple***
 - identified according to the offset and to the length of the field itself, after identifying the tuple by means of its key or its offset

Sequential structures

- Characterized by a sequential arrangement of tuples in the secondary memory
- Three cases: **entry-sequenced, array, sequentially-ordered**
 - In an *entry-sequenced* organization, the sequence of the tuples is dictated by their order of entry
 - In an *array* organization, the tuples (all of the same size) are arranged as in an array, and their positions depend on the values of an index (or indexes)
 - In a *sequentially-ordered* organization, the sequence depends on the value assumed in each tuple by a field that controls the ordering, known as the *key field*

“Entry-sequenced” sequential structure

- Optimal for carrying out **sequential** reading and writing operations
- Optimal for **space occupancy**, as it uses all the blocks available for files and all the spaces within the blocks
- **Non** optimal with respect to
 - searching specific data units or
 - updates that require more space

“Array” sequential structure

- Possible only when the tuples are of fixed length
- Made of n of adjacent blocks, each block with m slots available for tuples
- Each tuple has a numerical index i and is placed in the i -th position of the array

“Sequentially-ordered” sequential structure

- Each tuple has a position based on the value of the key field
- Historically, such structures were used on sequential devices ([tapes](#)). This had fallen out of use, but for data streams and system logs
- The main problems are insertions or updates which increase the physical space - they require reordering techniques for the tuples already present:
- Options to avoid global reorderings:
 - Differential files (example: yellow pages)
 - Leaving a certain number of slots free at the time of first loading, followed by ‘local reordering’ operations
 - Integrating the sequentially ordered files with an *overflow file*, where new tuples are inserted into blocks linked to form an *overflow chain*

Hash-based access structures

- Ensure an efficient *associative* access to data, based on the value of a *key* field
- A hash-based structure has B blocks (often adjacent)
- A hash algorithm is applied to the key field and returns a value between zero and $B-1$. This value is interpreted as the position of the block in the file, and used both for reading and writing the block
- This is the most efficient technique for queries with equality predicates, but it is rather inefficient for queries with interval predicates

Features of hash-based structures

- Primitive interface: `hash(fileId,Key) :BlockId`
- The implementation consists of two parts.
 - *folding*, transforms the key values so that they become positive integer values, uniformly distributed over a large range.
 - *hashing* transforms the positive binary number into a number between zero and $B - 1$.
- Optimal performance if the file is larger than necessary.
Let:
 - T be the number of tuples expected for the file,
 - F be the average number of tuples stored in each page;then a good choice for B is $T/(0.8 \times F)$, using only 80% of the available space

Collisions

- Collisions occur when the same block number is associated to too many tuples. They are critical when the maximum number of tuples per block is exceeded
- Collisions are solved by adding an overflow chain
 - This gives the additional cost of scanning the chain
- The average length of the overflow chain is a function of the ratio $T/(F \times B)$ and of the average number F of tuples per page:

	1	2	3	5	10	F
.5	0.5	0.177	0.087	0.031	0.005	
.6	0.75	0.293	0.158	0.066	0.015	
.7	1.167	0.494	0.286	0.136	0.042	
.8	2.0	0.903	0.554	0.289	0.110	
.9	4.495	2.146	1.377	0.777	0.345	

$T/(Fx B)$

An example

- 40 records
- hash table with 50 positions:
 - 1 collision of 4 values
 - 2 collisions of 3 values
 - 5 collisions of 2 values

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

About hashing

- Performs best for direct access based on equality for values of the key
- Collisions (overflow) can be managed by using the next available block or with linked blocks into an area called **overflow file**
- **Inefficient** for access based on interval predicates or based on the value of non-key attributes
- Hash files "degenerate" if the extra-space is too small (should be at least 120% of the minimum required space) and if the file size changes a lot over time

Tree structures

- The most frequently used in relational DBMSs
 - SQL indexes are implemented in this way
- Gives associative access based on the value of a *key*
 - no constraints on the physical location of the tuples
- Note: the **primary key** of the relational model and the **keys** for hash-based and tree structures are different concepts

Index file

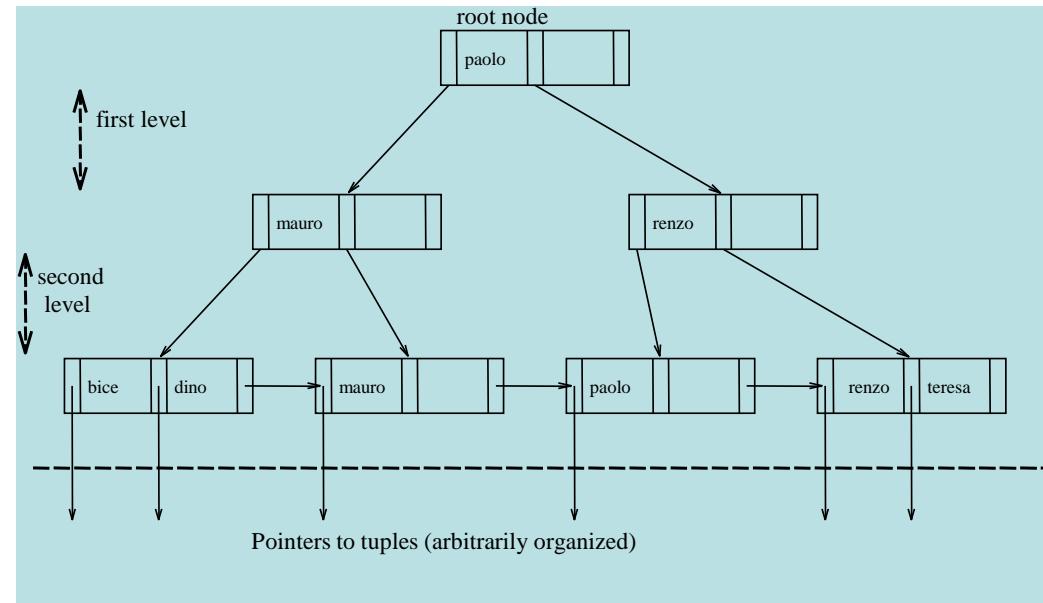
- Index: an auxiliary structure for the efficient access to the records of a file based upon the values of a given field or record of fields – called the index key.
- The index concept: index of a book, seen as a list of (term; page list) pairs, alphabetically ordered at the end of a book.
- The index key is not a key!

Types of indexes

- Primary index:
 - Based upon the primary key
- Secondary index
 - Based upon other attributes (including secondary keys)
- Clustered index
 - One such that the records of the physical file are physically ordered according to the index key
- Dense index:
 - One having an index entry for every record of the file
- Sparse index:
 - Having less index entries than the number of records of the file

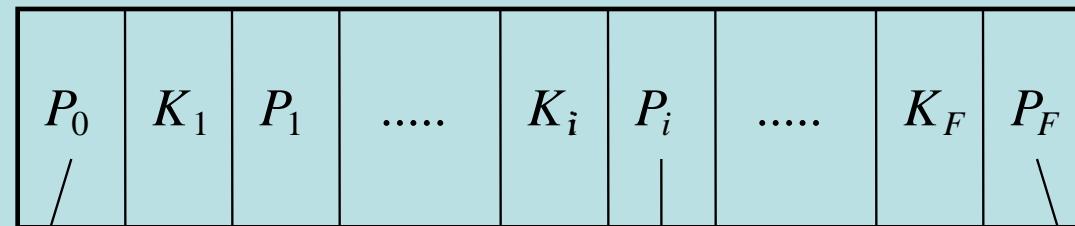
Tree structures

- Each tree has:
 - one root node
 - several intermediate nodes
 - several leaf nodes



- Each node corresponds to a block
- The links between the nodes are established by **pointers** to mass memory
- In general, each node has a large number of descendants (**fan out**), and therefore the majority of pages are leaf nodes
- In a **balanced tree**, the lengths of the paths from the root node to the leaf nodes are all equal. Balanced trees give **optimal** performance.

Structure of the tree nodes



sub-tree with keys
 $K < K_1$

sub-tree with keys
 $K_i \leq K < K_{i+1}$

sub-tree with keys
 $K \geq K_F$

B and B+ trees

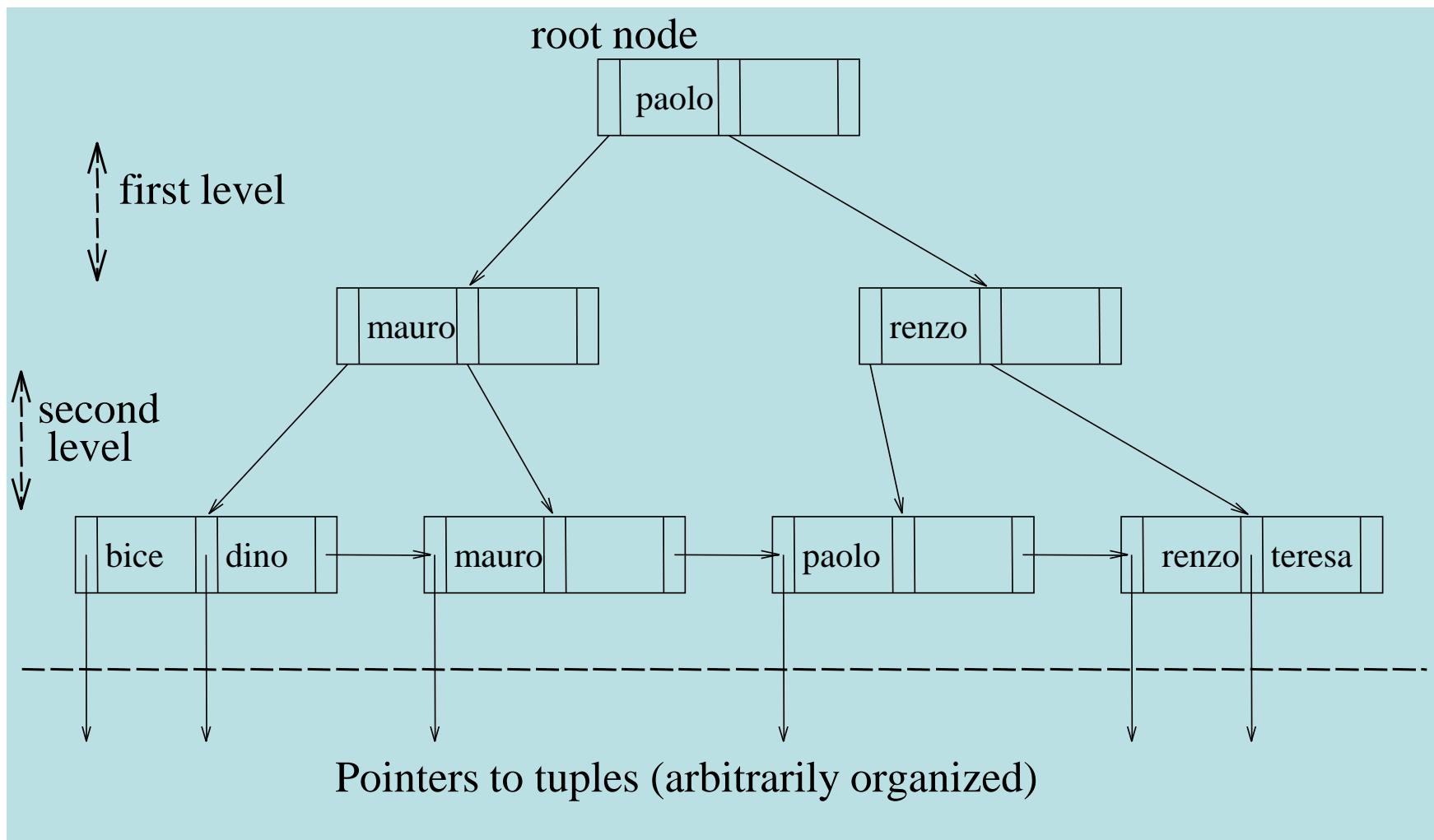
- **B+ trees**

- The leaf nodes are linked in a chain in the order imposed by the key.
- Supports interval queries efficiently
- The most used by relational DBMSs

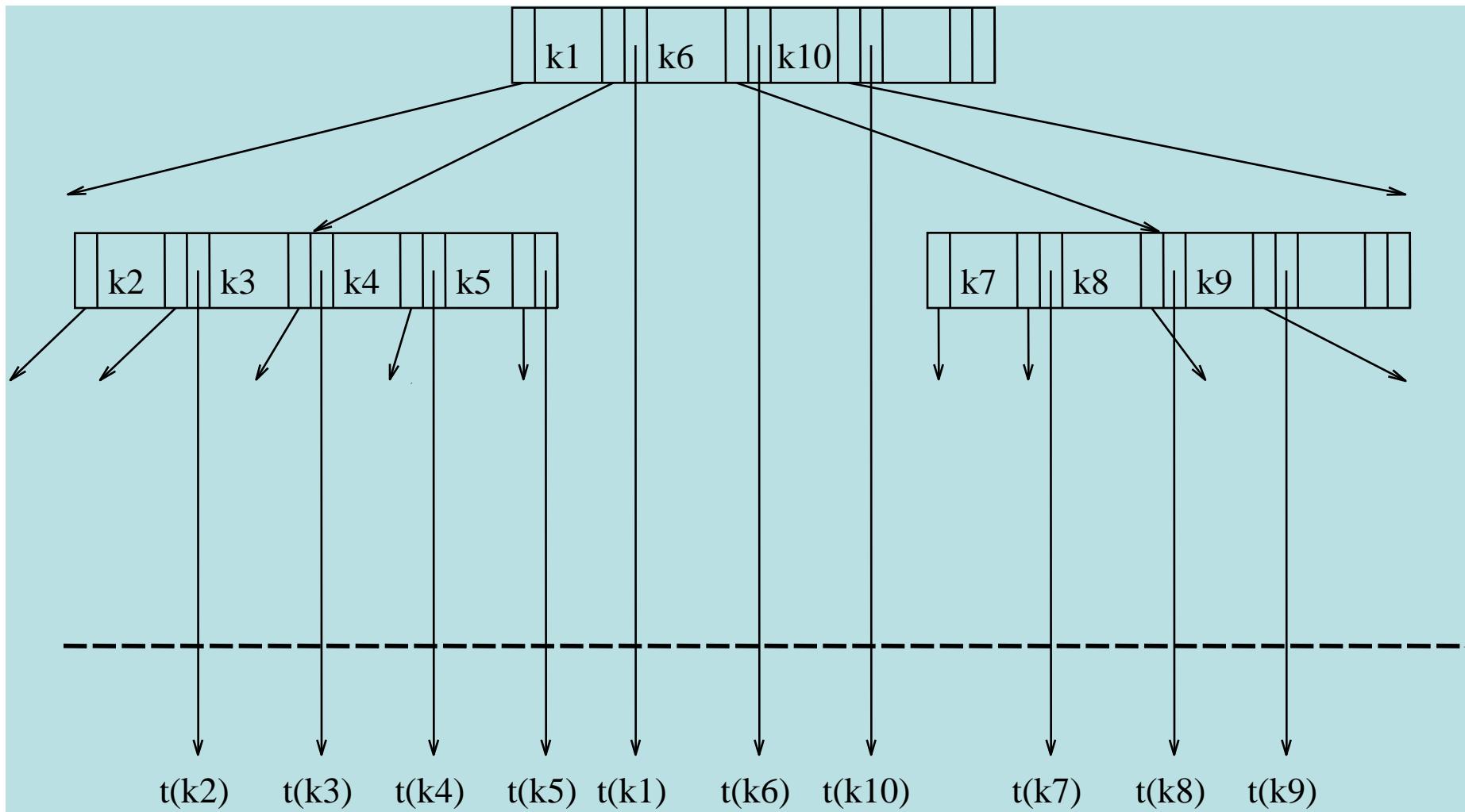
- **B trees**

- No sequential connection for leaf nodes
- Intermediate nodes use two pointers for each key value K_i
 - one points directly to the block that contains the tuple corresponding to K_i
 - the other points to a sub-tree with keys greater than K_i and less than K_{i+1}

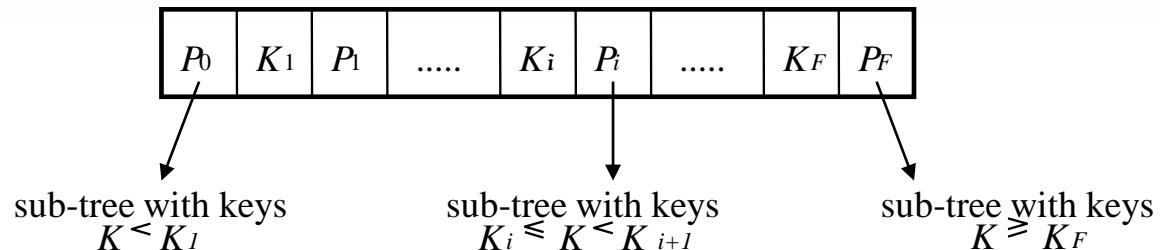
An example of B+ tree



An example of B tree



Search technique



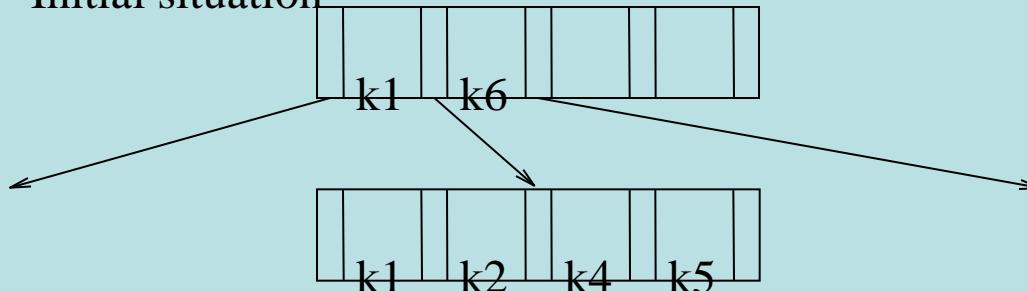
- Looking for a tuple with key value V , at each intermediate node:
 - if $V < K_1$ follow P_0**
 - if $V \geq K_F$ follow P_F**
 - otherwise, follow P_j such that $K_j \leq V < K_{j+1}$**
- The leaf nodes can be organized in two ways:
 - In *key-sequenced trees* tuples are contained in the leaves
 - In *indirect trees* leaf nodes contain pointers to the tuples, allocated with any other 'primary' mechanism (entry-sequenced, hash, key-sequenced, ...)

Split and Merge operations

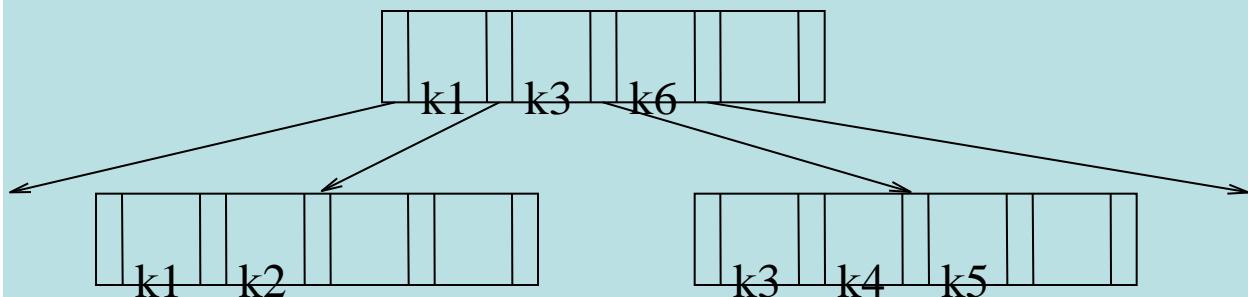
- SPLIT: required when the insertion of a new tuple cannot be done locally to a node
 - Causes an increase of pointers in the superior node and thus could recursively cause another split
- MERGE: required when two “close” nodes have entries that could be condensed into a single node. Done in order to keep a high node filling and minimal paths from the root to the leaves.
 - Causes a decrease of pointers in the superior node and thus could recursively cause another merge

Split and merge

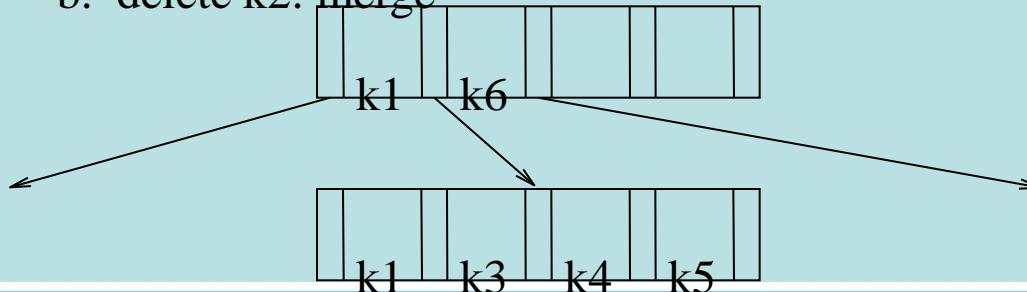
Initial situation



a. insert k3: split →



b. delete k2: merge ←



Index usage

- Syntax in SQL:
 - `create [unique] index IndexName on TableName(AttributeList)`
 - `drop index IndexName`
- Every table should have:
 - A **primary index**, with key-sequenced structure, normally unique, on the primary key
 - Several **secondary indexes**, both unique and not unique, on the attributes most used for selections and joins
- They are progressively added, checking that the system actually uses them, and without excess

Query optimization

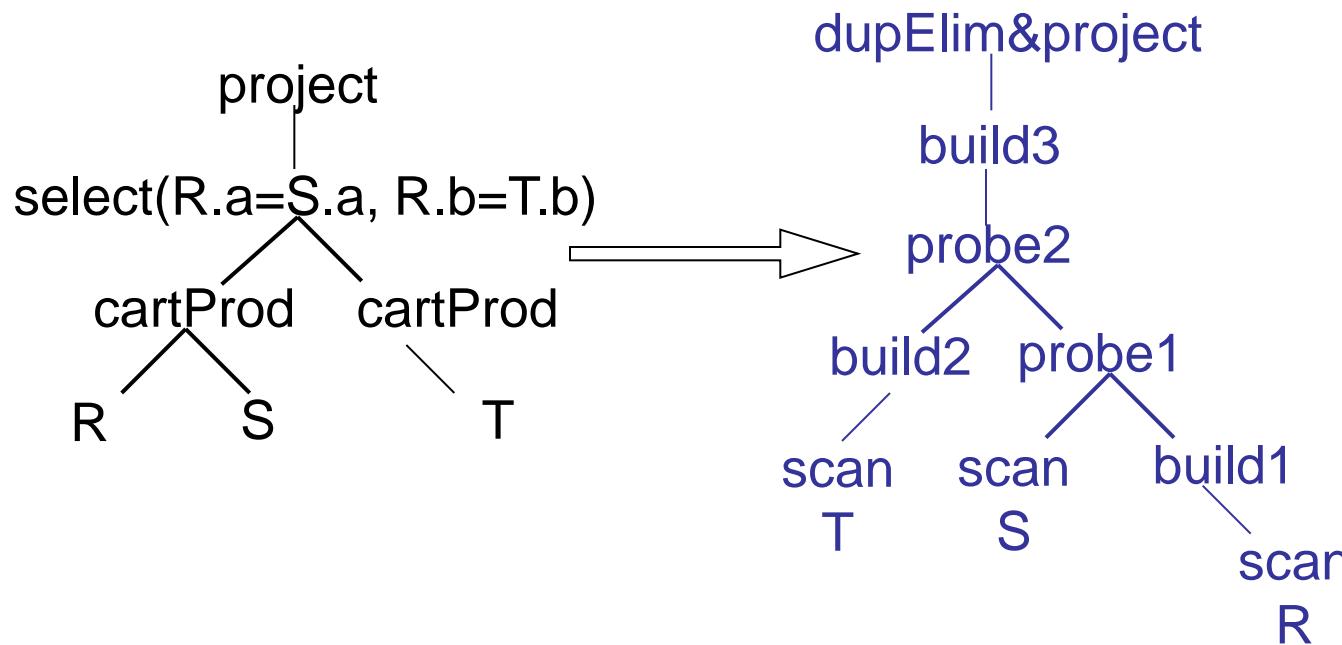
- Optimizer: an important module in the architecture of a DBMS
- It receives a query written in SQL and produces an access program in 'object' or 'internal' format, which uses the data access methods.
- Steps:
 - Lexical, syntactic and semantic analysis
 - Translation into an internal representation
 - Algebraic optimization
 - Cost-based optimization
 - Code generation

Internal representation of queries

- A tree representation, similar to that of relational algebra:
 - Leaf nodes correspond to the physical data structures (tables, indexes, files).
 - intermediate nodes represent physical data access operations that are supported by the access methods
- Typical operations include sequential scans, orderings, indexed accesses and various methods for evaluating joins and aggregate queries, as well as materialization choices for intermediate results

Query optimization input-output

- Input: query in SQL
SELECT A FROM R,S,T WHERE R.A=S.A AND R.B=T.B
- Output: execution plan



Approaches to query execution

- ***Compile and store***: the query is compiled once and executed many times
 - The internal code is stored in the DBMS, together with an indication of the dependencies of the code on the particular versions of catalog used at compile time
 - On relevant changes of the catalog, the compilation of the query is invalidated and repeated
- ***Compile and go***: immediate execution, no storage
 - Even if not stored, the code may live for a while in the DBMS and be available for other executions

Relation profiles

- Profiles contain quantitative information about tables and are stored in the data dictionary:
 - the cardinality (number of tuples) of each table T
 - the dimension in bytes of each attribute A_j in T
 - the number of distinct values of each attribute A_j in T
 - the minimum and maximum values of each attribute A_j in T
- Periodically calculated by activating appropriate system primitives (for example, the **update statistics** command)
- Used in cost-based optimization for estimating the size of the intermediate results produced by the query execution plan

Sequential scan

- Performs a sequential access to all the tuples of a table or of an intermediate result, at the same time executing various operations, such as:
 - Projection to a set of attributes
 - Selection on a simple predicate (of type: $A_i = v$)
 - Sort (ordering)
 - Insertions, deletions, and modifications of the tuples currently accessed during the scan
- Primitives:
Open, next, read, modify, insert, delete, close

Sort

- This operation is used for ordering the data according to the value of one or more attributes. We distinguish:
 - Sort in main memory, typically performed by means of ad-hoc algorithms
 - Sort of large files, which can not be transferred to main memory, performed by merging smaller parts with already sorted parts

Indexed access

- Indexes are used when queries include:
 - simple predicates (of the type $A_i = v$)
 - interval predicates (of the type $v_1 \leq A_i \leq v_2$)
- We say that such predicates are *supported* by the index
 - With conjunctions of supported predicates, the DBMS chooses the most selective supported predicate for the primary access, and evaluates the other predicates in main memory
- With disjunctions predicates:
 - if any of them is not supported a scan is needed;
 - if all are supported, indexes can be used only with duplicate elimination

Join Methods

- Joins are the most frequent (and costly) operations in DBMSs
- There are several methods for join evaluation, among which:
 - *nested-loop, merge-scan and hashed.*
- These three methods are based on scanning, hashing, and ordering.

Nested-loop join

External table

	A
-----	a

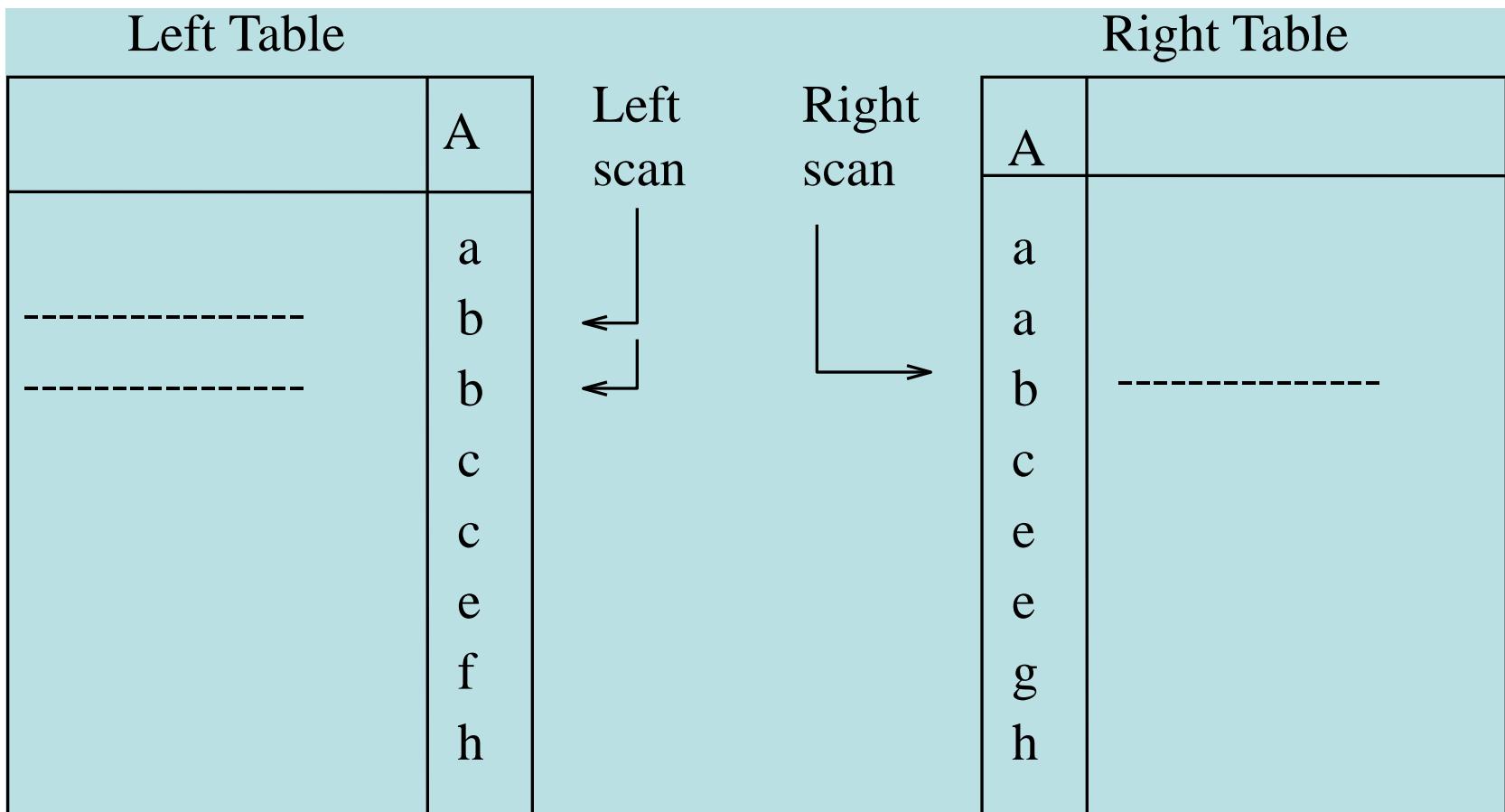
External
scan

Internal scan
or indexed
access

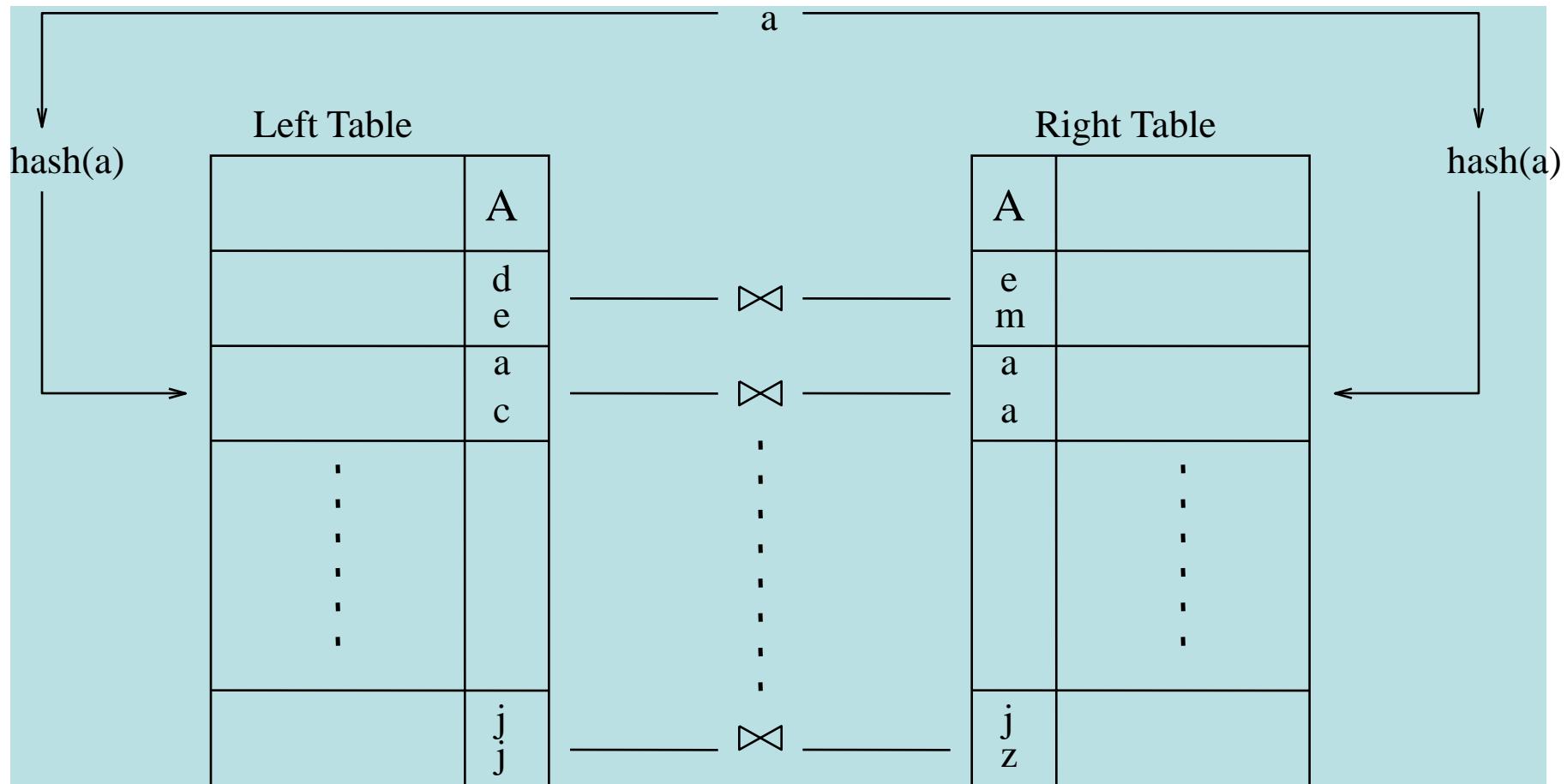
Internal table

A	
a	-----
a	-----
a	-----

Merge-scan join



Hashed join



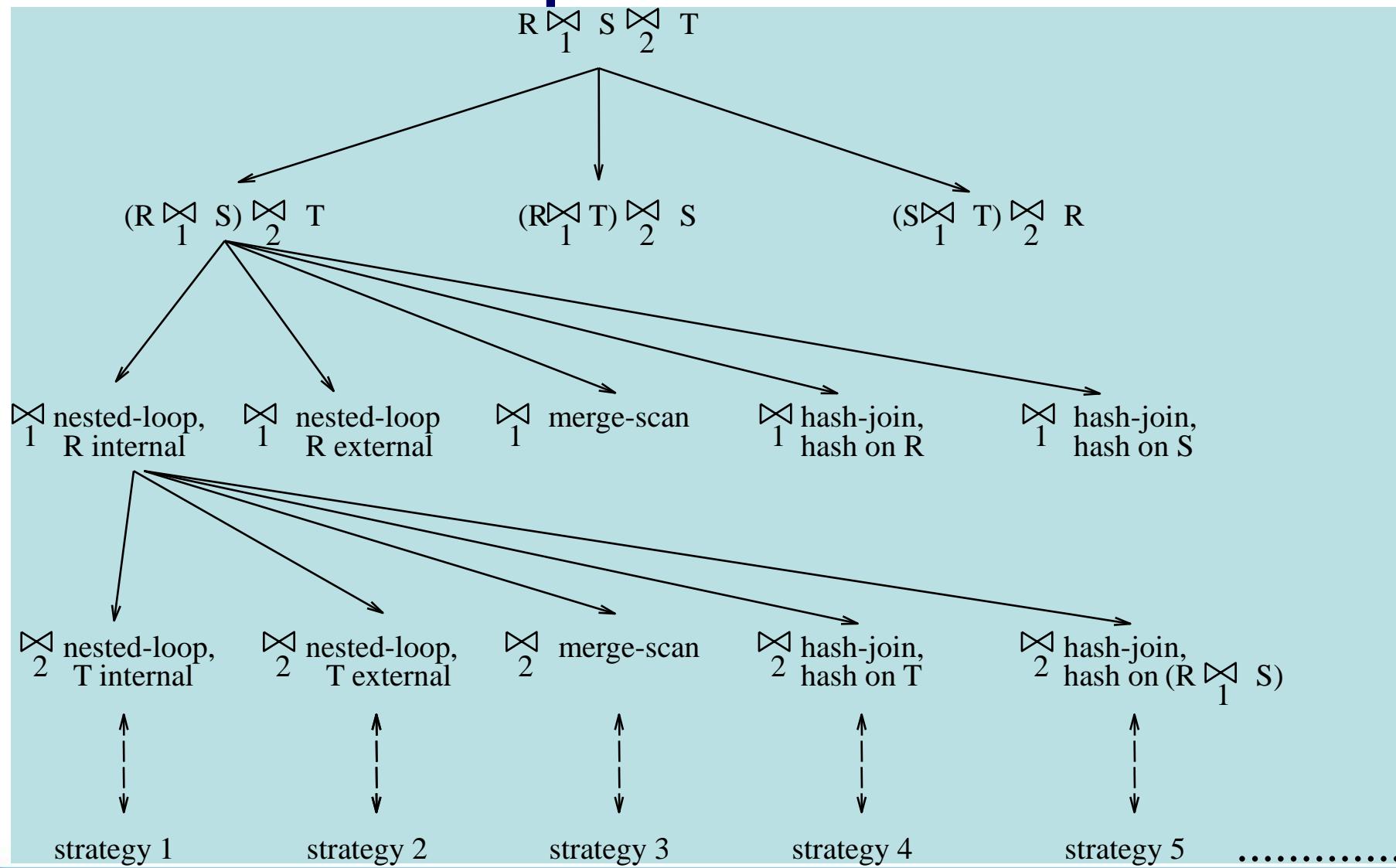
Cost-based optimization

- An optimization problem, whose decisions are:
 - The data access operations to execute (e.g., scan vs index access)
 - The order of operations (e.g., the join order)
 - The option to allocate to each operation (e.g., choosing the join method)
 - Parallelism and pipelining can improve performances
- Further options appear in selecting a plan within a distributed context

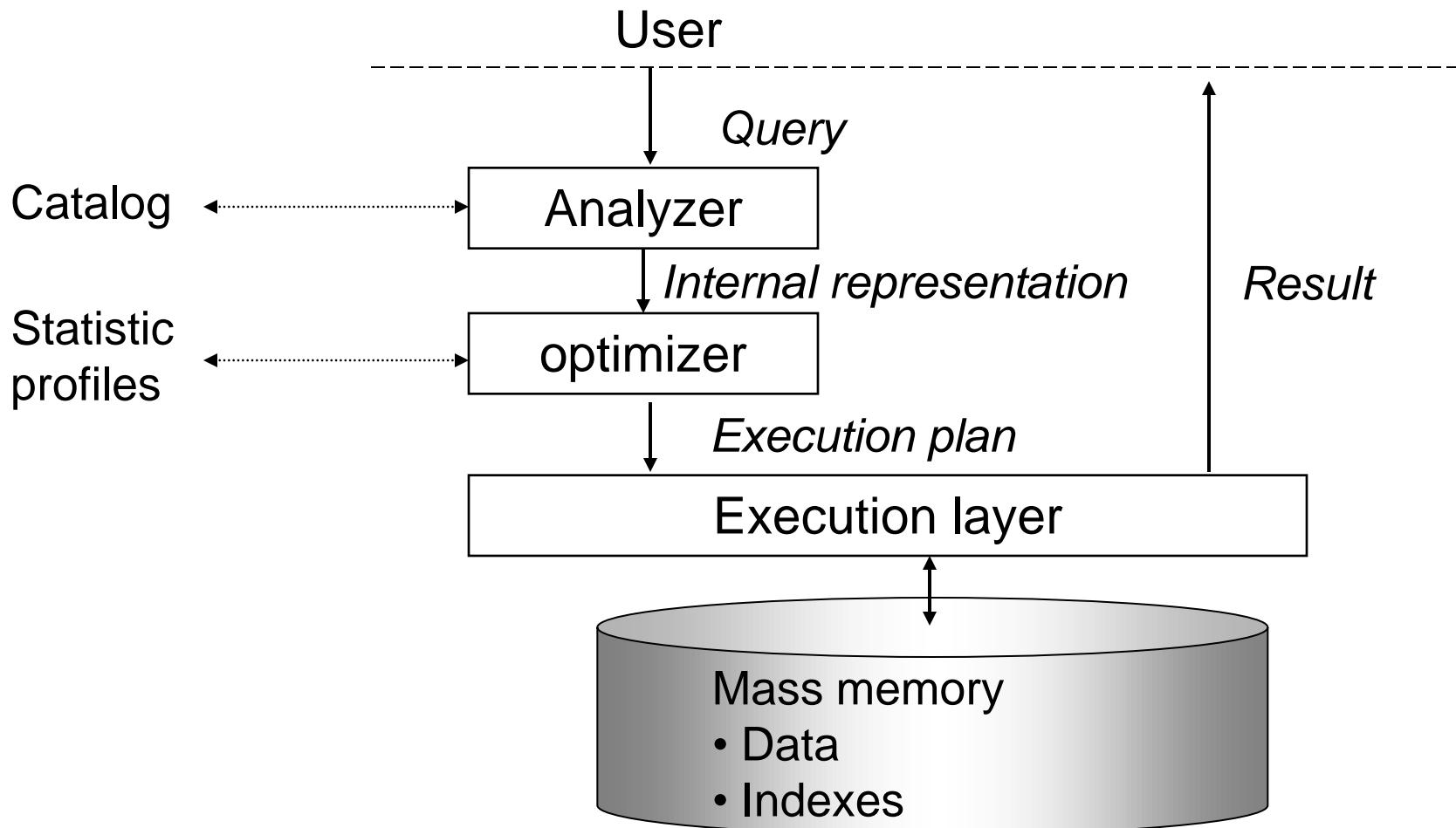
Approach to query optimization

- Optimization approach:
 - Make use of profiles and of approximate cost formulas.
 - Construct a *decision tree*, in which each node corresponds to a choice; each leaf node corresponds to a specific *execution plan*.
 - Assign to each plan a cost:
$$C_{total} = C_{I/O} \ n_{I/O} + C_{cpu} \ n_{cpu}$$
 - Choose the plan with the lowest cost, based on operations research (branch and bound)
- Optimizers should obtain 'good' solutions in a very short time

An example of decision tree



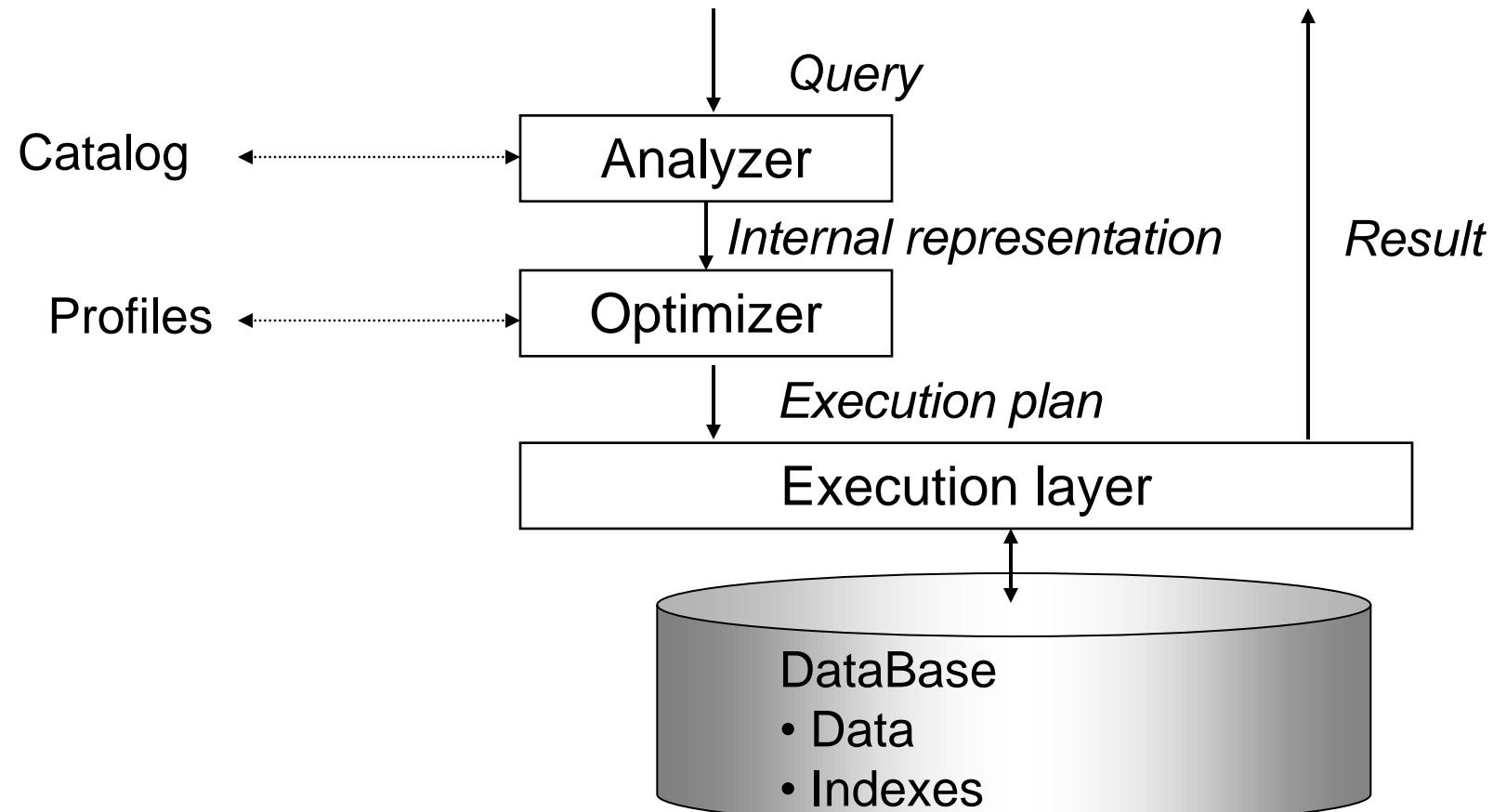
Query processing components



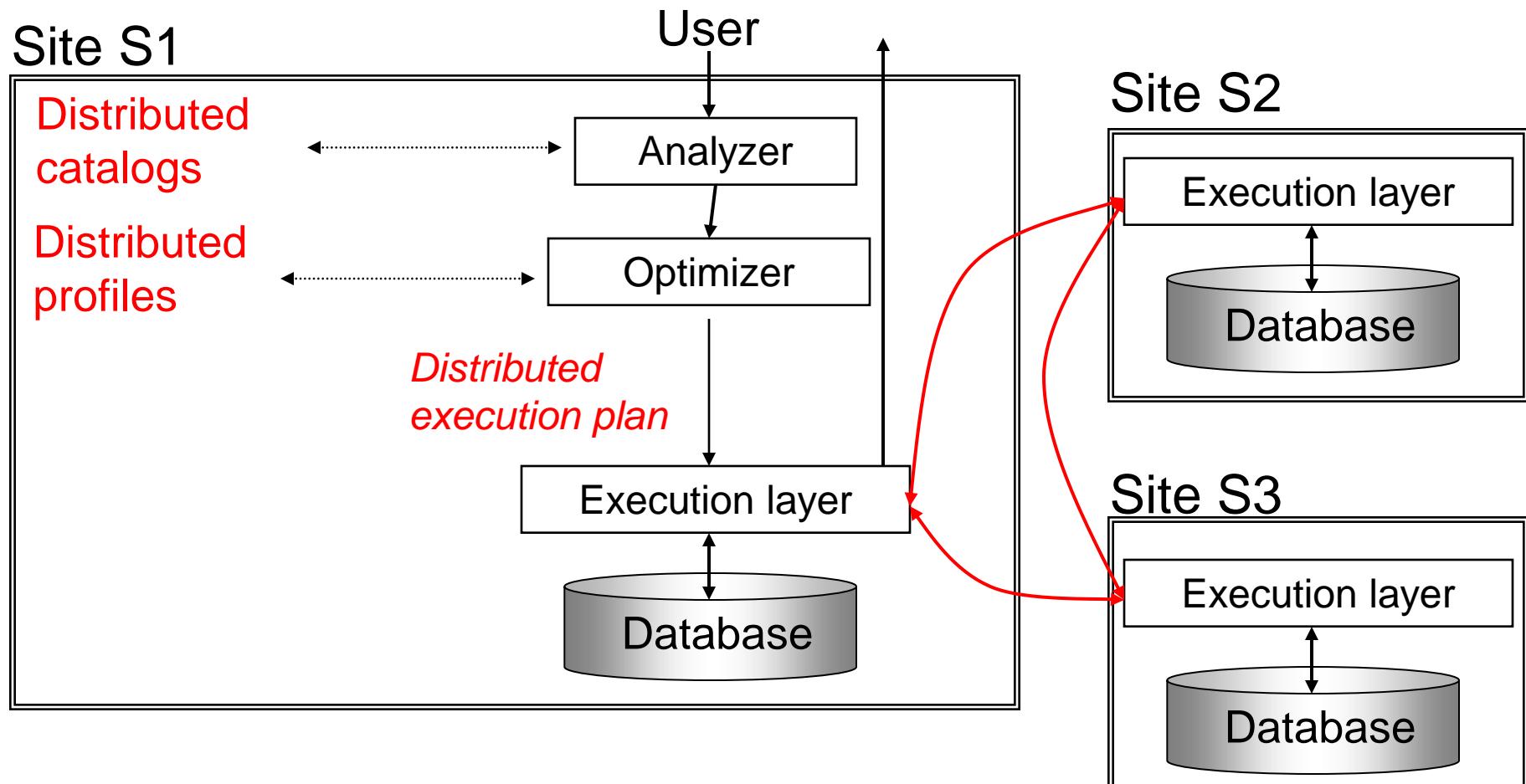
Centralized architecture (DBMS)

DBMS

User



Distributed database with master-slave optimization

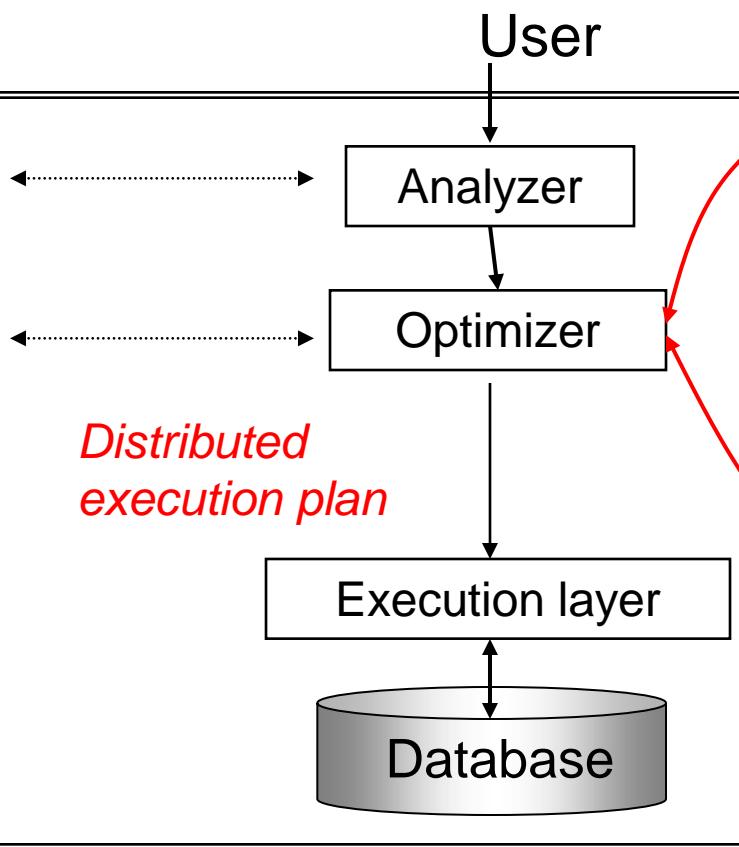


Distributed optimization with negotiation

Site S1

Distributed catalog

Distributed statistics



Site S2

Optimizer

Execution layer

Database

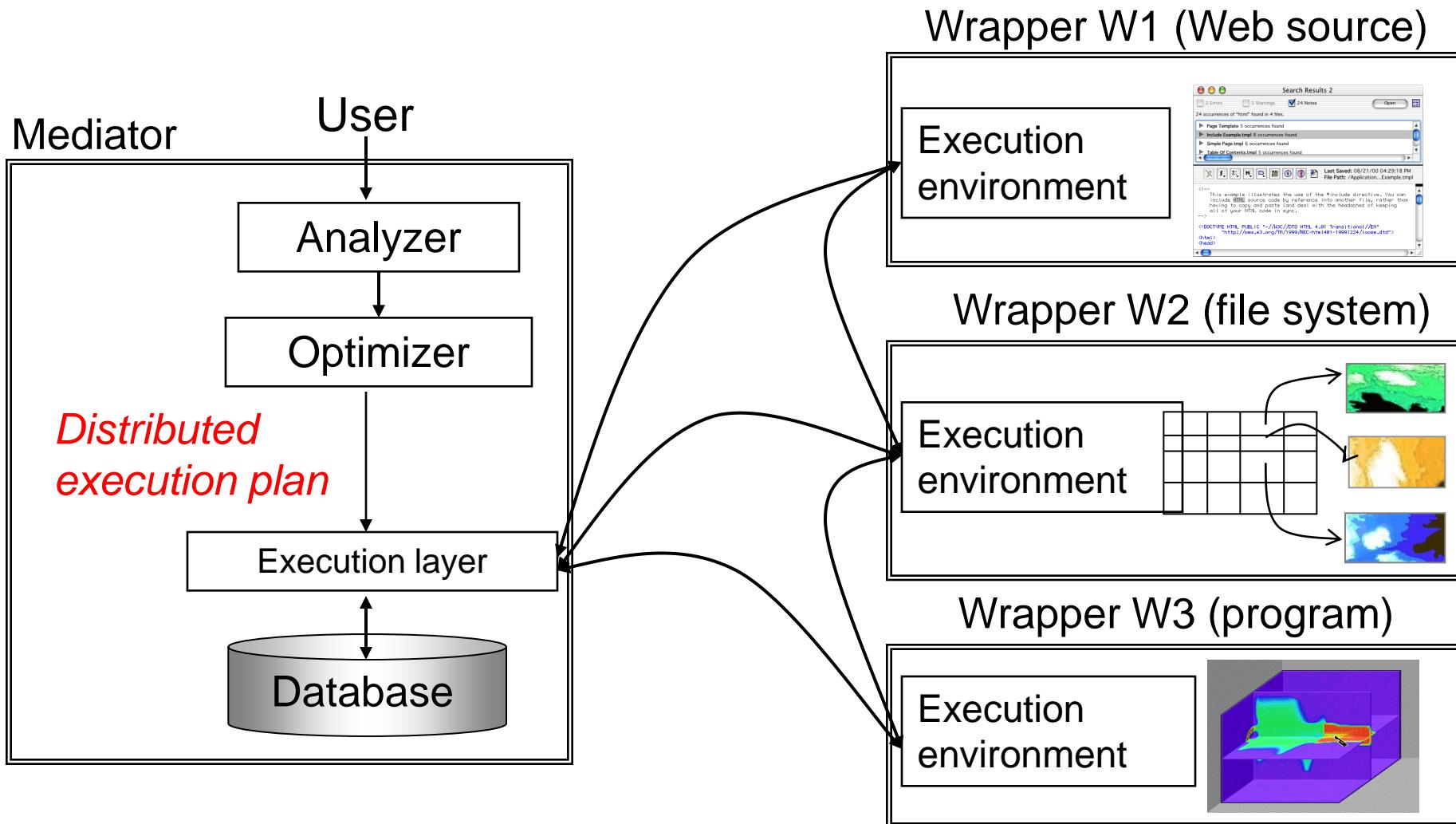
Site S3

Optimizer

Execution layer

Database

Distributed system with mediator and wrappers



Overall view: components of a DBMS

