Data Bases II
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The course aims to prepare software designers on the effective development of database applications.

First, the course presents the fundamental features of current database architectures, with a specific emphasis on the concept of transaction and its realization in centralized and distributed systems.

Then, the course illustrates the main directions in the evolution of database systems, presenting approaches that go beyond the relational model, like active databases, object systems and XML data management solutions.

Contents

CHAPTER 1

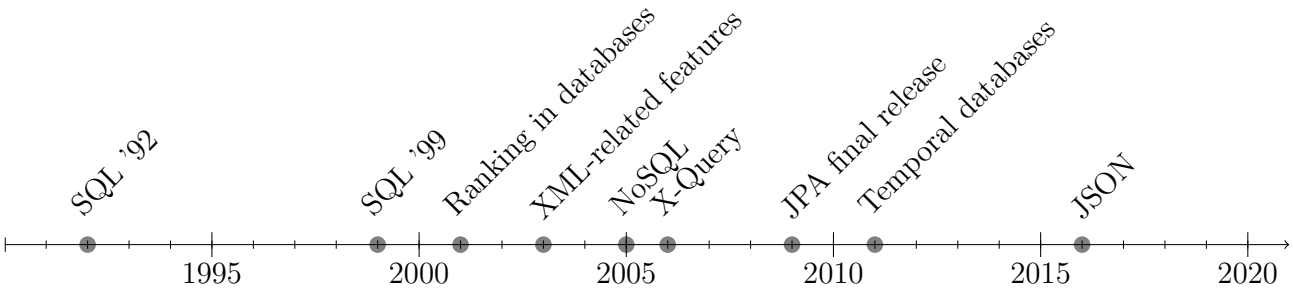
Introduction

1.1 Data Base Management System

Definition

A *Data Base Management System* is a software product capable of managing data collections that 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: used by several applications at a time.
- Reliable: ensuring tolerance to hardware and software failures.
- Data ownership respectful: by disciplining and controlling accesses.



1.2 Transactions

Definition

A *transaction* is an elementary, atomic unit of work performed by an application. Each transaction is conceptually encapsulated within two commands:

- Begin transaction.
- End transaction.

In the context of a transaction, the conclusion of the process is indicated by executing one of the following commands: either commit or rollback.

Definition

The *On-Line Transaction Processing* (OLTP) is a system that supports the execution of transactions on behalf of concurrent applications.

The application has the capability to execute multiple transactions, with transactions being an integral part of the application rather than the other way around. These transactions adhere to the ACID properties:

1. Atomicity: a transaction represents an indivisible unit of execution, ensuring that either all operations within the transaction are executed or none at all. The commitment of the transaction, marked by the execution of the commit command, signifies the successful conclusion. Any error occurring before the commit should trigger a rollback, and errors occurring afterward should not impact the transaction. The rollback can be initiated by either a rollback statement or the Database Management System (DBMS). In the event of a rollback, the executed work must be undone, restoring the database to its state before the transaction's initiation. It becomes the responsibility of the application to determine whether an aborted transaction needs to be redone.

2. Consistency: transactions must adhere to the integrity constraints of the database. If the initial state S_0 is consistent, the final state S_f must also be consistent. This consistency requirement, however, does not necessarily extend to intermediate states S_i.

3. Isolation: the execution of a transaction must remain independent of concurrently executing transactions. This ensures that the outcome of one transaction does not affect the execution of others.

4. Durability: the effects of a successfully committed transaction are permanent, enduring indefinitely and remaining unaffected by any system faults.

Property	Actions	Architectural element
Atomicity	Abort-rollback-restart	Query Manager
Consistency	Integrity checking	Integrity Control System
Isolation	Concurrency control	Concurrency Control System
Durability	Recovery management	Reliability Manager

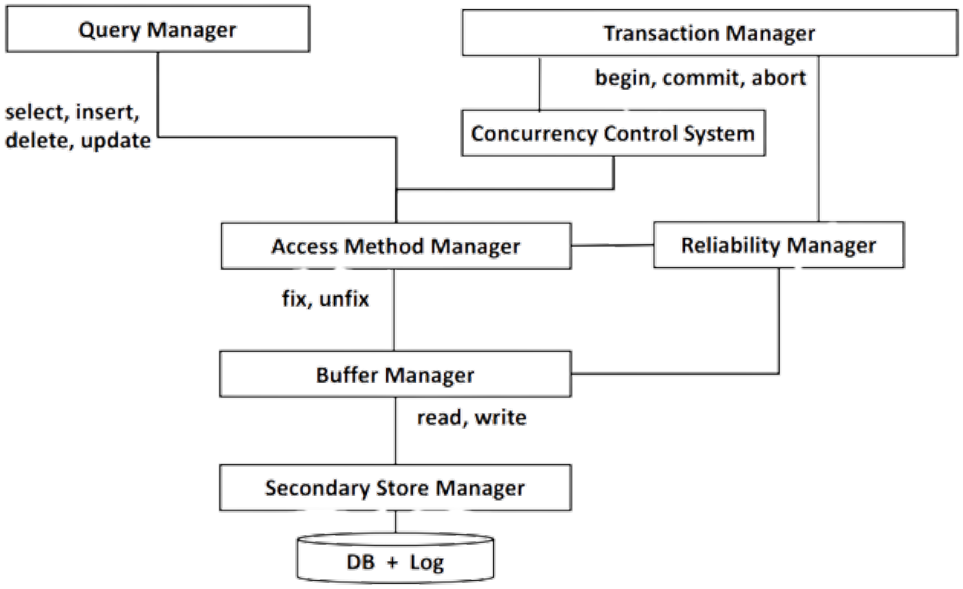


Figure 1.1: Architecture of a Data Base Management System

CHAPTER 2

Concurrency

2.1 Introduction

A Database Management System (DBMS) typically oversees the management of multiple applications. An essential metric for evaluating the workload of a DBMS is the number of transactions per second ("tps") that it can effectively handle. Efficient database utilization requires the DBMS to manage concurrency effectively while preventing the occurrence of anomalies. This is achieved through a concurrency control system, which is responsible for determining the order in which various transactions are scheduled.

2.2 Anomalies in concurrent transactions

The anomalies resulting from incorrect scheduling include:

- Lost update: an update is applied from a state that ignores a preceding update, resulting in the loss of the earlier update.

Transaction t_1	**Transaction t_2**
$r_1(x)$	
$x = x + 1$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$w_1(x)$	
commit	

- Dirty read: an uncommitted value is used to update the data.

Transaction t_1	**Transaction** t_2
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	$r_2(x)$
	commit
abort	

- Non-repeatable read: another transaction updates a previously read value.

Transaction t_1	**Transaction** t_2
$r_1(x)$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

- Phantom update: another transaction updates data that contributes to a previously valid constraint.

Transaction t_1	**Transaction** t_2
$r_1(x)$	
	$r_2(y)$
$r_1(y)$	
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = x + y + z$	
commit	

- Phantom insert: another transaction inserts data that contributes to a previously read datum.

2.3 Concurrency theory

Definition

A *model* is an abstraction of a system, object or process, which purposely disregards details to simplify the investigation of relevant properties.

Concurrency theory is founded on a model of transaction and concurrency control principles that aids in comprehending real systems. Actual systems leverage implementation-level mechanisms to attain desirable properties postulated by the theory.

Definition

An *operation* consist in a reading or in a writing of a specific datum by a specific transaction.

A *schedule* is a sequence of operations performed by concurrent transactions that respects the order of operations of each transaction.

Transactions can be categorized as serial, interleaved, or nested. The number of serial schedules for n transactions is given by:

$$N_S = n!$$

Meanwhile, the total number of distinct schedules, given the number of transactions n, is expressed as:

$$N_D = \frac{(\sum_{i=1}^{n} k_i)!}{\prod_{i=1}^{n} (k_i!)}$$

Example : For two transactions, T_1 and T_2, there are six possible schedules, with only two being serial:

1. $r_1(x)w_1(x)r_2(z)w2(z)$
2. $r_2(z)w_2(z)r_1(x)w_1(x)$
3. $r_1(x)r_2(z)w_1(x)w_2(z)$
4. $r_2(z)r_1(x)w_2(z)w_1(x)$
5. $r_1(x)r_2(z)w_2(z)w_1(x)$
6. $r_2(z)r_1(x)w_1(x)w_2(z)$

The first two are serial, the third and fourth are nested, and the last two are interleaved.

Concurrency control must reject all schedules that lead to anomalies.

Definition

The *scheduler* is a component that accepts or rejects operations requested by the transactions.

The *serial schedule* is a schedule in which the actions of each transaction occur in a contiguous sequence.

A serializable schedule leaves the database in the same state as some serial schedule of the same transactions, making it correct. To introduce other classes, two initial assumptions are made:

- The transactions are observed a posteriori.
- Commit-projection: consider only the committed transactions.

2.4 View-serializability

Definition

$r_i(x)$ *reads-from* $w_j(x)$ when $w_j(x)$ precedes $r_i(x)$ and there is no $w_k(x)$ in S between $r_i(x)$ and $w_j(x)$.

$w_i(x)$ in a schedule S is a *final write* if it is the last write on x that occurs in S.

Two schedules are said to be *view-equivalent* ($S_i \approx_V S_j$) if they have:

1. The same operations.
2. The same reads-from relationships.
3. The same final writes.

A schedule is view-serializable (VSR) if it is view-equivalent to a serial schedule of the same transactions. The value written by $w_j(x)$ could be uncommitted when $r_i(x)$ reads it, but we are certain that it will be committed (commit-projection hypothesis).

Example : Consider the following schedules:

- $S_1 : w_0(x)r_2(x)r_1(x)w_2(x)w_2(z)$
- $S_2 : w_0(x)r_1(x)r_2(x)w_2(x)w_2(z)$
- $S_3 : w_0(x)r_1(x)w_1(x)r_2(x)w_1(z)$
- $S_4 : w_0(x)r_1(x)w_1(x)w_1(z)r_2(x)$
- $S_5 : r_1(x)r_2(x)w_1(x)w_2(x)$
- $S_6 : r_1(x)r_2(x)w_2(x)r_1(x)$
- $S_7 : r_1(x)r_1(y)r_2(z)r_2(y)w_2(y)w_2(z)r_1(z)$

Only S_2 and S_3 are serial. S_1 is view-equivalent to the serial schedule S_2 (thus, view-serializable). S_3 is not view-equivalent to S_2 (due to different operations) but is view-equivalent to the serial schedule S_4, making it view-serializable.

S_5 corresponds to a lost update, S_6 corresponds to a non-repeatable read, and S_7 corresponds to a phantom update. All these schedules are non view-serializable.

Additionally, consider the following schedules:

- $S_a : w_0(x)r_1(x)w_0(z)r_1(z)r_2(x)w_0(y)r_3(z)w_3(z)w_2(y)w_1(x)w_3(y)$
- $S_b : w_0(x)w_0(z)w_0(y)r_2(x)w_2(y)r_1(x)r_1(z)w_1(x)r_3(z)w_3(z)w_3(y)$
- $S_c : w_0(x)w_0(z)w_0(y)r_2(x)w_2(y)r_3(z)w_3(z)w_3(y)r_1(x)r_1(z)w_1(x)$

S_a and S_b are view-equivalent because all the reads-from relationships and final writes are the same. Specifically, we have:

- Reads-from: $r_1(x)$ from $w_0(x)$, $r_1(z)$ from $w_0(z)$, $r_2(x)$ from $w_0(x)$, $r_3(z)$ from $w_0(z)$.
- Final writes: $w_1(x)$, $w_3(y)$, $w_3(z)$.

However, S_a and S_c are not view-equivalent because not all the reads-from relationships are the same.

Deciding if a generic schedule is in VSR is an $\mathcal{NP}$-complete problem. Therefore, a more stringent definition is needed, which is easier to check. This new definition may result in rejecting some schedules that would be acceptable under view-serializability but not under the stricter criterion.

2.5 Conflict-serializability

Definition

Two operations o_i and o_j $(i \neq j)$ are in *conflict* if they address the same resource and at least one of them is write. There are two possible cases:

1. Read-write conflicts ($r - w$ or $w - r$).
2. Write-write conflicts ($w - w$).

Two schedules are *conflict-equivalent* ($S_i \approx_C S_j$) if S_i and S_j contain the same operations and in all the conflicting pairs the transactions occur in the same order.

A schedule is *conflict-serializable* (CSR) if and only if it is conflict equivalent to a serial schedule of the same transactions.

It is observed that View-Serializability (VSR) is a strict subset of Conflict-Serializable (CSR), and CSR implies VSR.

Proof VSR is a subset of CSR : Consider the schedule $S = r_1(x)w_2(x)w_1(x)w_3(x)$, which is:

- View-serializable.
- Not conflict-serializable.

It can be verified that there is no conflict-equivalent serial schedule. ■

Proof CSR implies VSR : Assuming $S_1 \approx_C S_2$, we can prove that $S_1 \approx_V S_2$. S_1 and S_2 must have:

- The same final writes: if not, there would be at least two writes in 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 at least one pair of conflicting operations in a different order, and therefore $\approx_C$ would be violated. ■

To assess view-serializability, a conflict graph is constructed, with one node for each transaction T_i, and an arc from T_i to T_j if there exists at least one conflict between an operation o_i of T_i and an operation o_j of T_j such that o_i precedes o_j.

Theorem

A schedule is *conflict-serializable* if and only if its conflict graph is acyclic.

Example : Consider the schedule

$$S : w_0(x)r_1(x)w_0(z)r_1(z)r_2(x)w_0(y)r_3(z)w_3(z)w_2(y)w_1(x)w_3(y)$$

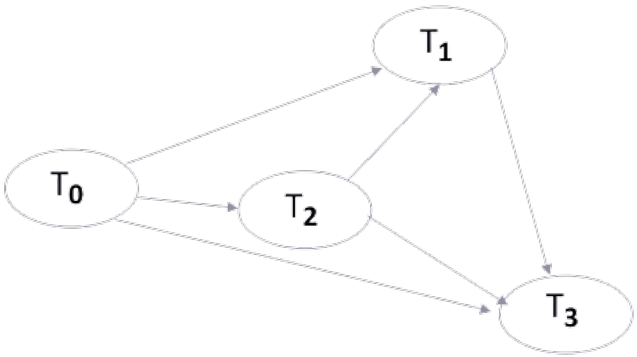
To test conflict serializability, follow these steps:

1. Create nodes based on the number of transactions in the schedule.
2. Group operations based on the requested resource.

3. Check all write-write and read-write relationships in each subset and add arcs accordingly.

For the given example, we obtain:

- $x : w_0 r_1 r_2 w_1$
- $y : w_0 w_2 w_3$
- $z : w_0 r_1 r_3 w_3$



Since there are no cycles in the graph, the schedule is conflict-serializable.

Proof CSR implies acyclicity of the conflict graph : Consider a schedule S in CSR. As such, it is $\approx_C$ to a serial schedule. Without loss of generality we can label the transactions of S to say that their order 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 the conflict 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 $i > j$. ■

Proof acyclicity of the conflict graph implies CSR : If S's graph is acyclic then it induces a topological (partial) ordering on its nodes. The same partial order exists on the transactions of S. Any serial schedule whose transactions are ordered according to the partial order is conflict-equivalent to S, because for all conflicting pairs (i, j) it is always $i < j$. ■

2.6 Concurrency control in practice

Checking conflict-serializability would be efficient if the conflict graph were known from the outset, but in practice, it is typically not. Therefore, a scheduler must work online. It is impractical to maintain, update, and check the conflict graph at each operation request. Additionally, the assumption that concurrency control can rely solely on the commit-projection of the schedule is unrealistic because aborts do occur. Thus, an online scheduler needs a simple decision criterion that avoids as many anomalies as possible with minimal overhead.

In the realm of online concurrency control, considering arrival sequences is crucial. The concurrency control system translates an arrival sequence into an effective a posteriori schedule. Two main families of techniques are commonly employed for online scheduling:

- Pessimistic techniques (locks): if a resource is taken, make the requester wait or pre-empt the holder.
- Optimistic techniques (timestamps and versions): serve as many requests as possible, possibly using out-of-date versions of the data.

In practice, commercial systems often combine elements from both pessimistic and optimistic approaches to leverage the strengths of each.

2.7 Locking

The prevalent method in commercial systems is the use of locking. A transaction is considered well-formed concerning locking if:

- Read operations are preceded by r_lock (shared) and followed by unlock.
- Write operations are preceded by w_lock (exclusive) and followed by unlock.

In both cases, unlocking can be delayed with respect to the completion of the operations. Consequently, every object can be in one of three states: free, r_locked, or w_locked.

Transactions that first read and then write an object may acquire a w_lock already when reading or acquire a r_lock first and then upgrade it into a w_lock (escalation).

The lock manager receives requests from transactions and allocates resources based on the conflict table:

	Resource status		
Request	*FREE*	*R_LOCKED*	*W_LOCKED*
r_lock	✓ R_LOCKED	✓ R_LOCKED$(n++)$	✗ W_LOCKED
w_lock	✓ W_LOCKED	✗ R_LOCKED	✗ W_LOCKED
unlock	ERROR	✓ $n--$	✓ FREE

Example : Consider a schedule with three transactions and the following operations:

$$r_1(x)w_1(x)r_2(x)r_3(y)w_1(y)$$

The resulting locks are as follows:

- $r_1(x)$: $r_1\_lock(x)$ request → Ok → x is *r_locked* with $n_x = 1$.
- $w_1(x)$: $w_1\_lock(x)$ request → Ok → x is *w_locked*.
- $r_2(x)$: $r_1\_lock(x)$ request → No, because x is *w_locked* → T_2 waits.
- $r_3(y)$: $r_3\_lock(y)$ request → Ok → y is *r_locked* with $n_y = 1$ and then T_3 unlocks y.
- $w_1(y)$: $w_1\_lock(y)$ request → Ok → y is *w_locked* and then x and y are freed.

The resulting a posteriori schedule becomes:

$$r_1(x)w_1(x)r_3(y)w_1(y)r_2(x)$$

Here, transaction two is delayed.

The locking system is implemented via lock tables, which are hash tables indexing lockable items via hashing. Each locked item has an associated linked list, with each node representing the transaction that requested the lock, the lock mode, and the current status. Each new lock request for the data item is appended to the list.

2.8 Two-phase locking

The previously presented locking mechanism does not eliminate anomalies caused by non-repeatable reads. To address this issue, a two-phase rule can be employed, requiring that a transaction cannot acquire any other lock after releasing one. This approach involves three phases: acquiring all locks, executing operations, and finally, unlocking.

Definition

The class of *two-phase locking* is the set of all schedules generated by a scheduler that:

- Only processes well-formed transactions.
- Grant locks according to the conflict table.
- Checks that all transactions apply the two-phase rule.

We have that 2PL is a strict subset of CSR and also that 2PL implies CSR.

Proof 2PL implies CSR : We assume that a schedule S is 2PL. Consider, for each transaction, the moment in which it holds all locks and is going to release the first one. We sort the transactions by this temporal value and consider the corresponding serial schedule $S^{'}$. We want to prove by contradiction that S is conflict-equivalent to $S^{'}$:

$$S^{'} \approx_C S, \ldots$$

Consider a generic conflict $o_i \rightarrow o_j$ in $S^{'}$ with $o_i \in T_i$, $0_j \in T_j$, and $i < j$. By definition of conflict, o_i and o_j address the same resource r, and at least one of them is write. The two operations cannot occur in reverse order of S. This proves that all 2PL schedules are view-serializable. ■

In this state, the remaining anomalies are limited to phantom inserts (requiring predicate locks) and dirty reads.

2.9 Strict two-phase and predicate locking

Definition

In *strict two-phase locking* (or long duration locks) we also have that locks held by a transaction can be released only after commit or rollback.

This locking variant is employed in many commercial database management systems (DBMS) when a high level of isolation is necessary.

To counteract phantom inserts, a lock should also be applied to future data using predicate locks. When a predicate lock is placed on a resource, other transactions are restricted from inserting, deleting, or updating any tuple that satisfies this predicate.

2.10 Isolation levels in SQL '99

SQL defines transaction isolation levels that specify the anomalies to be prevented when running at each level:

	Dirty read	Non-repeatable read	Phantoms
Read uncommitted	✓	✓	✓
Read committed	✗	✓	✓
Repeatable reads	✗	✗	✓(insert)
Serializable	✗	✗	✗

The four levels are implemented respectively with: no read locks, normal read locks, strict read locks, and strict locks with predicate locks. "Serializable" is not the default because its strictness can lead to the following problems:

- Deadlock: two or more transactions are in endless mutual wait.
- Starvation: a single transaction is in endless wait.

2.11 Deadlocks

A deadlock arises when concurrent transactions hold and request resources held by other transactions.

Definition

A *lock graph* is a bipartite graph in which nodes are resources or transactions and arcs are lock requests or lock assignments.

A *wait-for graph* is a graph in which nodes are transactions and arcs are waits for relationships.

A deadlock is indicated by a cycle in the wait-for graph of transactions.

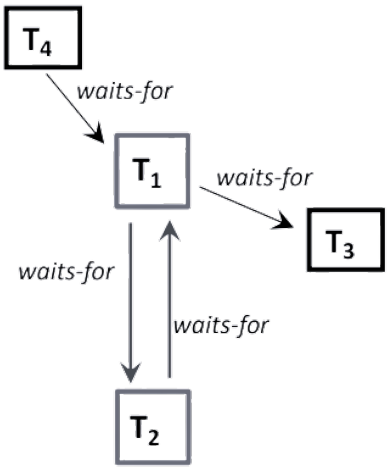


Figure 2.1: An example of deadlock in the wait-for graph

It is possible to solve deadlocks in three different ways:

- Timeout: a transaction is terminated and restarted after a specified waiting time, determined by the system manager.
- Deadlock prevention: kills transactions that could cause cycles. It is implemented in two ways:

1. Resource-based prevention puts restrictions on lock requests. The idea is that every transaction requests all resources at once, and only once. The main problem is that it's not easy for transactions to anticipate all requests.
2. Transaction-based prevention puts restrictions on transactions' IDs. Assigning IDs to transactions incrementally allows to give an age to each one. It is possible to choose to kill the holding transaction (preemptive) or the requesting one (non-preemptive). The main problem is that the number of killings is too big.

- Deadlock detection: it can be implemented with various algorithms and used for distributed resources.

Definition

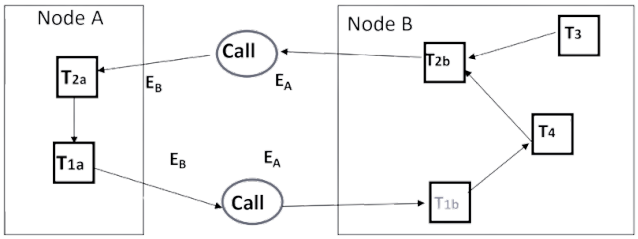
The *distributed dependency graph* is a wait-for graph where external call nodes represent a sub-transaction activating another sub-transaction at a different node.

The arrow shows a wait-for relation among local transactions. If one term is an external call, either the source is waited for by a remote transaction or waits for a remote transaction. The Obermarck's algorithm needs the following assumptions:

- Transactions execute on a single main node.
- Transactions may be decomposed in sub-transactions running on other nodes.
- When a transaction spawns a sub-transaction it suspends work until the latter completes.
- Two wait-for relationships:
  - T_i waits for T_j on the same node because T_i needs a datum locked by T_j.
  - A sub-transaction of T_i waits for another sub-transaction of T_i running on a different node.

The goal of this algorithm is to detect a potential deadlock looking only at the local view of a node. Nodes exchange information and update their local graph based on the received information. Node A sends its local info to a node B only if: it contains a transaction T_i that is waited for from another remote transaction and waits for a transaction T_j active on B and $i > j$.

Example : Consider the given distributed dependency graph:



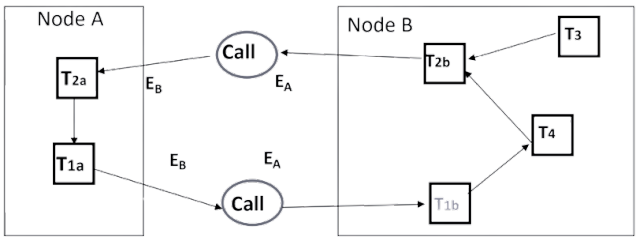
In this graph, a potential deadlock is indicated by cycles. Specifically, we observe that T_{2a} waits for T_{1a} (data lock), which, in turn, waits for T_{1b} (call), leading to a sequence where T_{2b} (data locks) waits for T_{2a} (call).

In this case the node A dispatches information to B, in fact we have $E_b \to T_2 \to T_1 \to E_b$ and node B cannot dispatch information to A, because the forwarding rule is not respected: $E_a \to T_1 \to T_2 \to E_a$.

The Obermarck's algorithm runs periodically at each node and consists in four steps:

1. Obtain graph info from the previous nodes.
2. Update the local graph by merging the received information.
3. Check for cycles among transactions, indicating potential deadlocks. If found, select one transaction in the cycle and terminate it.
4. Send the updated graph info to the next nodes.

Example : Given the following distributed system:



Let's apply the Obermarck's algorithm:

1. Use the forwarding rule, in this case we have:
   - At Node A: $E_b \to T_2 \to T_1 \to E_b$ info sent to Node B
   - At Node B: $E_a \to T_1 \to T_2 \to E_a$ info not sent $(i < j)$.
2. At node B there is the updated info $E_b \to T_2 \to T_1 \to E_b$. and it is added to the wait-for graph.
3. At node B a deadlock is detected (cycle between T_1 and T_2) and T_1, T_2 or T_4 are killed.
4. Updated information are sent to all nodes.

Four variants based on different conventions regarding message conditions and receivers.

	Message condition	**Message receiver**
A	$i > j$	Following node
B	$i > j$	Preceding node
C	$i < j$	Following node
D	$i < j$	Preceding node

In practical scenarios, the likelihood of encountering deadlocks (n^{-2}) is considerably lower than the probability of conflicts (n^{-1}). Various techniques are employed to minimize the occurrence of deadlocks:

- Update lock: the most frequent deadlock occurs when two concurrent transactions start by reading the same resources and then decide to write and try to upgrade their lock to write on the resource. To avoid this situation, systems offer the update lock, that is used by transactions that will read and then write. The lock table become:

	Resource status			
Request	*FREE*	*SHARED*	*UPDATE*	*EXCLUSIVE*
Shared lock	✓	✓	✓	×
Update lock	✓	✓	×	×
Exclusive lock	✓	×	×	×

- Hierarchical lock: locks can be specified with different granularity. The objective of this is to lock the minimum amount of data and recognize conflicts as soon as possible. The method used to do so consists in asking locks on hierarchical resources by requesting resources top-down until the right level is obtained and releasing locks bottom-up. This is done by using five locking modes: shared, exclusive, ISL (intention of locking a sub-element of the current element in shared mode), IXL (intention of locking a sub-element of the current element in exclusive mode), and SIXL (lock of the element in shared mode with intention of locking a sub-element in exclusive mode). The lock table is modified as follows:

	Resource status					
Request	*FREE*	*ISL*	*IXL*	*SL*	*SIXL*	*XL*
ISL	✓	✓	✓	✓	✓	×
IXL	✓	✓	✓	×	×	×
SL	✓	✓	×	✓	×	×
SIXL	✓	✓	×	×	×	×
XL	✓	×	×	×	×	×

Example : Given a table X with eight tuples divided in two pages:

P1	**P2**
$t1$	$t5$
$t2$	$t6$
$t3$	$t7$
$t4$	$t8$

And two transactions with the following schedules:

$$T_1 = r(P1)\ w(t3)\ r(t8)$$

$$T_2 = r(t2)\ r(t4)\ w(t5)\ w(t6)$$

We can see that they are not in a read-write conflict (because they are independent of the order). Without hierarchical locking both transactions needs to operate on the same table, so the concurrency will be almost useless in this case. But with this technique, calling X the table, we have that the transaction acquires the following locks:

$$T_1 :\ IXL(root)\ SIXL(P1)\ XL(t3)\ ISL(P2)\ SL(t8)$$

$$T_2 :\ IXL(root)\ ISL(P1)\ SL(t2)\ SL(t4)\ IXL(P2)\ XL(t5)\ XL(t6)$$

2.12 Timestamps

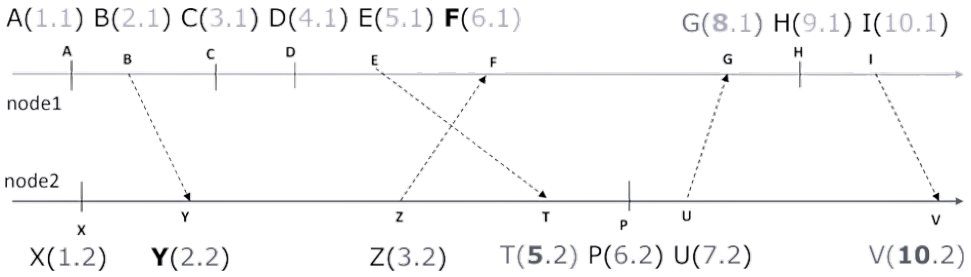
Locking, often referred to as pessimistic concurrency control, assumes collisions will occur, although in reality, collisions are infrequent. Optimistic concurrency control methods like timestamps can be employed to address this. Timestamps are identifiers that establish a total ordering of a system's events. Each transaction is assigned a timestamp representing its initiation time, enabling transactions to be ordered based on their timestamps. A schedule is

accepted only if it aligns with the serial ordering of transactions induced by their timestamps. Timestamps, given by a system's function upon request, have the syntax:

event-id.node-id

The synchronization algorithm, known as the Lamport method, relies on the send-receive of messages. It ensures that a message from the future cannot be received. If this occurs, the bumping rule is employed to adjust the timestamp of the receiving event beyond that of the sending event.

Example : Timestamp assignment at two different nodes might look like the following.



The scheduler uses two counters: one for writes (WTM(x)) and another for reads (RTM(x)). Read/write requests are tagged with the timestamp of the requesting transaction. For read operations:

- If $ts < \text{WTM}(x)$, the request is rejected, and the transaction is killed.
- Otherwise, access is granted, and $\text{RTM}(x) = \max(\text{RTM}(x), ts)$.

For write operations:

- If $ts < \text{RTM}(x)$ or $ts < \text{WTM}(x)$, the request is rejected, and the transaction is killed.
- Otherwise, access is granted, and $\text{WTM}(x) = ts$.

However, these rules may lead to excessive transaction killings.

Example : Assuming $\text{RTM}(x) = 7$ and $\text{WTM}(x) = 4$, consider the following schedule:

$$S = r_6(x)r_8(x)r_9(x)w_8(x)w_{11}(x)r_{10}(x)$$

Using timestamps, we obtain:

Request	Response	New value
$r_6(x)$	✓	-
$r_8(x)$	✓	$\text{RTM}(x) = 8$
$r_9(x)$	✓	$\text{RTM}(x) = 9$
$w_8(x)$	✗	T_8 killed
$w_{11}(x)$	✓	$\text{WTM}(x) = 11$
$r_{10}(x)$	✗	T_{10} killed

Comparing Two-Phase Locking (2PL) to Timestamps (TS) is challenging, and there is no subset relationship between the two. However, TS implies Conflict Serializable (CSR).

Proof TS implies CSR : Let S be a TS schedule of T_1 and T_2. Suppose S is not CSR, which implies that it contains a cycle between T_1 and T_2. S contains $op_1(x)$, $op_2(x)$ where at least one is a write. S contains also $op_2(y)$, $op_1(y)$ where at least one is a write. When $op_1(y)$ arrives:

- If $op_1(y)$ is a read, T_1 is killed by TS because it tries to read a value written by a younger transaction, so it is a contradiction.
- If $op_1(y)$ is a write, T_1 is killed no matter what $op_2(y)$ is, because it tries to write a value already read or written by a younger transaction, so it is a contradiction. ■

Basic TS-based control considers only committed transactions, ignoring aborted transactions. If aborts occur, dirty reads may happen. To handle dirty reads, a variant of basic TS must be used. A transaction T_i issuing $r_{ts}(x)$ or $w_{ts}(x)$ such that $ts > \text{WTM}(x)$ delays its read or write operation until the transaction T' that wrote the value of x has committed or aborted. This is similar to long-duration write locks.

Action	2PL	TS
Transaction	Wait	Killed and restarted
Serialization	Imposed by conflicts	Imposed by timestamp
Delay	Long (strict version)	Long
Deadlocks	Possible	Prevented

Since restarting a transaction is costlier than waiting, 2PL is preferable when used alone. Commercial systems often combine these techniques to leverage the best features of both. To reduce the number of killings, the Thomas rule can be used, altering the rule for write operations:

- If $ts < \text{RTM}(x)$ the request is rejected and the transaction is killed.
- If $ts < \text{WTM}(x)$ then our write is obsolete: it can be skipped.
- Else, access is granted, and we set $\text{WTM}(x) = ts$.

2.13 Multi-version timestamps

The concept of multi-versioning involves generating new versions with each write operation, and reads access the relevant version. Each write produces new copies, each with a new Write Timestamp ($\text{WTM}(x)$), ensuring that each object x always has $N \geq 1$ active versions. A unique global Read Timestamp ($\text{RTM}(x)$) is maintained, and old versions are discarded when there are no transactions requiring their values. In theory, the following rules can be applied:

- $r_{ts}(x)$ is always accepted. A copy x_k is selected for reading, where:
  - If $ts \geq \text{WTM}_N(x)$, then $k = N$.
  - Otherwise, k is chosen such that $\text{WTM}_k(x) \leq ts < \text{WTM}_{k+1}(x)$.
- $w_{ts}(x)$:
  - If $ts < \text{RTM}(x)$, the request is rejected.
  - Otherwise, a new version is created for timestamp ts (incrementing N).

Example : Assuming $\text{RTM}(x) = 7$, $N = 1$ and $\text{WTM}_1(x) = 4$, consider the schedule:

$$S = r_6(x)r_8(x)r_9(x)w_8(x)w_{11}(x)r_{10}(x)r_{12}(x)w_{14}(x)w_{13}(x)$$

Using multi-versioning, the results are:

Request	Response	New value
$r_6(x)$	✓	-
$r_8(x)$	✓	$\text{RTM}(x) = 8$
$r_9(x)$	✓	$\text{RTM}(x) = 9$
$w_8(x)$	✗	T_8 killed
$w_{11}(x)$	✓	$\text{WTM}_2(x) = 11,\ N = 2$
$r_{10}(x)$	✓ on $x_{(1)}$	$\text{RTM}(x) = 10$
$r_{12}(x)$	✓ on $x_{(2)}$	$\text{RTM}(x) = 12$
$w_{14}(x)$	✓	$\text{WTM}_3(x) = 14,\ N = 3$
$w_{13}(x)$	✓	$\text{WTM}_4(x) = 14,\ N = 4$

In practice, the rule set is modified slightly:

- $r_{ts}(x)$ is always accepted. A copy x_k is selected for reading, where:
  - If $ts \geq \text{WTM}_N(x)$, then $k = N$.
  - Otherwise, k is chosen such that $\text{WTM}_k(x) \leq ts < \text{WTM}_{k+1}(x)$.
- $w_{ts}(x)$:
  - If $ts < \text{RTM}(x)$ or $ts < \text{WTM}_N(x)$, the request is rejected.
  - Otherwise, a new version is created for timestamp ts (incrementing N).

Example : Assuming $\text{RTM}(x) = 7$, $N = 1$ and $\text{WTM}_1(x) = 4$, consider the schedule:

$$S = r_6(x)r_8(x)r_9(x)w_8(x)w_{11}(x)r_{10}(x)r_{12}(x)w_{14}(x)w_{13}(x)$$

Using multi-versioning, the results are:

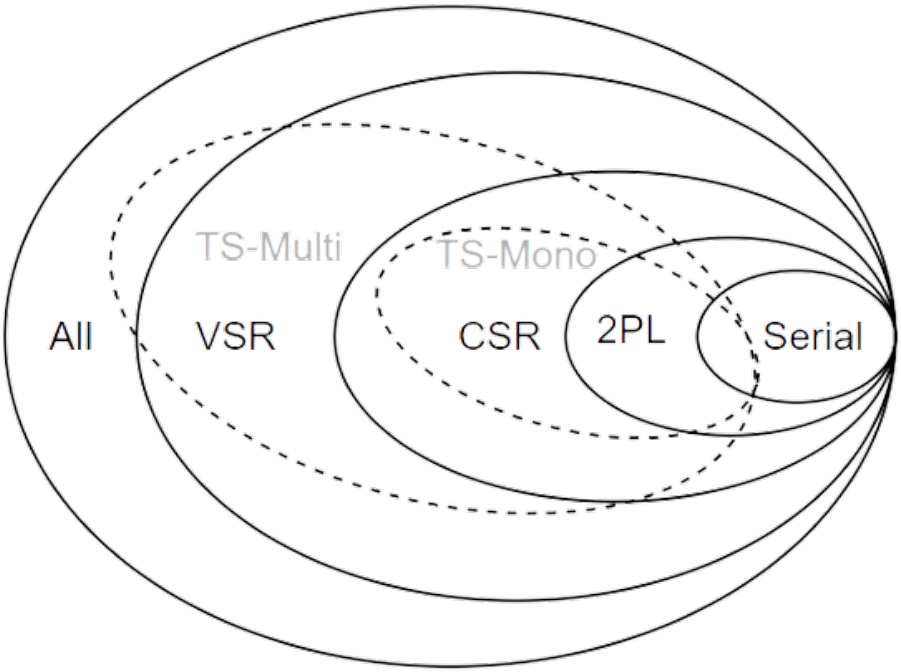
Request	Response	New value
$r_6(x)$	✓	-
$r_8(x)$	✓	$\text{RTM}(x) = 8$
$r_9(x)$	✓	$\text{RTM}(x) = 9$
$w_8(x)$	✗	T_8 killed
$w_{11}(x)$	✓	$\text{WTM}_2(x) = 11,\ N = 2$
$r_{10}(x)$	✓ on $x_{(1)}$	$\text{RTM}(x) = 10$
$r_{12}(x)$	✓ on $x_{(2)}$	$\text{RTM}(x) = 12$
$w_{14}(x)$	✓	$\text{WTM}_3(x) = 14,\ N = 3$
$w_{13}(x)$	✗	T_{13} killed

Isolation level introduced with TS-multi

The implementation of TS-multi opens the door to introducing another isolation level in the database management system (DBMS), known as snapshot isolation. In this level, only Write Timestamp (WTM(x)) is utilized. The rule applied in snapshot isolation dictates that every transaction reads the version consistent with its timestamp and defers writes until the end. If the scheduler detects conflicts between the writes of a transaction and the writes of other concurrent transactions after the snapshot timestamp, it aborts. It's essential to note that while snapshot isolation provides certain guarantees, it does not ensure serializability, and a new anomaly known as write skew (non-determinism) can occur.

2.14 Concurrency classes sets

Here is the final configuration for the sets of concurrency classes.



CHAPTER 3

Ranking

3.1 Introduction

It is possible to create ranking to get the best result out of the data using the multi-objective optimization (simultaneous optimization of different criteria).

A general multi-objective problem can be formulated as follows: given N objects described by d attributes, we need to find the best k objects.

The main approaches to solve the multi-objective optimization are:

- Ranking queries: selects the top k objects according to a given scoring function.
- Skyline queries: selects the set of non-dominated objects.

3.2 History

In the 13^{th} century this problem was introduced with the elections polls. Borda proposed to give a number of penalty points equivalent to the position of the electors in a voter's ranking, so the winner will be the candidate with the lowest overall penalty. Condorcet, instead, proposed to consider as winner the candidate who defeats every other candidate in pairwise majority rule election.

Example : Given ten voters and three candidates with the following votes:

1	2	3	4	5	6	7	8	9	10
A	A	A	A	A	A	C	C	C	C
C	C	C	C	C	C	B	B	B	B
B	B	B	B	B	B	A	A	A	A

For Borda we have:

- $A : 1 \cdot 6 + 3 \cdot 4 = 18$
- $B : 3 \cdot 6 + 2 \cdot 4 = 26$
- $C : 2 \cdot 6 + 1 \cdot 4 = 16$ (winner)

While for Condorcet we have that A wins in pairwise majority. So, the winner depends on the method used.

In 1950 Arrow proposed the axiomatic approach. He defined the aggregation as axioms and understood that a small set of natural requirements cannot be simultaneously achieved by any nontrivial aggregation function. So, he stated the Arrow's paradox: no rank-order electoral system can be designed that always satisfies these fairness criteria:

- No dictatorship (nobody determines, alone, the group's preference).
- If all prefer X to Y, then the group prefers X to Y.
- If, for all voters, the preference between X and Y is unchanged, then the group preference between X and Y is unchanged.

To solve this paradox, in the later years researchers tried to measure the values of all analyzed objects and created the metric approach. This consist in finding a new ranking R whose total distance to the initial rankings $R_1, \ldots, R_n$ is minimized. The distance between rankings can be found in several ways:

- Kendall tau distance $K(R_1, R_2)$, defined as the number of exchanges in a bubble sort to convert R_1 to R_2.
- Spearman's foot-rule distance $F(R_1, R_2)$, which adds up the distance between the ranks of the same item in the two rankings.

Finding an exact solution is computationally hard for Kendall tau (NP-complete), but tractable for Spearman's foot-rule (P time). These distances are related:

$$K(R_1, R_2) \leq F(R_1, R_2) \leq 2K(R_1, R_2)$$

And it is possible to find efficient approximation for $F(R_1, R_2)$.

3.3 Opaque rankings

The opaque rankings consider only the position and no other associated score. The algorithm called MedRank is based on the notion of median and provides an approximation of the foot-rule optimal aggregation. The inputs of this algorithm are: an integer k, and a ranked list $R_1, \ldots, R_m$ of N elements. The output is the top k elements according to median ranking. The idea of the algorithm is the following:

1. Use sorted accesses in each list, one element at a time, until there are k elements that occur in more than $m/2$ lists.
2. These are the top k elements.

Definition

The maximum number of sorted accesses made on each list is also called the *depth reached* by the algorithm.

Example : Suppose we have to sort the hotels based on three rankings criteria: price, rating, and distance. Using the MedRank algorithm we can make one sorted access at a time in each ranking and then look for hotels that appear in at least two rankings. We assume that price, rating and distance are opaque. The ranks of hotels are the following:

Price	**Rating**	**Distance**
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	...

If we use MedRank with $k = 3$, we will obtain the following rank:

Top k hotels	**Median rank**
Novotel	median$\{2, 3, 5\} = 3$
Hilton	median$\{4, 5, ?\} = 5$
Ibis	median$\{1, 5, ?\} = 5$

The depth in this case is equal to five.

Definition

An algorithm is *optimal* if its execution cost is never worse than any other algorithm on any input.

MedRank is not optimal, but it is instance optimal. This means that among the algorithms that access the lists in sorted order, this is the best possible algorithm on every input instance.

Definition

Let $\boldsymbol{A}$ be a family of algorithms, $\boldsymbol{I}$ a set of problem instances. Let cost be a cost metric applied to an algorithm-instance pair. Algorithm A^* is instance-optimal with respect to $\boldsymbol{A}$ and $\boldsymbol{I}$ for the cost metric cost if there exist constants k_1 and k_2 such that, for all $A \in \boldsymbol{A}$ and $I \in \boldsymbol{I}$:

$$\text{cost}(A^*, I) \leq k_1 \text{cost}(A, I) + k_2$$

If A^* is instance-optimal, then any algorithm can improve with respect to A^* by only a constant factor r, which is therefore called the optimality ratio of A^*. Instance optimality is a much stronger notion than optimality in the average or worst case.

3.4 Ranking queries

Ranking queries, also called top-k queries, aim to retrieve only the k best ansoftwareers from a potentially very large result set. Ranking is based on ordering objects based on their relevance.

Assume a scoring function S that assigns to each tuple t a numerical score for ranking tuples. The algorithm needs these inputs: cardinality k, dataset R, and a scoring function S. The output is the k highest-scored tuples with respect to S. The idea is the following:

1. For all tuples t in R compute $S(t)$.
2. Sort tuples based on their scores.
3. Return the first k highest-scored tuples.

This naïve approach is expensive for large dataset because it requires to sort a large amount of data. It is even worse if more than one relation is involved because it needs to join all tuples. So, we now know that two abilities are required:

- Ordering the tuples according to their scores. This is taken care of by

  ```
  ORDER BY
  ```

- Limiting the output cardinality to k tuples. This is taken care of by

  ```
  FETCH FIRST k ROWS ONLY
  ```

Example : Consider the following queries:

```
a)  SELECT *
FROM UsedCarsTable
WHERE Vehicle = 'Audi/A4'
AND Price <= 21000
ORDER BY 0.8*Price+0.2*Miles
b)  SELECT *
FROM UsedCarsTable
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price+0.2*Miles
```

The values 0.8 and 0.2, also called weights, are a way to normalize our preferences on price and mileage. The first query will likely miss some relevant ansoftwareers (near-miss). The second query will return all Audi/A4 in the dataset (information overload).

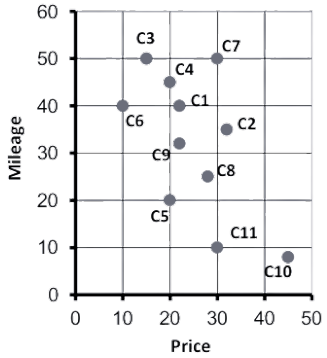
Only the first k tuples become part of the result. If more than one set of k tuples satisfies the ordering directive, any of such sets is a valid ansoftwareer (non-deterministic semantics).

To evaluate a top-k query it is possible to consider two basic aspects:

- Query type: one relation, many relations, and aggregate results.
- Access paths: no index, indexes on all/some ranking attributes.

In the simplest case, that is a top-k selection query with only one relation, if the input is sorted according to S it is possible to read only the first k tuples; If the tuples are not sorted we need to perform an in-memory sort (through the heap), that has a complexity of $O(N \log k)$.

Example : Consider the two-dimensional attribute space (Price, Mileage) of the previous example:



In the image each tuple is represented by a two-dimensional point (p, m), where p is the Price and m is the Mileage. Intuitively, minimizing $0.8 \cdot Price + 0.2 \cdot Mileage$ is equivalent to looking for points close to $(0, 0)$. Our preferences are essential to determine the result. Consider the line of equation:

$$0.8 \cdot Price + 0.2 \cdot Mileage = v$$

where v is a constant. This can also be written as:

$$Mileage = -4 \cdot Price + 5 \cdot v$$

from which we see that all the lines have a slope -4. By definition, all the points of the same line are equally good. If we change the weights, it is possible that the best choice also varies. The target of a query is not necessarily $(0, 0)$, rather it can be any point $q = (q_1, q_2)$.

In general, in order to determine the goodness of a tuple t, we compute its distance from the target point q: the lower the distance from q, the better. If we consider the distances the model become:

- An m-dimensional ($m \geq 1$) space $A = (A_1, A_2, \ldots, A_m)$ of ranking attributes.
- A relation $R(A_1, A_2, \ldots, A_m, B_1, B_2, \ldots)$, where $B_1, B_2, \ldots$ are other attributes.
- A target point $q = (q_1, q_2, \ldots, q_m)$, $q \in A$.
- A function $d : A \times A \to \mathbb{R}^+$, measuring the distance between points in A.

Definition

The top-k query is called *k-nearest neighbors* if given a point q, a relation R, an integer $k \geq 1$, and a distance function d it determines the k tuples in R that are closest to q according to d.

The distance function called *L_p-norms* is defined as:

$$L_p(t, q) = \left(\sum_{i=1}^{m} |t_i - q_i|^p \right)^{\frac{1}{p}}$$

Relevant cases of the L_p-norm are:

- Euclidean distance (ellipsoids): $L_2(t, q) = \sqrt{\sum_{i=1}^{m} |t_i - q_i|^2}$
- Manhattan distance (rhomboids): $L_1(t, q) = \sum_{i=1}^{m} |t_i - q_i|$
- Čebyšëv distance (rectangles): $L_\infty(t, q) = \max_i \{|t_i - q_i|\}$

The shape of the attribute space depends on the distance function and the weight associated to each coordinate:

- Euclidean distance (hyper-ellipsoids):

$$L_2(t, q; W) = \sqrt{\sum_{i=1}^{m} w_i |t_i - q_i|^2}$$

- Manhattan distance (hyper-rhomboids):

$$L_1(t, q; W) = \sum_{i=1}^{m} w_i \, |t_i - q_i|$$

- Čebyšëv distance (hyper-rectangles):

$$L_\infty(t, q; W) = \max_i \{w_i \, |t_i - q_i|\}$$

Definition

In a *top-k join query* we have $n > 1$ input relations and a scoring function S defined on the result of the join. The general formula is:

```
SELECT <attributes>
FROM R1,R2,...,Rn
WHERE <conditions>
ORDER BY S(p1,p2,...,pm) [DESC]
FETCH FIRST k ROWS ONLY
```

where $p_1, p_2, \ldots, p_m$ are the scoring criteria.

In the top-k $1-1$ join queries all the joins are on a common key attribute. It is possible to have two main scenarios:

- There is an index for retrieving tuples according to each preference.
- The relation is spread over several sites, each providing information only on part of the objects.

We make the following assumptions:

- Each input list supports sorted access: this means that each access returns the identifier of the next best object, its partial score p_j.
- Each input list supports random access: this means that each access returns the partial score of an object, given its identifier.
- The identifier of an object is the same across all inputs.
- Each input consists of the same set of objects.

Example : Given the following reviews on two different sites:

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

If we aggregate the two tables with the following query:

```
SELECT *
FROM EatWell EW, BreadAndWine BW
WHERE EW.Name = BW.Name
ORDER BY EW.Score + BW.Score DESC
FETCH FIRST 1 ROW ONLY
```

We will obtain the following result:

Name	Global score
Cheers!	16.8
The old mill	16.7
Da Gino	16.5
The canteen	16.0
Chez Paul	13.0
Let's eat!	12.4
Los pollos hermanos	11.5

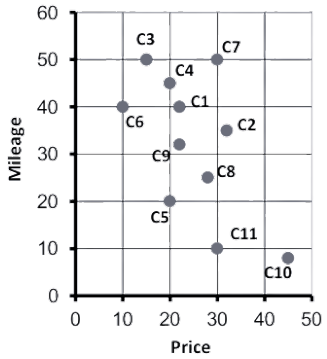
Note that the winner is not the best locally.

Each object o returned by the input L_j has an associated local/partial score $p_j(o) \in [0,1]$. For convenience, scores are normalized. The hypercube $[0,1]^m$ is called the score space. The point $p(o) = (p_1(o), p_2(o), \ldots, p_m(o)) \in [0,1]^m$ is the map of o into the score space. The global score $S(o)$ of o is computed by means of a scoring function S that combines in some way the local scores of o:

$$S(o) \equiv S(p(o)) = S(p_1(o), p_2(o), \ldots, p_m(o))$$

with $S : [0,1]^m \to \mathbb{R}^+$.

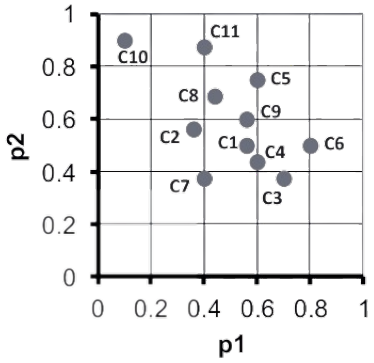
Example : Consider the attribute space $A = (Price, Mileage)$



If we choose $MaxP = 50,000$ and $MaxM = 80,000$, we have that a generic point o is mapped with these formulas:

$$p_1(o) = 1 - \frac{o.Price}{MaxP} \qquad p_2(o) = 1 - \frac{o.Mileage}{MaxM}$$

Graphically we have;



The most common scoring functions are:

- Average (SUM), that weighs preferences equally

$$\text{SUM}(o) \equiv \text{SUM}(p(o)) = p_1(o) + p_2(o) + \cdots + p_m(o)$$

- Weighted sum (WSUM), that weighs the preferences differently

$$\text{WSUM}(o) \equiv \text{WSUM}(p(o)) = w_1 p_1(o) + w_2 p_2(o) + \cdots + w_m p_m(o)$$

- Minimum (MIN), that considers the worst partial score

$$\text{MIN}(o) \equiv \text{MIN}(p(o)) = \min\{p_1(o), p_2(o), \ldots, p_m(o)\}$$

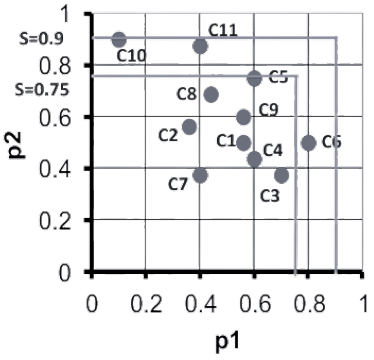
- Maximum (MAX), that considers the best partial score

$$\text{MAX}(o) \equiv \text{MAX}(p(o)) = \max\{p_1(o), p_2(o), \ldots, p_m(o)\}$$

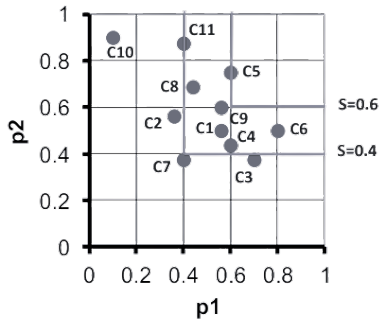
Note that even with the minimum scoring function we always want to retrieve the k objects with the highest global scores.

It is possible to define e iso-score curves in the score space, which are set of points with the same global score.

Example : In the previous graph we have that the iso-score curves for the maximum function are:



In the previous graph we have that the iso-score curves for the minimum function are:



B-zero algorithm

It is possible to efficiently compute the result of a top-k 1-1 join query using a maximum scoring function S using the B_0 algorithm. The inputs of this algorithm are: an integer $k \geq 1$, and a ranked list $R_1, \ldots, R_m$. The idea is:

1. Make k sorted accesses on each list and store objects and partial scores in a buffer B.
2. For each object in B, compute the MAX of its (available) partial scores.
3. Return the k objects with maximum score.

This algorithm is instance-optimal.

Example : Given the following reviews on two different sites:

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

If we want to use the algorithm B_0 with $k = 3$ we have to select the first three row from both tables (that are sorted), and by summing all the found values by identifier we obtain the final rank with the maximum value. In this case we obtain the following table:

Name	S
The old mill	9.2
Da Gino	9.0
The canteen	9.0
Cheers!	8.5

A-zero algorithm

The A-zero algorithm, also called Fagin's algorithm, works with every monotonic scoring function. The inputs of this algorithm are: an integer k, and a monotone function S combining ranked lists $R_1, \ldots, R_m$. The output is the top k (object-score) tuples. The idea of this algorithm is:

1. Extract the same number of objects by sorted accesses in each list until there are at least k objects in common.
2. For each extracted object, compute its overall score by making random accesses wherever needed.
3. Among these, output the k objects with the best overall score.

The complexity of this algorithm is $O(N^{\frac{m-1}{m}} k^{\frac{1}{m}})$, that is sublinear in the number N of objects. The stopping criterion is independent of the scoring function, and it is not instance optimal.

Example : Given the following hotel's review on different sites:

Name	**Cheapness**
Ibis	0.92
Etap	0.91
Novotel	0.85
Mercure	0.85
Hilton	0.825
Sheraton	0.8
Crillon	0.75
...	...

Name	**Rating**
Crillon	0.9
Novotel	0.9
Sheraton	0.8
Hilton	0.7
Ibis	0.7
Ritz	0.7
Lutetia	0.6
...	...

We decide to use the following scoring function:

$$0.5 \cdot cheapness + 0.5 \cdot rating$$

and $k = 2$. We iterate on the rows until k hotels are found in all columns. In this case we need to inspect 5 column to find at least two hotels (we found: Ibis, Novotel and Hilton). Now we have to complete the score of the remaining incomplete hotels, so we make random access to find the missing values. After summing completing the information about those hotels we can compute the final ranking with the give score function, and we obtain:

Top *k*	**Score**
Novotel	0.875
Crillon	0.825

The main drawback of this algorithm is that it is dependent from the accesses, so the specificity of some scoring function is not exploited at all, and the memory requirements can become prohibitive. It is possible to make some improvements, but only changing the stopping condition can really improve the complexity.

Threshold algorithm

The inputs of the threshold algorithm are: an integer k, and a monotone function S combining ranked lists $R_1, \ldots, R_m$. The output is the top k (object, score) tuples. The idea of this algorithm is:

1. Do sorted access in parallel in each list R_i.
2. For each object o, do random accesses in the other lists R_j, thus extracting score s_j.
3. Compute overall score $S(s_1, \ldots, s_m)$. If the value is among the k highest seen so far, and remember o.
4. Let s_{L_i} be the last score seen under sorted access for R_i.
5. Define threshold $T = S(s_{L_1}, \ldots, s_{L_m})$.
6. If the score of the k-th object is worse than T, go to step one.
7. Return the current top-k objects.

The threshold algorithm is instance optimal among all algorithms that use random and sorted accesses, and the stopping criterion depends on the function and not on the number of accesses.

Example : Given the following hotel's review on different sites:

Name	Cheapness
Ibis	0.92
Etap	0.91
Novotel	0.85
Mercure	0.85
Hilton	0.825
Sheraton	0.8
Crillon	0.75
...	...

Name	Rating
Crillon	0.9
Novotel	0.9
Sheraton	0.8
Hilton	0.7
Ibis	0.7
Ritz	0.7
Lutetia	0.6
...	...

We decide to use the following scoring function:

$$0.5 \cdot cheapness + 0.5 \cdot rating$$

and $k = 2$. We do a sorted access, and we put the hotels in the first row in the buffer with the mean value of both variables (the second can be found with a random access). The threshold point is $(0.92, 0.9)$ and has a value of 0.91, found with the scoring function. We continue to do this procedure, and we update the top k hotels only if the found hotel has a better score of at least one of the hotels in the buffer (note that we keep in the buffer only the top k hotels, while we delete the others). We stop the iteration when the value of all the objects in the buffer is greater or equal to the threshold value. In this example this happens after three iteration. In fact, we have that the buffer contains the following values:

Top k	Score
Novotel	0.875
Crillon	0.825

The threshold point has a value of 0.825

In general, TA performs much better than FA, since it can adapt to the specific scoring function. In order to characterize the performance of TA, we consider the middleware cost:

$$\text{cost} = SA \cdot c_{SA} + RA \cdot c_{RA}$$

where:

- SA is the total number of sorted accesses.
- RA is the total number of random accesses.
- c_{SA} is the base cost of sorted accesses.
- c_{RA} is the base cost of random accesses.

No random access algorithm

The NRA returns the top-k objects, but their scores might be uncertain. The idea at the base of this algorithm is to maintain, for each selected object o a lower bound and an upper bound on its score. So, we need an unlimited buffer to store the objects, sorted by decreasing lower bound values. This algorithm stops when the new object upper bound is less or equal to the worst lower bound of the best objects. The inputs are: an integer $k \geq 1$, a monotone function S combining ranked lists $R_1, \ldots, R_m$ and the output is the ranking without the score of the objects. The idea is the following:

1. Make sorted access to each list.
2. Store in B each retrieved object o and maintain $S^-(o)$ and $S^+(o)$ and a threshold τ.
3. Repeat from step one as long as:

$$S^-(B[k]) < \max\{\max\{S^+(B[i]), i > k\}, S(\tau)\}$$

Example : Given the following tables based on three different criterions:

R_1

ID	Score
o_1	1.0
o_7	0.9
o_2	0.7
o_6	0.2
...	...

R_2

ID	Score
o_2	0.8
o_3	0.75
o_4	0.5
o_1	0.4
...	...

R_3

ID	Score
o_7	0.6
o_2	0.6
o_3	0.5
o_5	0.1
...	...

We decide to use SUM as score function, and $k = 2$. For the first iteration we add to the buffer the objects in the first row with a lower bound corresponding to the sum of the score of each object and the upper bound corresponding to the sum of all the object in the row. The threshold is the sum of all the scores of the row:

Identifier	Lower bound	Upper bound
o_1	1.0	2.4
o_2	0.8	2.4
o_7	0.6	2.4

The threshold has a value of 2.4, and so we have that:

$$0.8 < \max\{2.4, 2.4\}$$

Since that the inequality is verified we have to do another iteration, that creates the following table:

Identifier	Lower bound	Upper bound
o_7	1.5	2.25
o_2	1.4	2.3
o_1	1.0	2.35
o_3	0.75	2.25

The threshold has a value of 2.25, and so we have that:

$$1.4 < \max\{2.35, 2.25\}$$

Since that the inequality is verified we have to do another iteration, that creates the following table:

Identifier	Lower bound	Upper bound
o_2	2.1	2.1
o_7	1.5	2.0
o_3	1.25	1.95
o_1	1.0	2.0
o_4	0.5	1.7

The threshold has a value of 1.7, and so we have that:

$$1.5 < \max\{2.0, 1.7\}$$

Since that the inequality is verified we have to do another iteration, that creates the following table:

Identifier	Lower bound	Upper bound
o_2	2.1	2.1
o_7	1.5	1.9
o_1	1.4	1.5
o_3	1.25	1.45
o_4	0.5	0.8
o_6	0.2	0.7
o_5	0.1	0.7

The threshold has a value of 1.7, and so we have that:

$$1.5 < \max\{1.5, 0.7\}$$

Since that the inequality is not verified the algorithm return the first two elements in the table.

NRA is instance optimal among all algorithms that do not make random accesses.

Summary

Algorithm	Scoring function	Data access	Notes
B_0	MAX	sorted	instance-optimal
FA	monotone	sorted and random	cost independent of scoring function
TA	monotone	sorted and random	instance-optimal
NRA	monotone	sorted	instance-optimal, uncertain

The main aspects of the ranking queries are:

- They are very effective identifying the best objects with respect to a specific scoring function.
- They have excellent control of the cardinality of the result.
- The computation is very efficient.
- It is easy to express the relative importance of attributes.
- For a user, it is difficult to specify a scoring function.

3.5 Skyline queries

The objective of the skyline queries is to find good objects according to several perspectives, that are based on the notion of dominance.

Definition

A tuple t *dominates* a tuple s ($t \prec s$) if and only if t in nowhere worse than s:

$$\forall i, 1 \leq i \leq m \rightarrow t[A_i] \leq s[A_i]$$

and better at least once:

$$\exists j, 1 \leq j \leq m \wedge t[A_j] < s[A_j]$$

The *skyline* of a relation is the set of its non dominated tuples

The convention on the skyline queries is that lower values are better than higher values. A tuple t is in the skyline if and only if it is the top-1 result with respect to at least one monotone scoring function. This means that the skyline is the set of potentially optimal tuples. Note that there is no scoring function that returns the same points that are in the skyline. A possible non-standard syntax for the skyline queries is the following:

```
SELECT <attributes>
FROM R1,R2,...,Rn
WHERE <conditions>
GROUP BY <conditions>
HAVING <conditions>
SKYLINE OF [DISTINCT] d1[MIN|MAX|DIFF], ..., dm[MIN|MAX|DIFF]
ORDER BY <conditions>
```

This query can be easily translated into a standard query, but the result is too slow and cannot be used in practice.

Block nested loop algorithm

The input of the block nested loop algorithm is a dataset D of multidimensional points, and the output is the skyline of D.

Algorithm 1 Block nested loop algorithm

```
1: W ← ∅
2: for every point p in D do
3:     if p not dominated by any point in W then
4:         remove from W the points dominated by p
5:         add p to W
6:     end if
7: end for
8: return W
```

The complexity is $O(n^2)$, where $n = |D|$. The complexity of this algorithm is very inefficient for large datasets.

Sort filter skyline algorithm

The input of the sort filter skyline algorithm is a dataset D of multidimensional points, and the output is the skyline of D.

Algorithm 2 Sort filter skyline algorithm

```
1: S ← D
2: W ← ∅
3: for every point p in S do
4:     if p not dominated by any point in W then
5:         add p to W
6:     end if
7: end for
8: return W
```

Where S is the list of sorted point in D by a monotone function. The initial sorting is useful for large datasets, thus this algorithm performs much better than the previous one, although the complexity is still $O(n^2)$.

Example : Given the following unordered dataset:

Name	Cost	Complaints
Crillon	0.25	0.1
Ibis	0.08	0.3
Hilton	0.175	0.3
Sheraton	0.2	0.2
Novotel	0.15	0.1

We decide to order them by the sum of cost and complaints, and the resulting dataset is:

Name	Cost	Complaints
Novotel	0.15	0.1
Crillon	0.25	0.1
Ibis	0.08	0.3
Sheraton	0.2	0.2
Hilton	0.175	0.3

The algorithm adds Novotel to the window, and the other hotel that is not dominated by Novotel is Ibis, so the skyline is composed by Novotel and Ibis.

Summary

The main aspects of the skyline queries are:

- They are effective in identifying potentially interesting objects if nothing is known about the preferences of a user.
- They are very simple to use.
- They return too many objects.
- The computation is not so efficient.
- They are agnostic with respect to user preferences

It is possible to extend the skyline queries adding the constraint of set of tuples dominated by less than k tuples. This method is called k-skyband.

3.6 Comparison between ranking and skyline

	Ranking queries	Skyline queries
Simplicity	No	Yes
Only interesting results	No	Yes
Control of cardinality	Yes	No
Trade-off among attributes	Yes	No

CHAPTER 4

Architectures and JPA

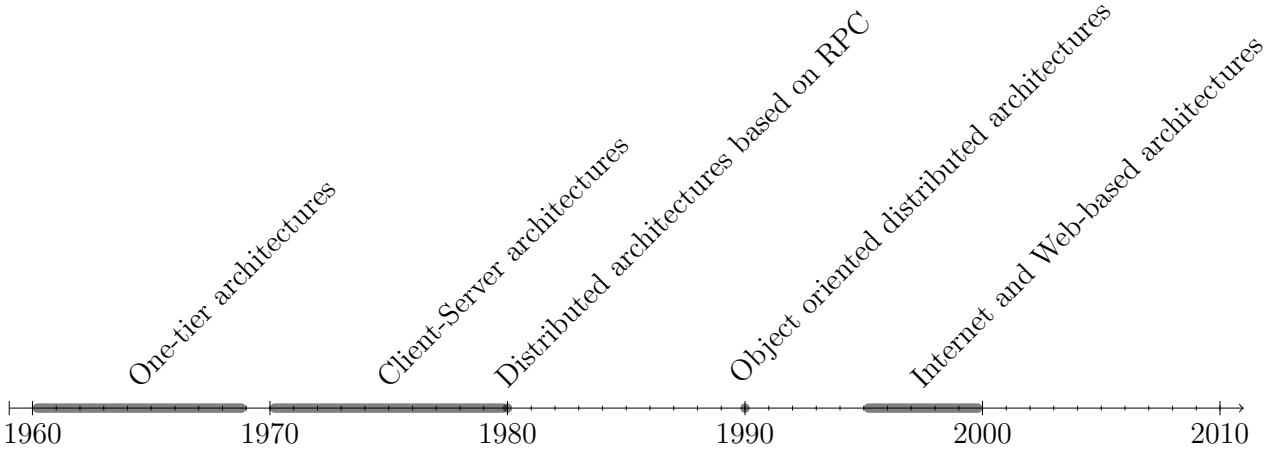
4.1 Introduction

Definition

The *architecture* is the union of hardware, software, and network resources.

The actual architectures are:

- Distributed web-based and mobile.
- Service oriented.
- Cloud and virtualized.
- Application service provisioning.



4.2 Client-server architecture

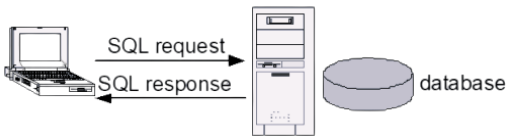


Figure 4.1: Client-server architecture

The hardware used by this the client-server architecture is:

- Server for data management.
- Client for presentation layout.

The software used by this type of architecture is:

- Client software sends requests to the server by means of SQL queries. The software in the client contains both business and presentation logic.
- Server software processes the query and responds sending the result set back to the client. The software in the server deals with data.

The network topology is based on a local are network with one or more servers and N clients.

4.3 Three-tier architecture

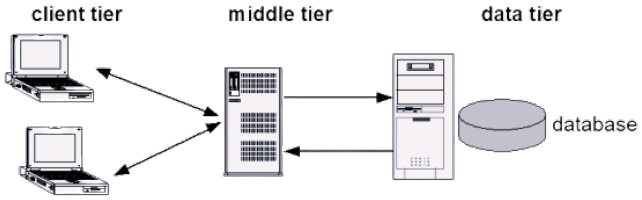


Figure 4.2: Three-tier architecture

The hardware used by the three-tier architecture is:

- Server for data management.
- Client for presentation layout.
- Middle tier to achieve a better separation between the client and the server.

This architecture has several variants, depending on the software features of the middle tier.

Web pure HTML three-tier architecture

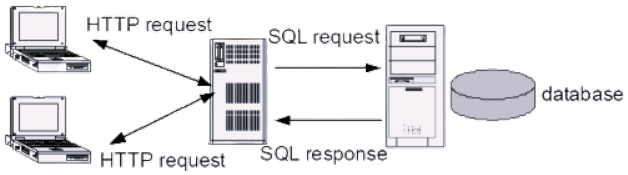


Figure 4.3: Web pure HTML three-tier architecture

In the pure HTML architecture the client is a standard web browser, and it has to cope with the presentation layout only (thin client). The middle tier:

- Includes a Web server that exploits HTTP.
- Hosts the business logic for dynamically generating content from the raw data of the data tier.
- Deals with the presentation layout.

Rich internet applications

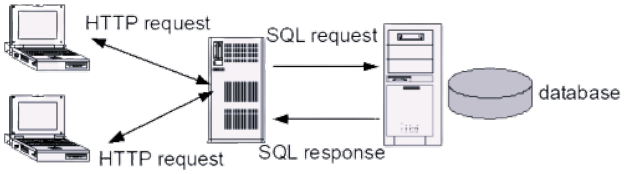


Figure 4.4: Rich internet applications three-tier architecture

The rich internet application architecture is the fusion of web and desktop applications (with JavaScript). In this case the client is called fat because it has standard communication protocol (HTTP, Web Socket), language (ECMAScript) and API (DOM, HTML5). The features of this architecture are:

- New interface event types, also specific to touch and mobile apps.
- Asynchronous interaction (AJAX).
- Client-side persistent data.
- Offline applications.
- Native multimedia and 3D support.

4.4 Java enterprise edition

Java enterprise edition is a platform aimed at the development, release and maintenance of three-tier web applications.

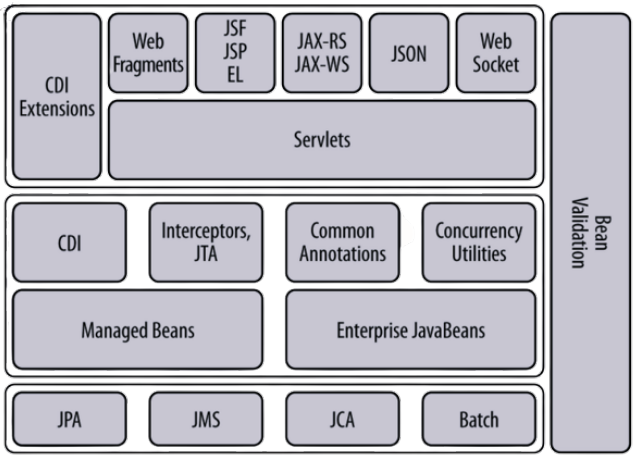


Figure 4.5: Jakarta EE stack

The main elements of this platform are:

- JDBC: this API was the first industry standard for database independent connectivity between Java and the database. Thanks to this API we can: establish a connection with a database, send SQL statements, and process the results.
- Servlet, that are the Java technology for the presentation tier in web application. The servlets: offer a component-based, platform independent method for building web-based applications, have access to other Java APIs to access enterprise databasesa and are executed within a container for concurrency control and lifecycle management.
- Jakarta Enterprise Beans: it is the technology used in the server. They enable the development of distributed, transactional, secure and portable applications. EJB components can be used by the web front-end to interact with the business functions and data access services.
- Java Persistence API: it is the specification of an interface for mapping relational data to object oriented data in Java. It comprises: the API implementation package javax.persistence, the Java compatible query language Java Persistence Query Language, and the specification of the metadata for defining object relational mappings.
- Java Transaction API: it is an API for managing transactions in Java. It allows a component to start, commit and rollback transactions in a resource agnostic way. With this technology, Java components can manage multiple resources in a single transaction with a unique interaction model.

Example : To extract data from a database using JPA we use:

```
public List<Mission> findMissionsByUser(int userId) {
Reporter reporter = em.find(Reporter.class, userId);
List<Mission> missions = reporter.getMissions();
```

```
return missions;
}
```

To insert data in a database using JPA we use:

```
public void createMission(Date startDate, int days, String destination, String
    ↪ description, int reporterId) {
Reporter reporter = em.find(Reporter.class, reporterId);
Mission mission = new Mission(startDate, days, destination, description,
    ↪ reporter);
reporter.addMission(mission);
em.persist(reporter);
}
```

To modify data in a database using JPA we use:

```
public void reportMission(int missionId, MissionStatus missionStatus) {
Mission mission = em.find(Mission.class, missionId);
mission.setStatus(MissionStatus.REPORTED);
}
```

4.5 JPA: Object Relational Mapping

The technique of bridging the gap between the object model and the relational model is known as object-relational mapping.

Definition

The challenge of mapping one model to the other lies in the concepts in one model for which there is no logical equivalent in the other is called *impedance mismatch*.

The automatic transformation of a model into another in managed by an element called mediator. The main differences between the object-oriented model and the relational one are the followings:

Object-oriented model	Relational model
Objects, classes	Tables, rows
Attributes, properties	Columns
Identity (physical memory address)	Primary key
Reference to other entity	Foreign key
Inheritance/Polymorphism	Not supported
Methods	Stored procedures, triggers
Code is portable	Not necessarily portable

The Java Persistence API bridges the gap between object-oriented domain models and relational database systems by using a Plain Old Java Object, that is a persistence model for object-relational mapping.

The main features of the Java Persistence API are:

- POJO Persistence: there is nothing special about the objects being persisted, any existing non-final object with a default constructor can be persisted.

- Non-intrusiveness: the persistence API exists as a separate layer from the persistent objects.
- Object queries: a powerful query framework offers the ability to query across entities and their relationships without having to use concrete foreign keys or database columns.

Definition

The *entity* is a class (Java bean) representing a collection of persistent objects mapped onto a relational table.

The *persistence unit* is the set of all classes that are persistently mapped to one database (analogous to the notion of db schema).

The *persistence context* is the set of all managed objects of the entities defined in the persistence unit (analogous to the notion of db instance).

The *managed entity* is an entity part of a persistence context for which the changes of the state are tracked.

The *entity manager* is the interface for interacting with a persistence context.

The *client* is a component that can interact with a persistence context, indirectly through an entity manager.

The entities are accessed through the entity manager interface of the Java Persistence API.

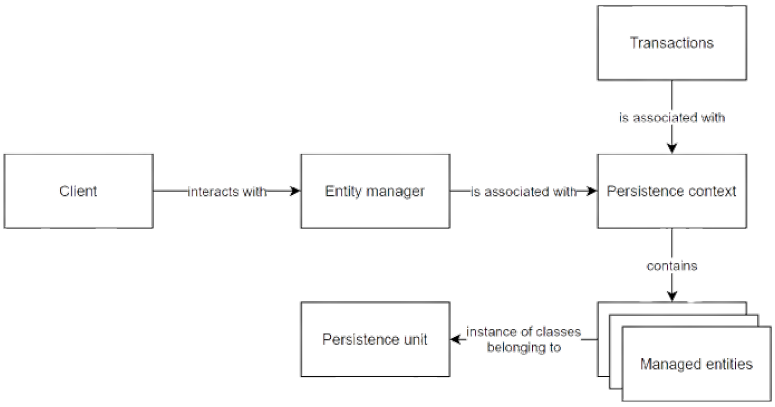


Figure 4.6: Structure of the Java Persistence API

The entity manager exposes all the operations needed to synchronize the managed entities in the persistence context to the database to:

- Persist an entity instance in the database.

```
public void persist(Object entity);
```

- Find an entity instance by its primary key.

```
public <T> T find(Class<T> entityClass, Object primaryKey);
```

- Remove an entity instance from the database.

```
public void remove(Object entity);
```

- Reset the entity instance from the database.

```
public void refresh(Object entity);
```

- Write the state of entities to the database immediately.

```
public void flush();
```

An entity is a Java Bean that gets associated to a tuple in a database. The persistent counterpart of an entity has a life longer than that of the application. The entity class must be associated with the database table it represents. An entity can enter a managed state, where all the modifications to the object's state are tracked and automatically synchronized to the database. The entities have the following properties: identification (primary key), nesting, relationship, referential integrity (foreign key), and inheritance. The entities must respect the following requirements:

- Must have a public or protected constructor with no arguments.
- Must not be final.
- No method or persistent instance variables may be final.
- Serializable interface must be implemented if you pass the entity by value.

In the database, objects and tuples have an identity (primary key), so an entity assumes the identity of the persistent data it is associated to. The primary key can be either simple or composite. To identify a primary key we use the @Id annotation, for the composite keys we use @EmbeddedId and @IdClass annotations.

Sometimes, applications do not want to explicitly manage uniqueness of data values. In this case the persistence provider can automatically generate an identifier for every entity instance of a given type. This persistence provider's feature is called identifier generation and is specified by the @GeneratedValue annotation. Applications can choose one of four different ID generation strategy:

1. Auto.
2. Table: identifiers are generated according to a generator table.
3. Sequence: the identifiers are generated with sequences.
4. Identity: the identifiers are generated with primary keys identity columns.

Example : An identifier can be generated as follows:

```
@Entity
public class Mission implements Serializable {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
private String city;
}
```

The attributes can be qualified with properties that direct the mapping between POJO and relational tables, such as:

- Large objects.
- Enumerated types: Java enumerations and strings.
- Temporal types.

The fetch policy can be lazy (retrieve item when needed) or eager (retrieve item as soon as possible). The first policy is used mainly for large objects.

Example : The qualifiers can be used as follows:

```
@Entity
public class Mission implements Serializable {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
@Temporal(TemporalType.DATE)
private Date date;
private MissionStatus status;
@Basic(fetch=FetchType.LAZY)
@LOB
private byte[] photo;
}
```

By default, entities are mapped to tables with the same name and their fields to columns with the same names, but it is possible to use some annotations to change this behavior. If the entity must be not persistent, we have to denote it with @Transient annotation.

Example : The mapping can be redefined as follows:

```
@Entity @Table(name="T_BOOKS")
public class Book {
@Column(name="BOOK_TITLE", nullable=false)
private String title;
private CoverType coverType;
private Date publicationDate;
@Transient
private BigDecimal discount;
}
```

Mapping

The relationships in any object model has four characteristics:

- Directionality: each of the two entities may have an attribute that enables access to the other one. If two entities are related with each other we have a bidirectional relationship; otherwise we have a unidirectional relationship. It is possible to have one or more references.
- Role: each entity in the relationship is said to play a role with respect to one direction of access. The entities are classified in source and target, based on the direction of the relationship.

- Cardinality: the number of entity instances that exist on each side of the relationship. There are four possibilities:
  - Many-to-one: many source entities, one target entity.
  - One-to-many: one source entity, many target entities.
  - One-to-one: one source entity, one target entity.
  - Many-to-many: many source entities, many target entities.
- Ownership: one of the two entity in the relationship is said to own the relationship. In the database, relationships are implemented by a foreign key column (called join column in JPA) that refers to the key of the referenced table. The entities that have a foreign key column is called the owner of the relationship and its side is called the owning side

One-to-many relationship

The one-to-many bidirectional relationship is defined with mappedBy and @ManyToOne, @OneToMany annotations, where:

- @ManyToOne annotation to the entity that participates with multiple instances. The entity that contains this annotation is the owner of the relationship.
- @JoinColumn annotation to specify the foreign key column of underlying table.

Example : First part of the definition of a one-to-many bidirectional mapping.

```
@Entity
public class Employee {
@Id private int id;
@ManyToOne
@JoinColumn(name="dept_fk")
private Department dept;
}
```

To achieve bi-directionality, the one-to-many mapping direction must be specified too. This is done by including a @OneToMany annotation in the entity that participates with one instance. The @OneToMany annotation is placed on a collection data member and comprises a mappedBy element to indicate the property that implements the inverse of the relationship.

Example : Second part of the definition of a one-to-many bidirectional mapping.

```
@Entity
public class Department {
@Id private int id;
@OneToMany(mappedBy="dept")
private Collection<Employee> employees;
}
```

Sometimes applications require the access to relationships only along one direction, and in this case the bidirectional mapping is not necessary.

Many-to-one relationship

The many-to-one relationship is defined in the same way as the one-to-many, but the source and the target are switched in the definition.

One-to-one relationship

To define a one-to-one relationship in JPA it is possible to choose between two different alternatives:

- We map the relationship as in the bidirectional case and use only the one-to-many direction.
- We do not map the collection attribute in the entity that participates with one instance and use a query instead to retrieve the correlated instances, relying on the inverse (many-to-one) relationship direction mapping.

The difference between @joincolumn and mappedby is the following:

- The annotation @JoinColumn indicates the foreign key column that implements the relationship in the database; such annotation is normally inserted in the entity owner of the relationship. Used to drive the generation of the SQL code to extract the correlated instances.
- The mappedBy attribute indicates that this side is the inverse of the relationship, and the owner resides in the other related entity. Used to specify bidirectional relationships. In absence of the mappedBy parameter the default JPA mapping uses a bridge table.

In a one-to-one mapping the owner can be either entity, depending on the database design. A one-to-one mapping is defined by annotating the owner entity with the @OneToOne annotation.

Example : First part of the definition of a one-to-one mapping.

```
@Entity
public class Employee {
@Id private int id;
@OneToOne
private ParkingSpace parkingSpace;
}
```

If the one-to-one mapping is bidirectional, the inverse side of the relationship needs to be specified too. In the non-owner entity we need both @OneToOne annotation and the mappedBy element (used to JPA to understand where to put the foreign key).

Example : Second part of the definition of a one-to-one mapping.

```
@Entity
public class ParkingSpace {
@Id private int id;
@OneToOne(mappedBy="parkingSpace")
private Employee employee;
}
```

Many-to-many relationship

In a many-to-many mapping there is no foreign key column, but there is a join table. Therefore, we can arbitrarily specify as owner either entity.

Example : First part of the definition of a many-to-many mapping.

```
@Entity
public class Employee {
@Id private int id;
@ManyToMany
private Collection<Project> projects;
}
```

If the many-to-many mapping is bidirectional, the inverse side of the relationship needs to be specified too. In the non-owner entity we need both @OneToOne annotation and the mappedBy element.

Example : First part of the definition of a many-to-many mapping.

```
@Entity
public class Project {
@Id private int id;
@ManyToMany(mappedBy="projects")
private Collection<Employee> employees;
}
```

The logical model of a many-to-many relationship requires a bridge table (join table in JPA).

Example : The non-default mapping of the entity to the bridge table is specified via annotation.

```
@Entity
public class Employee {
@Id private long id;
private String name;
@ManyToMany
@JoinTable(name="EMP_PROJ",
           joinColumns=@JoinColumn(name="EMP_ID"),
           inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
private Collection<Project> projects;
}
```

Relationship fetch mode

When the fetch mode is not specified, by default:

- A single-valued relationship is fetched eagerly.
- Collection-valued relationships are loaded lazily

In case of bidirectional relationships, the fetch mode might be lazy on one side but eager on the other. Note that the best practice is to consider lazy loading as the most appropriate mode for all relationships because if an entity as many single-valued relationships that are not all used by applications, the eager mode may incur performance penalties.

Example : The annotation used to define the lazy loading.

```
@Entity
public class Employee {
@Id private int id;
@OneToOne(fetch=FetchType.LAZY)
private ParkingSpace parkingSpace;
}
```

The directive to lazily fetch an attribute is meant only to be a hint to the persistence provider, that can still use an eager policy. The other way round is not true: the eager policy cannot be replaced with a lazy one by the provider.

Cascading operations

By default, every entity manager operation will not cascade to other entities that have a relationship with the entity that is being operated on. In some cases we want to have the propagation of the changes to the entities in relationship with the modified one. It is possible to do so by activating manual cascading.

Example : Activation of manual cascading:

```
Employee emp = new Employee();
Address addr = new Address();
emp.setAddress(addr);
em.persist(addr);
em.persist(emp);
```

With this mode activated, when the entity manager adds the Employee instance to the persistence context, it navigates the address relationship looking for a new Address entity to manage as well. The Address instance must be set on the Employee instance before invoking persist() on the employee object. If an Address instance has been set on the Employee instance and not persisted explicitly or implicitly via cascading, an error occurs.

The cascade attribute is used to define when operations should be automatically cascaded across relationships. It accepts several values:

- Persist.
- Refresh.
- Remove.
- Merge.
- Detach.

If we want to use all the previous operations we can use the operator "all".

Example : Activation of manual cascading of type persist:

```
@Entity
public class Employee {
@ManyToOne(cascade=CascadeType.PERSIST)
Address address;
}
```

The cascade settings are unidirectional, so they must be set on both side if we want a bidirectional behavior.

JPA also supports an additional remove cascading called "orphanRemoval". It is used in @OneToOne and @OneToMany annotations for privately owned parent-child relationship in which every child entity is associated only to one parent entity through just one relationship. This operation causes the child entity to be removed when the parent-child relationship is broken either by:

- Removing the parent or by setting to null the attribute that holds the related entity.
- In the one-to-many case, by removing the child entity from the parent's collection.

The difference between the attribute "CascadeType.REMOVE" and the mode "orphanRemoval" is that if we set manually the value of an entity to null, only "orphanRemoval" will automatically remove the linked entities from the database.

4.6 JPA: entity manager

The entity instances are plain Java objects, and for this reason they not become managed until the application invokes an API method to initiate the process. The entity manager is the central authority for all persistence actions. In particular, it manages the mapping and APIs for interacting with the database and the objects.

Definition

The *persistence context* is the fundamental and exclusive concept of Java Persistence API: it is a kind of main memory database that holds the objects in the managed state.

A managed object is tracked, so all the modifications to its state are monitored for automatic alignment to the database. Database writes by default occur asynchronously, and when they happen the persistence context must hook up to a transaction. So, we have that a managed entity has two lives: one as a Java object and one as a relational tuple bound to it. Such a binding exists only inside the persistence context. When the POJO exits the persistence context the binding breaks: it gets untracked and no longer synchronized to the database. Note that the application never sees the persistence context, it interacts only with the entity manager.

Interface

The interface of the entity manages exposes the following methods:

- Makes an entity instance become part of the persistence context.

  ```
  public void persist(Object entity);
  ```

- Finds an entity instance by its primary key.

```
public <T> T find(Class<T> entityClass, Object primaryKey);
```

- Removes an entity instance from the persistence context and thus from the database.

```
public void remove(Object entity);
```

- Resets the state of entity instance from the content of the database.

```
public void refresh(Object entity);
```

- Writes the state of entities to the database as immediately as possible.

```
public void flush();
```

When an entity is first instantiated, it is in the transient state since the entity manager does not know it exists yet. Transient entities are not part of the persistence context associated with the entity manager.

Example : A new POJO can be created in the following way:

```
Employee emp = new Employee(ID, "John Doe");
```

To make the transient entities managed we have to use the method persist() of the entity manager. Note that when an entity becomes managed, all the changes apply to it will apply also to the database itself, and the other way round. The managed entity and the corresponding tuple become associated until the entity exits the managed state. It is possible to call persist() on a managed entity: this will trigger the cascade process.

Example : A new POJO can be created and later made managed in the following way:

```
Employee emp = new Employee(ID, "John Doe");
em.persist(emp);
```

We can find an entity with find() method. It takes as an input the class of the entity that is being sought and the primary key value that identifies the desired entity instance. When the call completes, the returned object will be managed. If the entity instance is not found, then the find() method returns null.

Example : Finding an entity

```
Employee emp = em.find(Employee.class, ID);
```

We can remove an entity with remove() method. It breaks the association between the entity and the persistence context. When the transaction associated with the entity manager's persistence context commits or the entity manager flush() method is called, the tuple associated with the entity is scheduled for deletion from the database. The entity still exists, but its changes are no longer tracked for being synchronized to the database.

Example : Removing an entity

```
em.remove(emp);
```

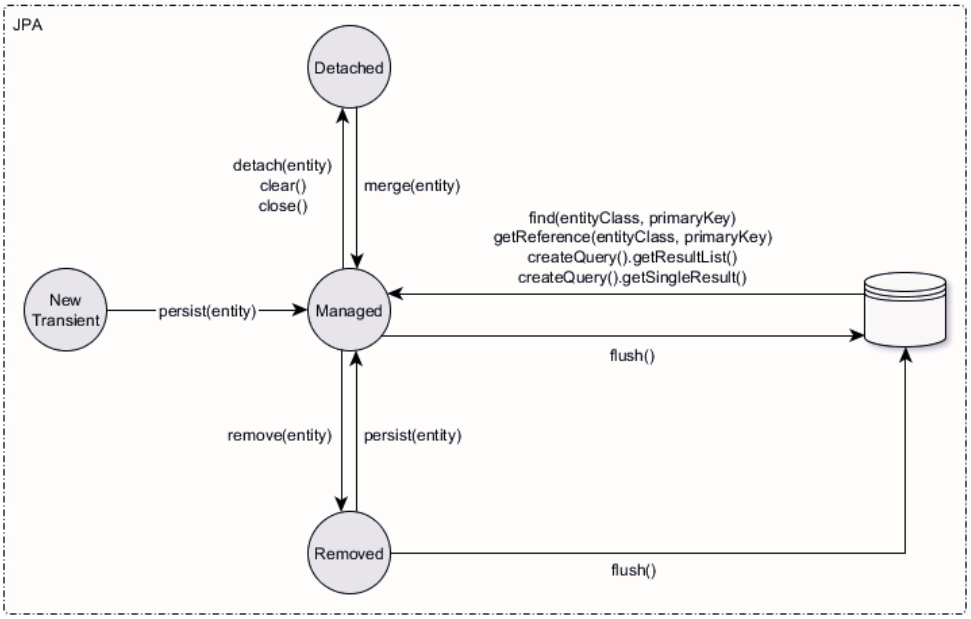


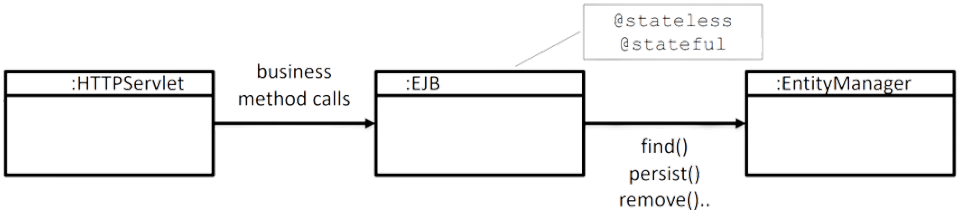
Figure 4.7: Possible states for the entities

According to the previous graph we have that an entity can be in five main states (deleted is omitted in the image):

- New: the entity is unknown to the entity manager, has no persistent identity, and has no tuple associated.
- Managed: the entity is associated with persistence context, changes to objects automatically, and synchronizes to database.
- Detached: the entity has an identity potentially associated with a database tuple, but changes are not automatically propagated to the database.
- Removed: the entity is scheduled for removal from the database.
- Deleted: the entity is erased from the database.

Application architecture

In Java Enterprise Edition the client exploits the services the EJB container to connect to the entity manager. In particular, the business level interacts with the entity manager. The advantage of the EJB is that this container provides the support to make JPA entity method calls transactional through the automatic creation of transactions.



The transactions exist at three levels:

- DBMS transactions: they are managed by the DBMS and use SQL.
- Resource-local transactions: they are managed by the application and use the JDBC connection interface. They are mapped to the JDBC.
- Container transaction: they are managed by the application or the container and use the JTA interface. They are mapped to the JDBC.

The level used by the entity manager is the container transaction one. After defining a business object, the container injects the entity manager into it. The container manages instances of the entity manager transparently to the application. The container provides the transaction needed for saving the modifications made to the entities of the persistence context associated with the entity manager into the database.

Example : Definition of a business object EJB:

```
@Stateless
public class myEJBService {
@PersistenceContext(unitName = "MyPersistenceUnit")
private EntityManager em;
}
```

When a client calls a method of a business object that exploits a container managed entity manager for persistence, the container provides a transaction for saving the modifications to the database (if the same transaction is called multiple times it will be reused). Note that this is the most common and default behavior, but the business objects methods can be annotated to specify a different way to use the transactions provided by the container. A method can be annotated to obtain the desired transactional behaviour with @TransactionAttribute(TransactionAttributeType.type), where type can be:

- Mandatory: a transaction is expected to have already been started and be active when the method is called. If no transaction is active, an exception is thrown.
- Required: the default, if no transaction is active one is started. If one is active this is used.
- Requires_new: the method always needs to be in its own transaction. Any active transaction is suspended.
- Supports: the method does not access transactional resources, but tolerates running inside one if it exists.
- Not_supported: the method will cause the container to suspend the current transaction if one is active when the method is called.
- Never: the method will cause the container to throw an exception if a transaction is active when the method is called.

4.7 Benefits of Java Persistence API

The main benefits of using the Java Persistence API are:

- No need to write SQL code.
- Application code ignores table names, since the mapping is expressed via annotations with defaults.
- No need to create/destroy the connection.
- No need to create and terminate the transaction.
- Business objects methods are automatically made transactional.
- Default transactional behavior, with annotations to specify different transaction management policies.

CHAPTER 5

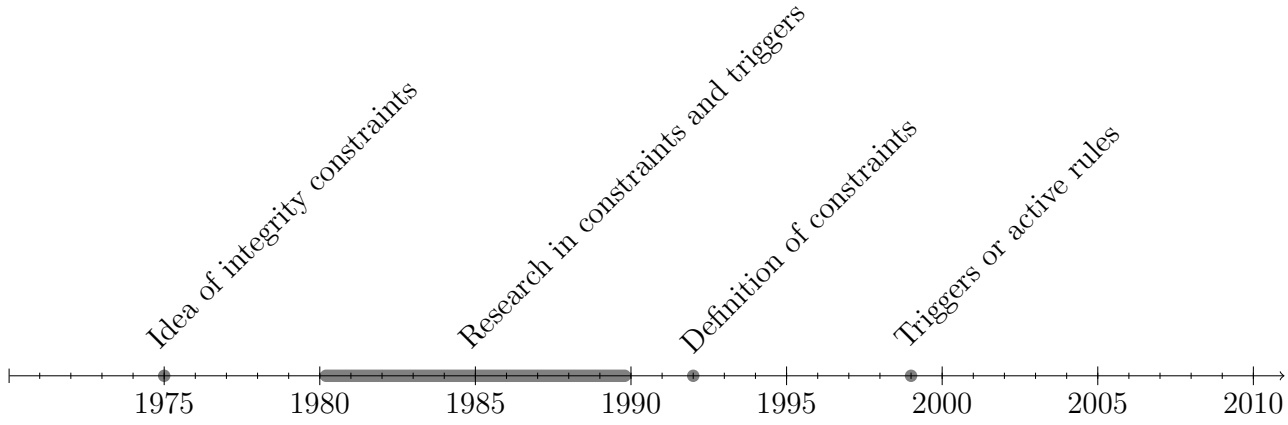
Triggers

5.1 Definition and history

Definition

A *trigger* refers to a set of actions that are automatically executed when an INSERT, UPDATE, or DELETE operation is carried out on a designated table.

Triggers are utilized to transition from a passive state to an active state, enabling automated responses to database changes.



5.2 Introduction

Triggers are built on the Event-Condition-Action paradigm, where an action A is automatically executed whenever a specific event E occurs, contingent upon the truth of a condition C. To be more specific, the components of triggers can be described as follows:

- Event: typically, this pertains to a change in the database's status, encompassing operations like insertions, deletions, and updates.
- Condition: this is a predicate that defines the specific circumstances in which the trigger's action must be executed.

- Action: the action entails a general update statement or a stored procedure, typically involving database modifications such as insertions, deletions, and updates. It may also encompass error notifications.

Triggers operate alongside integrity constraints and provide the capability to assess intricate conditions. These triggers are compiled and stored within the Database Management System much like stored procedures. However, unlike stored procedures that are invoked by the client, triggers execute automatically in response to predefined events.

5.3 Triggers definition

The typical syntax of a trigger is as follows:

```
CREATE TRIGGER <TriggerName>
{BEFORE|AFTER}
{INSERT|DELETE|UPDATE [OF <ColumnName>]} ON <TableName>
REFERENCING {
              [OLD TABLE [AS] <OldTableAlias>]
              [NEW TABLE [AS] <NewTableAlias>] |
              [OLD [ROW] [AS] <OldTupleName>]
              [NEW [ROW] [AS] <NewTupleName>]
}
[FOR EACH {ROW|STATEMENT}]
[WHEN <Condition>]
<SQLProceduralStatement>
```

Triggers can be executed in different modes, including:

- BEFORE: in this mode, the trigger's action occurs before any changes to the database's status, but only if the specified condition is met. It is primarily used for validating modifications before they are applied, potentially allowing for conditional effects. A significant constraint is that "before" triggers cannot directly update the database. However, they can impact transition variables at a row-level granularity, often implemented as:

  ```
  SET NEW.t = <expression>
  ```

- AFTER: this mode involves executing the trigger's action after the database has undergone modifications, contingent upon the satisfaction of the specified condition. It is the most commonly used mode and is well-suited for a wide range of applications.

Triggers can function at various levels of granularity:

- Row-level granularity: tn this approach, the trigger is evaluated and potentially executed separately for each tuple affected by the triggering statement. Writing triggers at the row-level is more straightforward but may be less efficient.

- Statement-level granularity: triggers are assessed and potentially executed only once for each activating statement, regardless of the number of tuples impacted in the target table. This occurs even if no tuple is affected. This mode aligns more closely with the traditional SQL statement approach, which is typically set-oriented.

Definition

In the context of triggers, *transition variables* are special variables that indicate the state of a modification, both before and after, and their syntax depends on the chosen granularity:

- Row-level: in this case, the variables old and new are used, where old represents the value before the modification of the specific row (tuple) being considered, and new represents the value after the modification.
- Statement-level: for this granularity, table variables are employed. These include old table and new table, with old table containing the old values of all affected rows and new table containing the new values of those rows.

t's important to note that, in specific cases, these variables may be undefined. In triggers with an event of "insert", the variables old and old table are undefined, while in triggers with an event of "delete", the variables new and new table are undefined.

Example : Table T2 serves as a replica of table T1. Whenever an update is made to T1, this modification is automatically replicated in T2. This replication process is accomplished using triggers. The insertion of a new tuple is done with the following trigger:

```
CREATE TRIGGER REPLIC_INS
AFTER INSERT ON T1
FOR EACH ROW
INSERT INTO T2 VALUES (NEW.ID, NEW.VALUE);
```

The deletion of a tuple is done with the following trigger:

```
CREATE TRIGGER REPLIC_DEL
AFTER DELETE ON T1
FOR EACH ROW
DELETE FROM T2 WHERE T2.ID = OLD.ID;
```

The update of only the value a tuple is done with the following trigger:

```
CREATE TRIGGER REPLIC_UPD
AFTER UPDATE OF VALUE ON T1
WHEN NEW.ID = OLD.ID
FOR EACH ROW
UPDATE T2 SET T2.VALUE = NEW.VALUE
WHERE T2.ID = OLD.ID;
```

The previously mentioned trigger will not activate when the ID of a tuple is modified. Nonetheless, for a comprehensive and robust implementation, this scenario must also be taken into account.

If we want to replicate the rows only for the tuples whose value is ≥ 10, we will havwe the following triggers:

```
-- Insertion
CREATE TRIGGER CON_REPL_INS
AFTER INSERT ON T1
FOR EACH ROW
WHEN (new.VALUE >= 10)
INSERT INTO T2 VALUES (new.ID, new.VALUE);
-- Deletion
```

```
CREATE TRIGGER Cond_REPL_DEL
AFTER DELETE ON T1
FOR EACH ROW
WHEN (old.VALUE >= 10)
DELETE FROM T2 WHERE T2.ID = old.ID;
```

When multiple triggers are linked to a common event, SQL:1999 dictates the sequence in which they execute:

1. "before" statement level triggers.
2. "before" row level triggers.
3. Modification is applied, and integrity constraints are checked.
4. "after" row level triggers.
5. "after" statement level triggers.

However, if there are multiple triggers within the same category, the order of execution is contingent upon the specific system's implementation.

Definition

The action of a trigger can lead to the activation of another trigger, a phenomenon known as *cascading* or nesting.

Recursive cascading occurs when a statement, S, executed on table T initiates a sequence of triggers that generates the same event, S, on table T, often referred to as looping. It is important to ensure that recursive cascading does not. produce undesired effects.

Definition

In the context of triggers, the concept of *termination* ensures that, regardless of the initial state and any sequence of modifications, there is always a definitive final state, thereby preventing the possibility of infinite activation cycles.

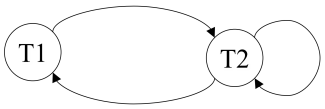
The simplest check exploits the triggering graph, that consists of a node i for each trigger T_i, and an arc from a node i to a node j if the execution of trigger T_i's action may activate trigger T_j. The graph is created through a straightforward syntactic analysis. If it is acyclic, the system will definitely terminate. However, in the presence of cycles, triggers may or may not terminate. It's important to emphasize that acyclicity is adequate for ensuring termination but not a mandatory requirement.

Example : Consider the following triggers:

```
-- T1
CREATE TRIGGER AdjustContributions
AFTER UPDATE OF Salary ON Employee
REFERENCING NEW TABLE AS NewEmp
FOR EACH STATEMENT
UPDATE Employee
SET Contribution = Salary * 0.8
WHERE RegNum IN (SELECT RegNum FROM NewEmp)
-- T2
```

```
CREATE TRIGGER CheckOverallBudgetThreshold
AFTER UPDATE ON Employee
WHEN (SELECT sum(Salary+Contribution) FROM Employee) > 50000
UPDATE Employee
SET Salary = 0.9 * Salary;
```

The triggering graph associated with these triggers is illustrated below:



In this triggering graph, there are two cycles, but it's important to note that the system still terminates.

5.4 Triggers application

Triggers introduce robust data management capabilities in a seamless and reusable fashion. This system empowers databases to incorporate business and management rules that would otherwise be scattered across various applications. Nevertheless, comprehending the interplay between triggers can be intricate, as certain database management system providers leverage triggers to execute internal functions.

View materialization

A view is a virtual table defined through a query stored in the database catalog and subsequently utilized in queries as if it were a conventional table. When a view is referenced in a "select" query, the query processor revises the query by employing the view definition, ensuring that the executed query solely involves the base tables linked to the view. If the queries involving a view significantly outnumber the updates to the base tables that alter the view's contents, view materialization can be considered. Certain systems offer the "create materialized view" command, which enables the DBMS to automatically materialize the view. Alternatively, materialization can be implemented through the use of triggers.

Design principles

The design principles encompass the following guidelines:

1. Employ triggers to ensure that specific operations trigger related actions.
2. Avoid defining triggers that replicate functionality already inherent in the DBMS.
3. Keep trigger code concise. If your trigger's logic extends beyond 60 lines of code, consider placing the majority of the code within a stored procedure and invoke the procedure from the trigger.
4. Utilize triggers exclusively for centralized, global operations intended to be executed for the triggering statement, regardless of the issuing user or database application.

5. Minimize the use of recursive triggers unless absolutely necessary, as triggers may inadvertently trigger one another until the DBMS exhausts its memory.

6. Exercise caution when implementing triggers, as they are executed for every user each time the relevant trigger event occurs.

Summary

All prominent relational DBMS vendors offer varying degrees of support for triggers. However, it's important to note that most products provide support for only a subset of the SQL-99 trigger standard, and they may not fully adhere to some of the more intricate aspects of the execution model. Additionally, certain trigger implementations rely on proprietary programming languages, making portability across different DBMS platforms a challenging endeavor.

It's crucial to emphasize that the central management of semantics within the database, under the control of the DBMS and not replicated across all applications, is imperative. This ensures the enforcement of data properties that cannot be explicitly specified through integrity constraints. As triggers often operate in the background, their behavior should always be well-documented, as it tends to be somewhat concealed from users and developers

CHAPTER 6

Physical databases