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

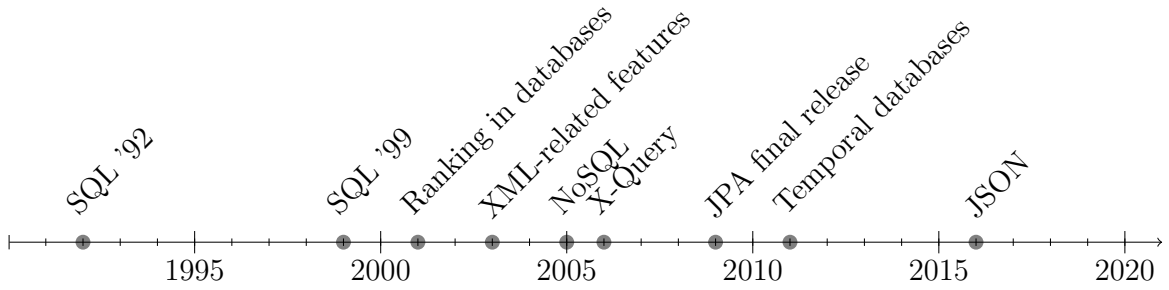
Introduction

1.1 Data Base Management System

Definition (*DataBase Management System*)**.** A DBMS 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.

Evolution The key developments in DBMS innovation are highlighted in the chronological timeline below:



Architecture The architecture of a DBMS is depicted in the figure below:

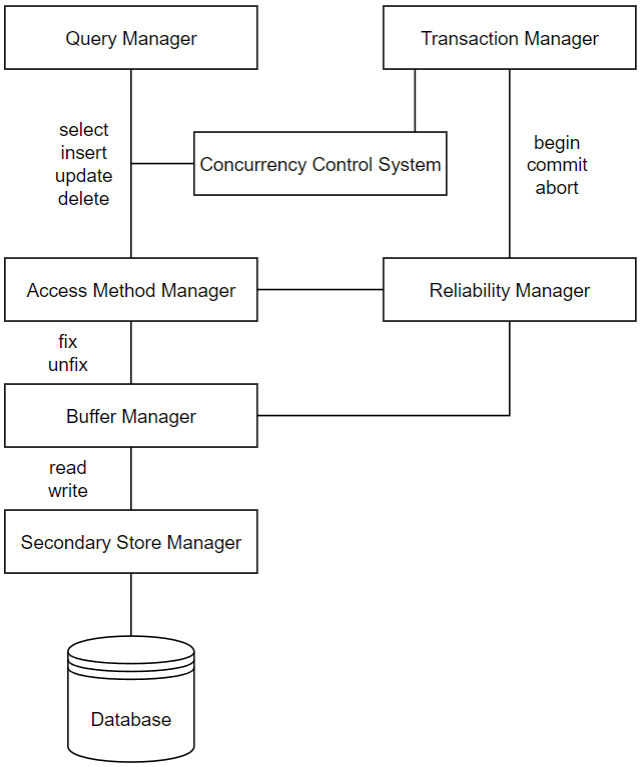


Figure 1.1: DBMS architecture

1.2 Transactions

Definition (*Transaction*)**.** A transaction is an elementary, atomic unit of work performed by an application.

Each transaction is conceptually encapsulated within two commands: `begin` and `end`. In the context of a transaction, the conclusion of the process is indicated by executing one of the following commands `commit` or `rollback`.

Definition (*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.

ACID properties 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.

Each of these properties is assured by corresponding components of the displayed DBMS. Specifically, we have that:

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

CHAPTER 2

Concurrency

2.1 Introduction

A DBMS typically manage 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 updates*: update applied from a state that ignores a preceding update, resulting in the loss of the earlier update.
- *Dirty reads*: an uncommitted value is used to update the data.
- *Non-repeatable reads*: another transaction updates a previously read value.
- *Phantom updates*: another transaction updates data that contributes to a previously valid constraint.
- *Phantom inserts*: another transaction inserts data that contributes to a previously read datum.

2.3 Concurrency theory

Definition (*Model*)**.** 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 (*Operation*)**.** An operation consist in a reading or in a writing of a specific datum by a specific transaction.

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

Schedule Transactions can be categorized as serial, interleaved, or nested. Let N_S and N_D be respectively the number of serial schedules and distinct schedules for n transactions $\langle T_1, \ldots, T_n \rangle$ each with k_i operations. Then:

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

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

1. $r_1(x)w_1(x)r_2(z)w_2(z) \rightarrow$ serial
2. $r_2(z)w_2(z)r_1(x)w_1(x) \rightarrow$ serial
3. $r_1(x)r_2(z)w_1(x)w_2(z) \rightarrow$ nested
4. $r_2(z)r_1(x)w_2(z)w_1(x) \rightarrow$ nested
5. $r_1(x)r_2(z)w_2(z)w_1(x) \rightarrow$ interleaved
6. $r_2(z)r_1(x)w_1(x)w_2(z) \rightarrow$ interleaved

Concurrency control must reject all schedules that lead to anomalies.

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

Definition (*Serial schedule*)**.** 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 (*Read from*)**.** The operation $r_i(x)$ 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 $w_j(x)$.

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

Definition (*View equivalence*)**.** Two schedules are said to be view-equivalent ($S_i \approx_V S_j$) if they have the same: operations, reads-from relationships and final writes.

Definition (*View serializable*)**.** 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 (*Conflict*)**.** 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: read-write conflicts or write-write conflicts.

Definition (*Conflict equivalent*)**.** 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.

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

Property 2.1. VSR is a strict subset of CSR.

Proof. Consider the schedule $S = r_1(x)w_2(x)w_1(x)w_3(x)$, which is VSR but not CSR. It can be verified that there is no conflict-equivalent serial schedule. □

Property 2.2. CSR implies VSR.

Proof. 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 CSR), and 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). □

Conflict graph 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 2.1. *A schedule is conflict-serializable if and only if its conflict graph is acyclic.*

Example:
Consider the schedule

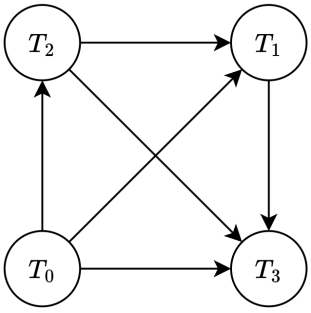
$$S : w_0(x)r_1(x)w_0(z)r_1(z)r_2(x)w_0(y)r_3(z)w_3(z)w_2(y)w_1(x)w_3(y)$$

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:

- $x : w_0r_1r_2w_1$
- $y : w_0w_2w_3$
- $z : w_0r_1r_3w_3$



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

Proof. We prove that CSR implies acyclicity. 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. We prove that acyclicity 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.

Online concurrency control 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*): 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).

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

Lock tables 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.

2.7.1 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 (*Two-phase locking*)**.** 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, and checks that all transactions apply the two-phase rule.

Property 2.3. 2PL is a strict subset of CSR.

Proof. 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 $S^{'}$:

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

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. □

Property 2.4. 2PL implies CSR.

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

2.7.2 Strict two-phase locking

Definition (*Strict two-phase locking*)**.** 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 Data Base Management Systems when a high level of isolation is necessary.

Predicate locking 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.

2.8 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.

2.9 Deadlocks

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

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

Definition (*Wait-for graph*)**.** 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.

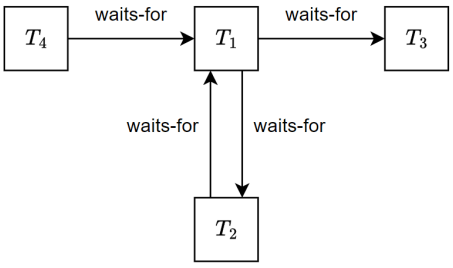


Figure 2.1: An example of deadlock in the wait-for graph

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.

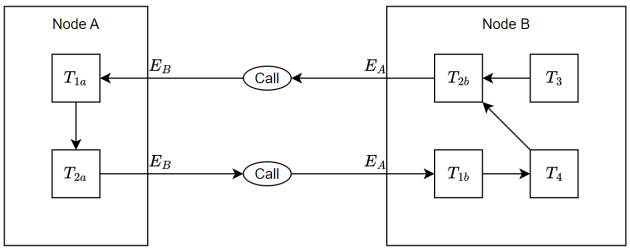
Dependency graph The distributed dependency graph is a wait-for graph where external call nodes represent a sub-transaction activating another sub-transaction at a different node. The arrow shows a wait-for relation among local transactions. If one term is an external call, either the source is waited for by a remote transaction or waits for a remote transaction.

Obermarck's algorithm 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:



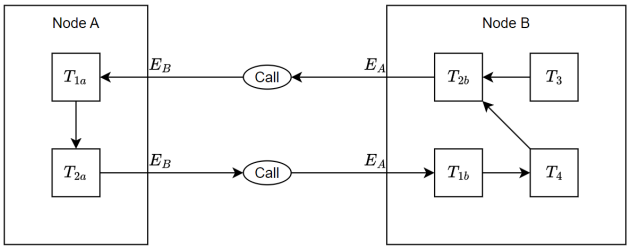
In this graph, a potential deadlock is indicated by cycles. Specifically, we observe that T_{2a} waits for T_{1a} (data lock), which, in turn, waits for T_{1b} (call), leading to a sequence where T_{2b} (data locks) waits for T_{2a} (call).

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

Practical application 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:

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

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

2.10 Timestamps

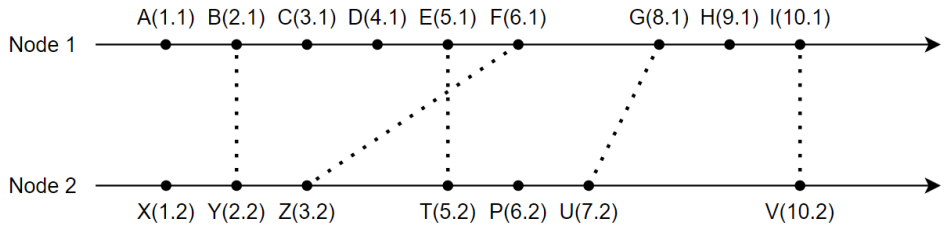
Locking assumes collisions will occur, although in reality, collisions are infrequent. Optimistic concurrency control methods like timestamps can be employed to address this. Timestamps are identifiers that establish a total ordering of a system's events. Each transaction is assigned a timestamp representing 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:

$$\langle \text{event-id.node-id} \rangle$$

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.



Counters 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 $<$ WTM(x), the request is rejected, and the transaction is killed.
- Otherwise, access is granted, and RTM(x) = max(RTM(x), ts).

For write operations:

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

However, these rules may lead to excessive transaction killings.

Example:
Assuming RTM(x) = 7 and 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**
$r_6(x)$	✓	-
$r_8(x)$	✓	RTM(x) = 8
$r_9(x)$	✓	RTM(x) = 9
$w_8(x)$	✗	T_8 killed
$w_{11}(x)$	✓	WTM(x) = 11
$r_{10}(x)$	✗	T_{10} killed

Comparing 2PL to is challenging, and there is no subset relationship between the two. However, TS implies CSR.

Proof. 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 we have two possibilities. 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. □

Transaction abort 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 $r_{\text{ts}}(x)$ or $w_{\text{ts}}(x)$ such that ts $>$ 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.

Thomas rule To reduce the number of killings, the Thomas rule can be used, altering the rule for write operations:

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

2.10.1 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 (WTM(x)), ensuring that each object x always has $N \geq 1$ active versions. A unique global read timestamp (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:

- $r_{\text{ts}}(x)$ is always accepted. A copy x_k is selected for reading, where:
  - If ts $\geq$ WTM$_N(x)$, then $k = N$.
  - Otherwise, k is chosen such that WTM$_k(x) \leq$ ts $<$ WTM$_{k+1}(x)$.
- $w_{\text{ts}}(x)$:
  - If ts $<$ 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 $x_{(2)}$	$\text{RTM}(x) = 12$
$w_{14}(x)$	✓	$\text{WTM}_3(x) = 14,\ N = 3$
$w_{13}(x)$	✓	$\text{WTM}_4(x) = 14,\ N = 4$

Multiversioning in practice In practice, the rule set is modified slightly:

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

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

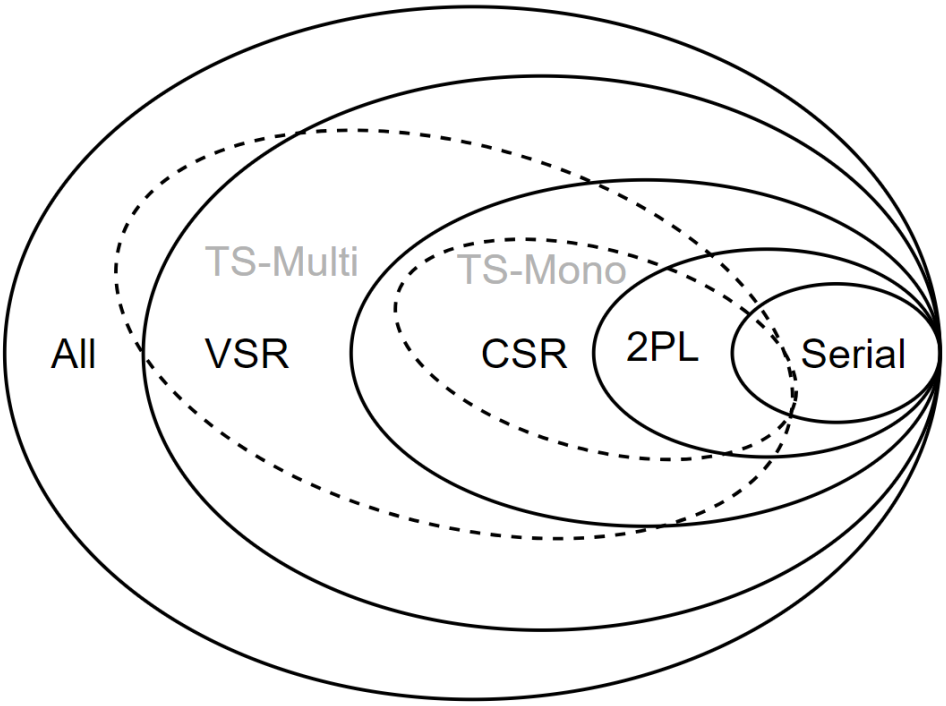
Using multi-versioning, the results are:

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 levels 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.

2.11 Concurrency classes sets

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



CHAPTER 3

Ranking

3.1 Introduction

Achieving optimal results from data can be accomplished through the application of ranking, involving the simultaneous optimization of various criteria. A typical formulation for a multi-objective problem is as follows: with N objects characterized by d attributes, the objective is to identify the top k objects.

Methods The primary methods for addressing multi-objective optimization include:

- *Ranking queries*: this approach involves selecting the top k objects based on a specified scoring function.
- *Skyline queries*: this method focuses on choosing the set of non-dominated objects, ensuring they are not inferior in any criterion.

3.2 History

In the 13th century, the problem of elections and polls was introduced. Borda proposed a solution involving assigning penalty points based on the position of candidates in a voter's ranking, aiming for the candidate with the lowest overall penalty to win. On the other hand, Condorcet suggested determining the winner as the candidate who defeats every other candidate in pairwise majority rule elections.

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)

However, for Condorcet, A wins in pairwise majority. The winner depends on the method used.

In 1950, Arrow proposed the axiomatic approach, defining aggregation as axioms. He formulated Arrow's paradox, stating that no rank-order electoral system can satisfy the following fairness criteria simultaneously:

- 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 address this paradox, researchers introduced the metric approach. This involves finding a new ranking R minimizing the total distance to the initial rankings $R_1, \ldots, R_n$. Distances between rankings can be measured using methods such as:

- 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 ($\mathcal{NP}$-complete), but tractable for Spearman's foot-rule ($\mathcal{P}$ time). These distances are related as follows:

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

Efficient approximations for $F(R_1, R_2)$ are also possible.

3.3 Opaque rankings

The concept of opaque rankings focuses only on positions without considering associated scores.

MedRank algorithm The MedRank algorithm, inspired by the median, offers an approximation of the foot-rule optimal aggregation. The algorithm takes an integer k and a ranked list $R_1, \ldots, R_m$ of N elements as inputs, producing the top k elements based on the median ranking. The algorithm proceeds as follows:

1. Use sorted accesses in each list, extracting one element at a time until there are k elements occurring in more than $\frac{m}{2}$ lists.
2. Identify these as the top k elements.

Definition (*Depth*)**.** The maximum number of sorted accesses made on each list is called the depth reached by the algorithm.

Example:
Consider sorting the following hotels based on three criteria using the MedRank algorithm. The hotels' reviews are as follows:

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
...	...	...

Using MedRank with $k = 3$, the resulting rank is:

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

Definition (*Istance optimal*)**.** An algorithm is instance-optimal if is the best possible algorithm on every input of a specific instance.

Property 3.1. MedRank is instace-optimal on ordered lists.

Theorem 3.1 (Instance optimality)**.** *Let A be a family of algorithms, 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 A and I for the cost metric cost if there exist constants k_1 and k_2 such that, for all $A \in A$ and $I \in I$:*

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

If A^* is instance-optimal, then any algorithm can improve with respect to A^* by only a constant factor r, known as 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 known as top-k queries, focus on retrieving the k best answers from a potentially extensive result set based on the concept of relevance.

Algorithm The algorithm takes as inputs the cardinality k, the dataset R, and the scoring function S, with the output being the top k tuples with the highest scores according to S. The algorithm's approach involves computing scores for all tuples, sorting them based on these scores, and finally returning the top k tuples.

However, this straightforward method becomes expensive for large datasets, especially when multiple relations are involved, necessitating the sorting of significant amounts of data. Two crucial abilities are required:

- Ordering the tuples according to their scores with `ORDER BY`.
- Limiting the output cardinality to k tuples with `FETCH FIRST k ROWS ONLY`.

Example:
Consider the following queries:

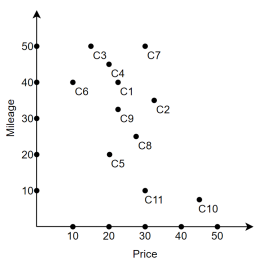
```
-- Query one
SELECT *
FROM UsedCarsTable
WHERE Vehicle = 'Audi/A4'
AND Price <= 21000
ORDER BY 0.8*Price+0.2*Miles
-- Query two
SELECT *
FROM UsedCarsTable
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price+0.2*Miles
```

Here, the values 0.8 and 0.2, termed weights, normalize preferences on price and mileage. The first query might miss relevant answers (near-miss), could result in information overload by returning all Audi A4 cars in the dataset.

Only the initial k tuples are included in the result. If multiple sets of k tuples meet the ordering directive, any of these sets is considered a valid answer, demonstrating non-deterministic semantics.

Evaluation The evaluation of a top-k query considers two basic aspects: query type (one relation, many relations, and aggregate results) and access paths (no index, indexes on all/some ranking attributes). In the simplest case, in a top-k selection query with a single relation, if the input is sorted according to S, only the first k tuples need to be read. If the tuples are not sorted, an in-memory sort is performed with a complexity of $O(N \log k)$.

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



In this representation, each tuple corresponds to a two-dimensional point (p, m), where p denotes the Price, and m represents the Mileage. The intuition behind minimizing $0.8 \cdot \text{Price} + 0.2 \cdot \text{Mileage}$ is the same as searching for points in proximity to $(0, 0)$. Our preferences play a crucial role in determining the outcome. Consider the equation of the line:

$$0.8 \cdot \text{Price} + 0.2 \cdot \text{Mileage} = v$$

Here, v is a constant. This equation can be alternatively expressed as:

$$\text{Mileage} = -4 \cdot \text{Price} + 5 \cdot v$$

From the previous equation we see that all the lines have a slope -4. By definition, all points on the same line are equally good. Modifying the weights may lead to variations in the optimal choice. Importantly, the target of a query is not confined to $(0, 0)$; it can be any point $q = (q_1, q_2)$.

Distance functions In a general context, to assess the quality of a tuple t, we calculate its distance from the target point q. The desirability increases as the distance from q decreases. When examining the distances, the model can be characterized by the following components:

- An m-dimensional space $A = (A_1, A_2, \dots)$ comprising ranking attributes, where $m \geq 1$.
- A relation $R(A_1, A_2, \dots, B_1, B_2, \dots)$, with $B_1, B_2, \dots$ representing additional attributes.
- A target point $q = (q_1, q_2, \dots)$ located in A.
- A distance function $d : A \times A \to \mathbb{R}^+$, quantifying the distance between points in A.

Definition (*Nearest neighbors*)**.** 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.

Definition (*L_p-norms*)**.** The distance function, denoted as L_p-norms, is defined by the formula:

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

Significant instances of the L_p-norm include:

- *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):

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

Top-k join query In a top-k join query with $n > 1$ input relations and a scoring function S defined on the result of the join, the general formulation is as follows:

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

In this query, $p_1, p_2, \ldots, p_m$ represent the scoring criteria. In top-k $1-1$ join queries, where all joins are on a common key attribute, two main scenarios are possible:

- 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, returning the identifier of the next best object and its partial score p_j.
- Each input list supports random access, returning 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 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
```

The result would be:

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

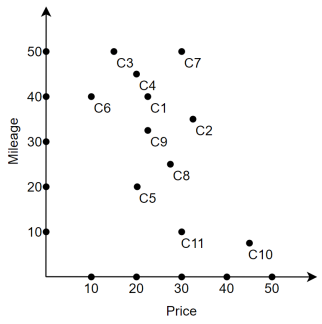
Note that the winner is not the best locally.

Score space Each object o returned by the input L_j has an associated local/partial score $p_j(o) \in [0,1]$. Scores are normalized, and 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 using a scoring function S that combines 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 $A = (\text{Price}, \text{Mileage})$

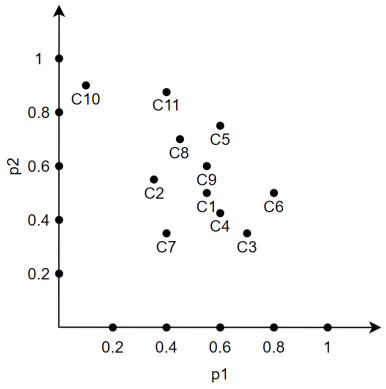


If we choose MaxP $= 50,000$ and MaxM $= 80,000$, a generic point o is mapped with these formulas:

$$p_1(o) = 1 - \frac{o.\text{Price}}{\text{MaxP}}$$

$$p_2(o) = 1 - \frac{o.\text{Mileage}}{\text{MaxM}}$$

The new graph is as follows:



Scoring functions The most common scoring functions are:

- Average, that weighs preferences equally:

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

- Weighted sum, 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, that considers the worst partial score:

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

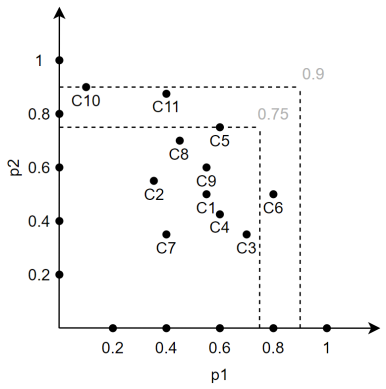
- Maximum, that considers the best partial score:

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

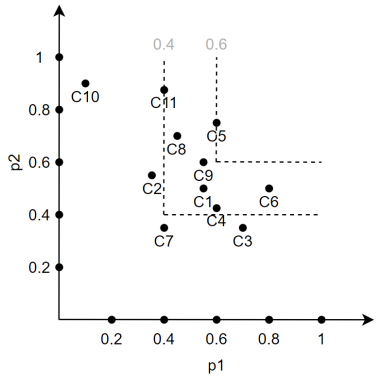
Even with the minimum scoring function, the goal is to retrieve the k objects with the highest global scores.

Iso-score curves Iso-score curves in the score space can be defined, representing sets 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:



3.4.1 B-zero algorithm

Efficient computation of the result of a top-k 1-1 join query using a maximum scoring function S can be achieved through the B_0 algorithm. The inputs for this algorithm are an integer $k \geq 1$ and a ranked list $R_1, \ldots, R_m$. The algorithm proceeds as follows:

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

Property 3.2. B_0 algorithm is instance-optimal.

Example:
For instance, given 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 apply the B_0 algorithm with $k = 3$, selecting the first three rows from both tables (which are sorted), and summing all the values by identifier, we obtain the final rank with the maximum value:

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

3.4.2 A-zero algorithm

The A-zero algorithm, also known as Fagin's algorithm, is applicable to every monotonic scoring function. The algorithm takes an integer k and a monotone function S combining ranked lists $R_1, \ldots, R_m$ as inputs, producing the top k (object-score) tuples as output. The algorithm follows these steps:

1. Extract the same number of objects through sorted accesses in each list until there are at least k common objects.
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 $O(N^{\frac{m-1}{m}} k^{\frac{1}{m}})$, which is sublinear in the number N of objects. The stopping criterion is function-independent.

Property 3.3. The algorithm is not instance-optimal.

Example:
Consider hotel reviews 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
...	...

Using the scoring function $0.5 \cdot \text{cheapness} + 0.5 \cdot \text{rating}$ and $k = 2$, iterate on the rows until k hotels are found in all columns (in this case, Ibis, Novotel, and Hilton). Complete the score of the remaining incomplete hotels by making random accesses to find the missing values. After summing up the information, compute the final ranking with the given score function, resulting in:

Top k	Score
Novotel	0.875
Crillon	0.825

Drawbacks The primary limitation of this algorithm lies in its dependence on accesses, leading to an underutilization of the specificity inherent in certain scoring functions. Additionally, the memory requirements have the potential to become impractical. While there are possibilities for improvements, substantial enhancements in complexity can only be achieved by modifying the stopping condition.

3.4.3 Threshold algorithm

The threshold algorithm takes as inputs an integer k and a monotone function S that combines ranked lists $R_1, \ldots, R_m$. It outputs the top k (object, score) tuples. The algorithm follows these steps:

1. Perform sorted access in parallel in each list R_i.
2. For each object o, conduct random accesses in the other lists R_j to extract score s_j.
3. Compute the 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 the 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 stopping criterion depends on the function and not on the number of accesses.

Property 3.4. The threshold algorithm is instance optimal among all algorithms that use random and sorted accesses.

Example:
Consider hotel reviews with cheapness and rating values.

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
...	...

The scoring function is $0.5 \cdot \text{cheapness} + 0.5 \cdot \text{rating}$, 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. The buffer contains the top k hotels with their corresponding scores.

Top k	Score
Novotel	0.875
Crillon	0.825

The threshold point has a value of 0.825.

Performance evaluation In general, the Threshold Algorithm outperforms the Fagin Algorithm since it can adapt to the specific scoring function. To assess the performance of Threshold Algorithm, we evaluate the middleware cost using the following formula:

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

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.

3.4.4 No Random Access algorithm

The No Random Access algorithm returns the top-k objects, but their scores might be uncertain. The fundamental idea behind this algorithm is to maintain, for each selected object o, both a lower bound and an upper bound on its score. Thus, an unlimited buffer is required to store the objects, sorted by decreasing lower bound values. The algorithm stops when the upper bound of a new object is less than or equal to the worst lower bound among the best objects. The inputs include 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 scores of the objects. The procedure is as follows:

1. Perform a sorted access to each list.
2. Store each retrieved object o in buffer B, maintaining $S^-(o)$ and $S^+(o)$ along with 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 tables based on three different criteria:

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.

Property 3.5. No Random Access algorithm is instance optimal among all algorithms that do not make random accesses.

3.4.5 Summary

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

The main aspects of the ranking queries are:

- Effectiveness in pinpointing the top objects based on a designated scoring function.
- Exceptional control over the result's cardinality.
- High computational efficiency.
- Ease of expressing the relative importance of attributes.
- Challenges for users in specifying a scoring function.

3.5 Skyline queries

Skyline queries aim to identify superior objects across multiple perspectives, relying on the concept of dominance.

Definition (*Domination*)**.** 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 is better at least once:

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

Definition (*Skyline*)**.** 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. The convention is that lower values are considered better. A tuple is part of the skyline if it is the top-1 result with respect to at least one monotone scoring function.

Query syntax The syntax for skyline queries can be expressed as follows:

```
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.

3.5.1 Block Nested Loop algorithm

The Block Nested Loop algorithm takes a dataset D of multidimensional points as input and outputs the skyline of D. Its complexity is $O(n^2)$, making it inefficient for large datasets.

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
```

3.5.2 Sort Filter Skyline algorithm

The Sort Filter Skyline algorithm also takes a dataset D of multidimensional points as input and outputs the skyline of D. It performs better than the Block Nested Loop algorithm, especially for large datasets, but still has a complexity of $O(n^2)$.

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
```

Example:
Consider 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

The skyline, consisting of Novotel and Ibis, is identified by the algorithm.

3.5.3 Summary

Skyline queries are effective in identifying potentially interesting objects when user preferences are unknown. They are simple to use but return a large number of objects, and their computation is not highly efficient. Skyline queries are agnostic with respect to user preferences.

K-skyband To enhance skyline queries, the concept of k-skyband is introduced, where the result includes a set of tuples dominated by fewer than k tuples.

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

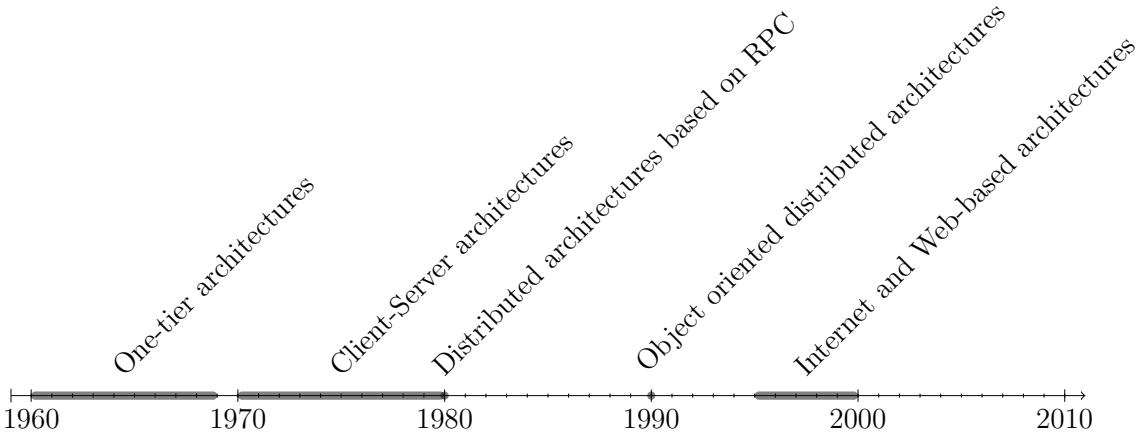
Architectures and JPA

4.1 Introduction

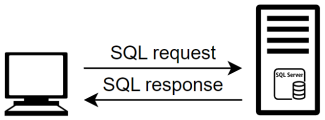
Definition (*Architecture*)**.** The architecture is the union of hardware, software, and network resources.

Nowadays, it encompasses:

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



4.2 Client-server architecture



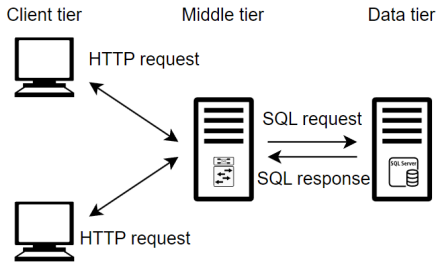
Hardware In the client-server architecture, the hardware components consist of:

- A server responsible for data management.
- A client handling presentation layout.

Software The client software dispatches requests to the server through SQL queries and encompasses both business and presentation logic. The server software is tasked with processing queries and responding by transmitting result sets back to the client and This software is dedicated to data management.

Network The network topology is structured as a local area network comprising one or more servers and multiple clients.

4.3 Three-tier architecture



The three-tier architecture employs the following hardware components:

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

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

Web pure HTML In the pure HTML architecture, the client functions as a standard web browser, focusing solely on the presentation layout (thin client). The middle tier:

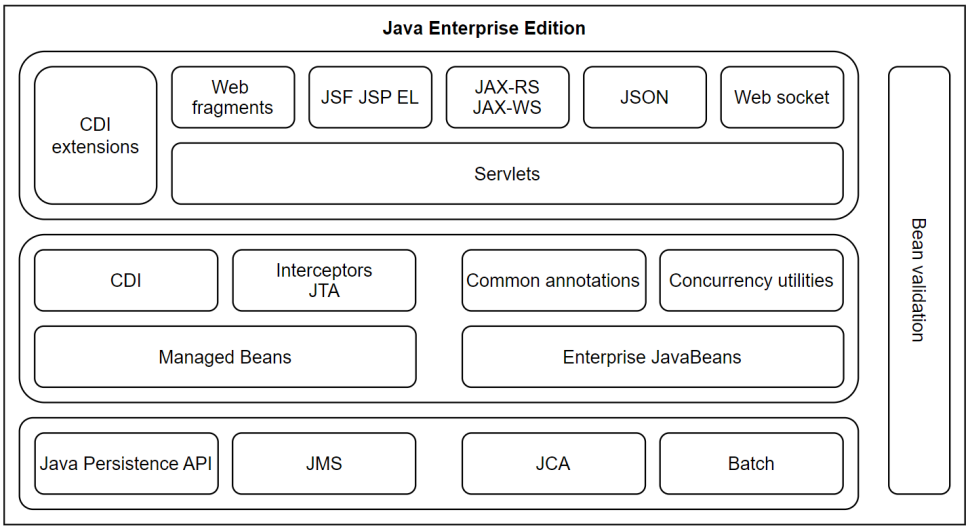
- Incorporates a Web server utilizing HTTP.
- Hosts the business logic for dynamically generating content from the raw data of the data tier.
- Manages the presentation layout.

Rich internet application The rich internet application architecture is the fusion of web and desktop applications (with JavaScript). In this scenario, the client is termed fat due to its standard communication protocol (HTTP, Web Socket), language (ECMAScript), and API (DOM, HTML5). Noteworthy features of this architecture include:

- Novel interface event types, including those specific to touch and mobile apps.
- Asynchronous interaction (AJAX).
- Client-side persistent data.
- Offline application capabilities.
- Native multimedia and 3D support.

4.4 Java enterprise edition

Java Enterprise Edition (Java EE) serves as a platform designed for the creation, deployment, and maintenance of three-tier web applications.



Java Database Connectivity The JDBC API stands as the initial industry standard for establishing database-independent connectivity between Java and databases. Through JDBC, one can establish connections with databases, send SQL statements, and process the results.

Servlets Servlets are Java technologies employed in the presentation tier of web applications. Servlets offer a platform-independent, component-based approach for constructing web-based applications. They have access to other Java APIs for interfacing with enterprise databases and operate within a container for concurrency control and lifecycle management.

Jakarta Enterprise Beans EJB technology, utilized on the server side, enables the development of distributed, transactional, secure, and portable applications. EJB components facilitate interaction between the web front-end and business functions, as well as data access services.

Java Persistence API JPA defines an interface for mapping relational data to objects in Java. It includes the API implementation package `javax.persistence`, the Java-compatible query language Java Persistence Query Language, and specifications for metadata defining object-relational mappings.

Java Transaction API JTA manages transactions in Java, allowing components to initiate, commit, and rollback transactions in a resource-agnostic manner. With JTA, Java components can oversee multiple resources within a single transaction using a unified 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 (*Impedance mismatch*)**.** 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, referred to as impedance mismatch.

The automatic transformation of a model into another in managed by an element called mediator. The principal distinctions between the object-oriented model and the relational one are as follows:

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 addresses the gap between object-oriented domain models and relational database systems by utilizing Plain Old Java Objects, providing a persistence model for object-relational mapping.

Features Key features of the Java Persistence API include:

- *POJO Persistence*: objects being persisted need not possess any special characteristics; 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 (*Entity*)**.** The entity is a class (Java bean) representing a collection of persistent objects mapped onto a relational table.

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

Definition (*Persistence context*)**.** 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).

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

Definition (*Entity manager*)**.** The entity manager is the interface for interacting with a persistence context.

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

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

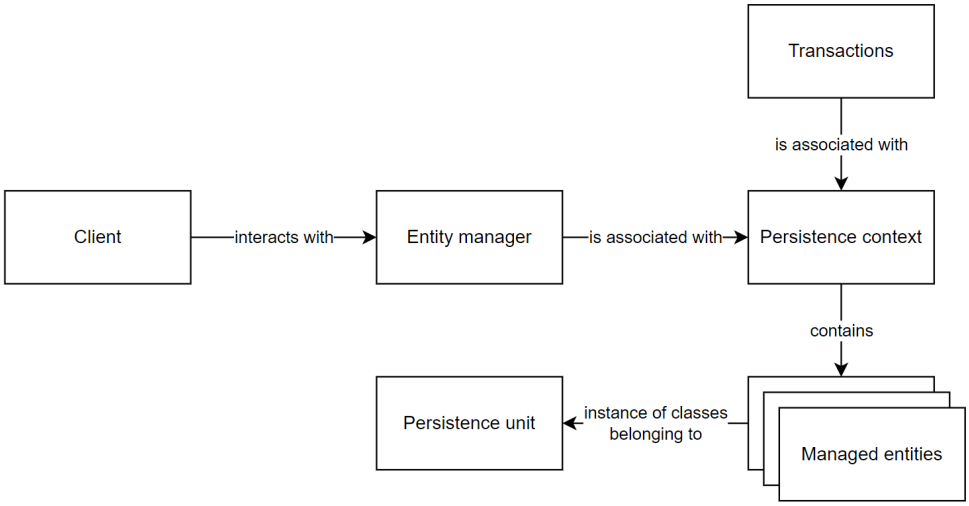


Figure 4.1: Structure of the Java Persistence API

Interfaces 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. The method's signature to accomplish this is: `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()`.

Entities An entity is a Java Bean associated with a tuple in a database. The persistent counterpart of an entity has a lifespan 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 modifications to the object's state are tracked and automatically synchronized to the database. Entities have properties such as identification (primary key), nesting, relationship, referential integrity (foreign key), and inheritance. Entities must meet certain requirements:

- Must have a public or protected constructor with no arguments.
- Must not be final.
- No method or persistent instance variables may be final.
- The 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 with. The primary key can be either simple or composite. To identify a primary key, we use the `@Id` annotation, and for composite keys, we use `@EmbeddedId` and `@IdClass` annotations.

Uniqueness of data At times, applications may prefer not to handle the explicit management of data value uniqueness. In such scenarios, the persistence provider has the capability to automatically create an identifier for each entity instance of a specified type. This functionality, known as identifier generation, is defined by the `@GeneratedValue` annotation. There are four available ID generation strategies that applications can opt for:

1. `Auto`.
2. `Table`: identifiers are generated based on a generator table.
3. `Sequence`: the identifiers are generated using sequences.
4. `Identity`: the identifiers are generated using 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;
}
```

Attributes can be qualified with properties that guide the mapping between Plain Old Java Objects (POJO) and relational tables. These properties include: large objects, enumerated types, and temporal types.

Fetch policies The fetch policy can be either lazy (retrieve the item when needed) or eager (retrieve the item as soon as possible). Lazy policy is primarily employed 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;
}
```

Entities are ordinarily mapped to tables with corresponding names, and their fields are mapped to columns with identical names by default. However, it is possible to alter this behavior using certain annotations. In cases where an entity should not be persisted, it can be designated with the `@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;
}
```

4.5.1 Mapping

In any object model, relationships exhibit four key characteristics:

- *Directionality*: each of the two entities involved may possess an attribute that facilitates access to the other entity. If two entities are mutually connected, it forms a bidirectional relationship; otherwise, it constitutes a unidirectional relationship. Multiple references are possible.
- *Role*: each entity within the relationship plays a role in one direction of access. Entities are categorized as source and target based on the relationship's direction.
- *Cardinality*: this refers to the number of instances of entities on each side of the relationship. Four cardinality possibilities exist:
  - *Many-to-one*: many source entities to one target entity.
  - *One-to-many*: one source entity to many target entities.
  - *One-to-one*: one source entity to one target entity.
  - *Many-to-many*: many source entities to many target entities.
- *Ownership*: in a relationship, one of the two entities is deemed the owner. In the database, relationships are implemented through a foreign key column (referred to as the join column in JPA) that points to the key of the referenced table. The entity with a foreign key column is considered the owner of the relationship, and its side is termed the owning side.

One-to-many relationship The bidirectional one-to-many relationship is established using the `mappedBy`, `@ManyToOne`, and `@OneToMany` annotations, where:

- `@ManyToOne` annotation is applied to the entity participating with multiple instances, designating the entity containing this annotation as the owner of the relationship.
- `@JoinColumn` annotation is utilized to specify the foreign key column of the underlying table.

Example:
Here's the first part of the definition for a one-to-many bidirectional mapping:

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

To achieve bidirectionality, the mapping direction for the one-to-many relationship must also be specified. This is accomplished by including a `@OneToMany` annotation in the entity participating with one instance. The `@OneToMany` annotation is placed on a collection data member and includes a `mappedBy` element to indicate the property implementing the inverse of the relationship.

Example:
Here's the second part of the definition for a one-to-many bidirectional mapping:

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

In cases where applications only require access to relationships along one direction, bidirectional mapping may not be 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 establish a one-to-one relationship in JPA, two alternative approaches can be chosen:

- Map the relationship similarly to the bidirectional case and utilize only the one-to-many direction.
- Omit the mapping of the collection attribute in the entity participating with one instance, and instead, use a query to retrieve correlated instances, relying on the inverse (many-to-one) relationship direction mapping.

The distinction between `@joincolumn` and `mappedby` is as following:

- The `@joincolumn` annotation specifies the foreign key column implementing the relationship in the database. This annotation is typically applied to the entity that owns the relationship and aids in generating SQL code for extracting correlated instances.
- The `mappedby` attribute indicates that this side is the inverse of the relationship, with the owner residing in the other related entity. It is used to define bidirectional relationships.

In the absence of the mappedBy parameter, the default JPA mapping involves 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 declared by annotating the owner entity with the `@OneToOne` annotation.

Example:
Here's the first part of the definition for a one-to-one mapping:

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

For bidirectional one-to-one mapping, the inverse side of the relationship must also be specified. In the non-owner entity, both the `@OneToOne` annotation and the `mappedBy` element are needed to guide JPA in placing the foreign key.

Example:
Here's the second part of the definition for 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 instead, a join table is utilized. Consequently, either entity can be arbitrarily designated as the owner.

Example:
Here's the first part of the definition for a many-to-many mapping:

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

For bidirectional many-to-many mapping, the inverse side of the relationship must also be specified. In the non-owner entity, both the `@ManyToMany` annotation and the `mappedBy` element are required.

Example:
Here's the second part of the definition for 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 necessitates a bridge table (join table in JPA).

Example:
Here's an example specifying the non-default mapping of the entity to the bridge table via annotations:

```
@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;
}
```

4.5.2 Relationship fetch mode

When the fetch mode is unspecified, the default behavior is as follows:

- Single-valued relationships are fetched eagerly.
- Collection-valued relationships are loaded lazily.

In the case of bidirectional relationships, the fetch mode may be lazy on one side but eager on the other. It's important to note that the recommended practice is to consider lazy loading as the most suitable mode for all relationships. This is because if an entity has many single-valued relationships that are not all utilized by applications, eager loading may result in performance drawbacks.

Example:
Here's an example using the annotation to define lazy loading:

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

It's essential to understand that the directive to lazily fetch an attribute serves as a hint to the persistence provider, which retains the flexibility to employ an eager loading policy if deemed necessary. However, the reverse is not true: the eager policy cannot be replaced with a lazy one by the provider.

4.5.3 Cascading operations

By default, every operation performed by the entity manager does not cascade to other entities that have a relationship with the entity being operated on. However, in certain cases, it might be desirable to propagate changes to entities in a relationship with the modified one. This can be achieved by enabling manual cascading.

Example:
Here's an example illustrating the activation of manual cascading:

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

With manual cascading activated, when the entity manager adds the Employee instance to the persistence context, it navigates the address relationship, searching 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 specify when operations should automatically cascade across relationships, accepting values such as: `persist`, `refresh`, `remove`, `merge`, and `detach`. If all the previous operations are desired, the `all` operator can be used.

Example:
Activation of manual cascading of type persist:

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

It's important to note that cascade settings are unidirectional and must be set on both sides for bidirectional behavior.

Cascade remove JPA also supports an additional remove cascading called `orphanRemoval`. This is utilized in `@OneToOne` and `@OneToMany` annotations for privately owned parent-child relationships where each child entity is associated only with one parent entity through just one relationship This operation causes the child entity to be removed when the parent-child relationship is broken by removing the parent, setting to null the attribute holding the related entity, or in the one-to-many case, by removing the child entity from the parent's collection. The distinction between the attribute `CascadeType.REMOVE` and the mode `orphanRemoval` is that if the value of an entity is manually set to null, only `orphanRemoval` will automatically remove the linked entities from the database.

4.6 JPA: entity manager

The entity instances in Java Persistence API (JPA) are plain Java objects, and they don't become managed until the application invokes an API method to initiate the process. The entity manager serves as the central authority for all persistence actions, managing the mapping and APIs for interacting with the database and the objects.

Definition (*Persistence context*)**.** The persistence context is a fundamental concept in JPA, acting as a main memory database that holds objects in a managed state.

A managed object is tracked, enabling automatic synchronization of modifications to its state with the database. Database writes typically occur asynchronously, requiring the persistence context to connect to a transaction when they happen. A managed entity has two lives: one as a Java object and one as a relational tuple bound to it, existing only within the persistence context. When the POJO exits the persistence context the binding breaks: it gets untracked and no longer synchronized to the database. The application interacts only with the entity manager, not directly with the persistence context.

Interfaces The interface of the entity manager exposes several methods:

- Makes an entity instance become part of the persistence context. The method's signature is: `public void persist(Object entity)`.
- Finds an entity instance by its primary key. The method's signature is: `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()`.

Operations on entities When an entity is initially instantiated, it is in the transient state because the entity manager is not yet aware of its existence. 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 transition transient entities to the managed state, the `persist()` method of the entity manager is used. Once an entity becomes managed, any changes made to it are automatically applied to the database, and vice versa. The managed entity and its corresponding tuple are associated until the entity exits the managed state. It is even possible to call `persist()` on an already managed entity, triggering 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);
```

Entities can be retrieved using the `find()` method, which takes the class of the sought entity and the primary key value identifying the desired entity instance. If the entity instance is found, the returned object becomes managed; otherwise, the `find()` method returns null.

Example:
Finding an entity:

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

To remove an entity, the `remove()` method is used. This action breaks the association between the entity and the persistence context. Upon the transaction associated with the entity manager's persistence context committing or when the `flush()` method of the entity manager is called, the tuple linked with the entity is scheduled for deletion from the database. Although the entity still exists, its changes are no longer tracked for synchronization with the database.

Example:
Removing an entity:

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

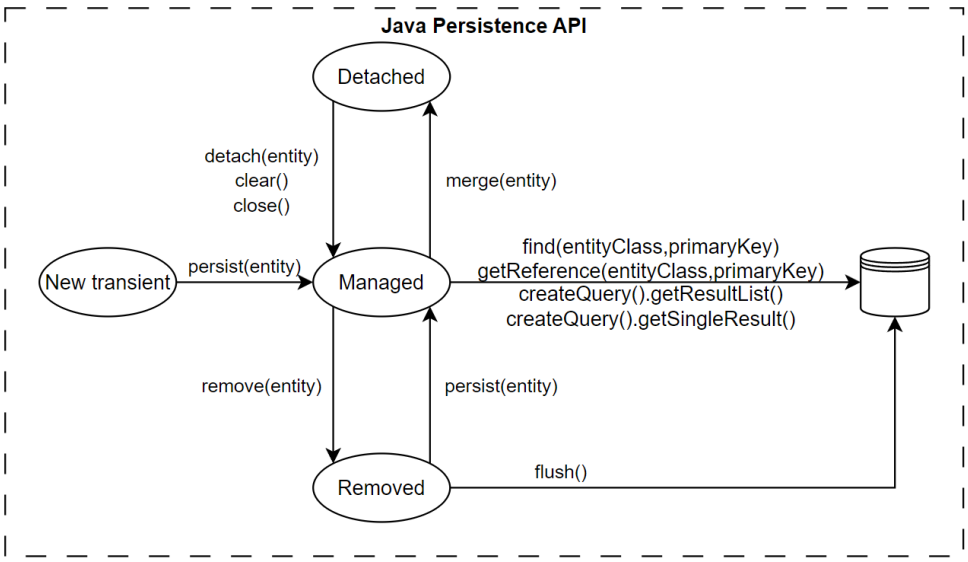


Figure 4.2: Possible states for the entities

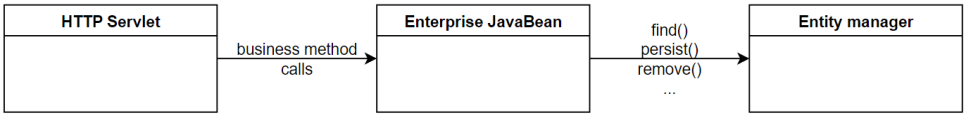
Referring to the preceding diagram, entities can exist in five primary states (deleted is excluded in the illustration):

- *New*: the entity is unknown to the entity manager, lacking a persistent identity, and without any associated tuple.

- *Managed*: the entity is associated with persistence context, c with changes to objects automatically synchronized with the 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.

4.6.1 Application architecture

In the Java Enterprise Edition (JEE), clients leverage the services of the Enterprise JavaBeans (EJB) container to establish a connection with the entity manager. Specifically, the business layer interacts with the entity manager. The advantage of using EJB is that this container provides support to make Java Persistence API (JPA) entity method calls transactional through the automatic creation of transactions.



Transactions in this context exist at three levels:

- *DBMS transactions*: managed by the Database Management System (DBMS) and utilize SQL. This level is used by the entity manager.
- *Resource-local transactions*: managed by the application and use the Java Database Connectivity (JDBC) connection interface, mapped to JDBC.
- *Container transaction*: managed by the application or the container and use the Java Transaction API (JTA) interface, also mapped to JDBC. The level utilized by the entity manager is the container transaction level.

After defining a business object, the container injects the entity manager into it. The container transparently manages instances of the entity manager for the application. It provides the transaction required to save modifications made to the entities in 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 utilizes a container-managed entity manager for persistence, the container offers a transaction for saving modifications to the database. If the same transaction is called multiple times, it is reused. This behavior is the most common and default, but the business object methods can be annotated to specify a different way to use

the transactions provided by the container. A method can be annotated to achieve the desired transactional behavior using: `@TransactionAttribute(TransactionAttributeType.type)`, where `type` can be:

- `Mandatory`: expects a transaction 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 behavior; starts a new transaction if none is active. If one is active, it 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:

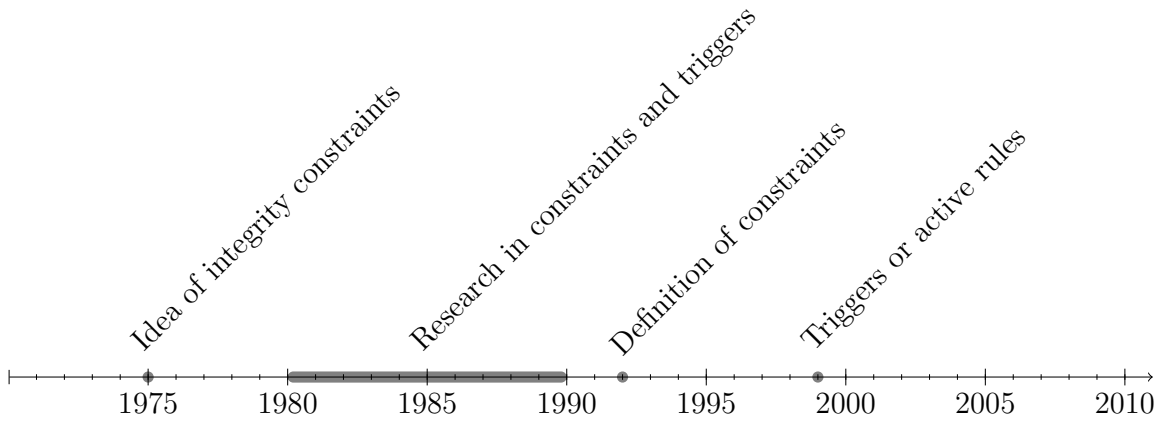
- No need to write SQL code.
- Application code ignores table names, since the mapping is expressed via annotations with defaults.
- No need to create/destroy the connection.
- No need to create and terminate the transaction.
- Business objects methods are automatically made transactional.
- Default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

CHAPTER 5

Triggers

5.1 Definition and history

Definition (*Trigger*)**.** 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 DBMS 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>
```

Modes 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.

Granularity 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 (*Transition variables*)**.** In the context of triggers, transition variables are special variables that indicate the state of a modification, both `BEFORE` and `AFTER`.

The syntax depends on the level of 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.

It'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 have 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;
```

Execution order 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.

Cascading 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.

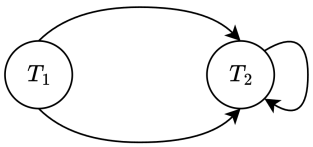
Definition (*Termination*)**.** In the context of triggers, the concept of termination ensures that, regardless of the initial state and any sequence of modifications, there is always a definitive final state, thereby preventing the possibility of infinite activation cycles.

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.

5.4.1 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

6.1 Introduction

The structure of a physical database is organized as illustrated below:

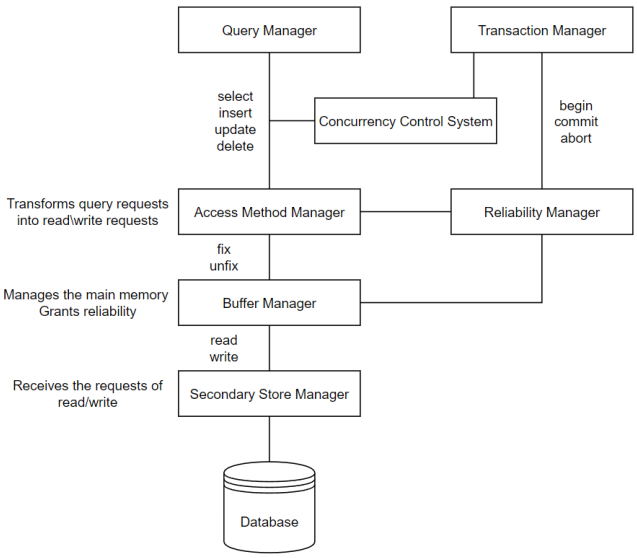


Figure 6.1: Architecture of a DBMS

Data are primarily stored in files on secondary memory due to considerations of size and persistence. Data in secondary memory becomes usable only when loaded into main memory through an I/O operation. The storage units are referred to as:

- *Block*, when in secondary memory, with a fixed length of a few kilobytes.
- *Page*, when in primary memory, assuming a size equivalent to that of blocks.

The time required for reading a block from a disk depends on the storage technology: Hard Disk Drive or Solid State Drive.

Hard disk drive Hard Disk Drives involve multiple disks stacked and rotating at a constant angular speed. A head stack, mounted on an arm, moves radially to reach tracks at various distances from the rotation axis, called spindle. A specific sector is accessed by waiting for it to pass under one of the heads. Multiple blocks can be accessed simultaneously (equal to the number of heads/disks).

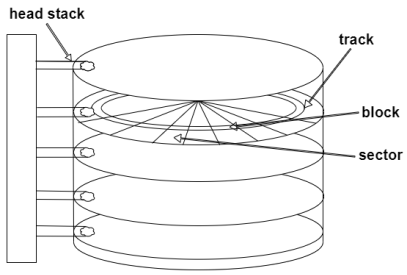


Figure 6.2: Structure of a Hard Disk Drive

Access time The secondary memory access time comprises:

- *Seek time* (8-12ms): head positioning on the correct track.
- *Latency time* (2-8ms): disc rotation on the correct sector.
- *Transfer time* ($\sim$ 1ms): data transfer.

The cost of accessing secondary memory is four orders of magnitude higher than that of main memory. In many applications, the cost depends solely on the number of accesses to secondary memory. The cost of a query is closely related to the amount of data read (moved) from secondary memory.

File System The File System, the operating system layer managing secondary memory, is utilized minimally by the DBMS. DBMS directly manage file organization, both in data distribution within blocks and internal block structures. Additionally, a DBMS may control the physical allocation of blocks on the disk to optimize access time or enhance reliability.

6.2 Physical access structures

Definition (*Access methods*)**.** The access methods are software modules that provide data access and manipulation primitives for each physical data structure.

Each DBMS has a distinctive and limited set of access methods. These methods employ specific data structures to organize data, with each table being stored in precisely one primary physical data structure, and potentially having one or more optional secondary access structures.

Classification These structures are categorized into:

- *Primary structure*: contains all the tuples of a table. Its primary purpose is to store the table content.

- *Secondary structures*: used to index primary structures, they only contain the values of certain fields interleaved with pointers to the blocks of the primary structure. Their primary function is to accelerate the search for specific tuples based on designated search criteria.

Three main types of data access structures exist: sequential, hash-based, and tree-based.

	Primary	**Secondary**
Sequential structures	Typical	Not used
Hash-based structures	In some DBMS	Frequent
Tree-based structures	Rare	Typical

Blocks and tuples Blocks serve as the physical components of files, while tuples are the logical components of tables. The block size is usually fixed and depends on the file system and disk formatting, whereas the tuple size (also known as a record) is variable within a file and contingent on the database design. For sequential and hash-based methods, a block is subdivided into the following components:

- Block header and trailer containing control information utilized by the File System.
- Page header and trailer with control information related to the access method.
- Page dictionary comprising pointers to each elementary item of useful data within the page.
- A useful part containing the data.
- Typically, page dictionaries and useful data expand as a stack in opposite directions.
- A checksum to verify the integrity of the block.

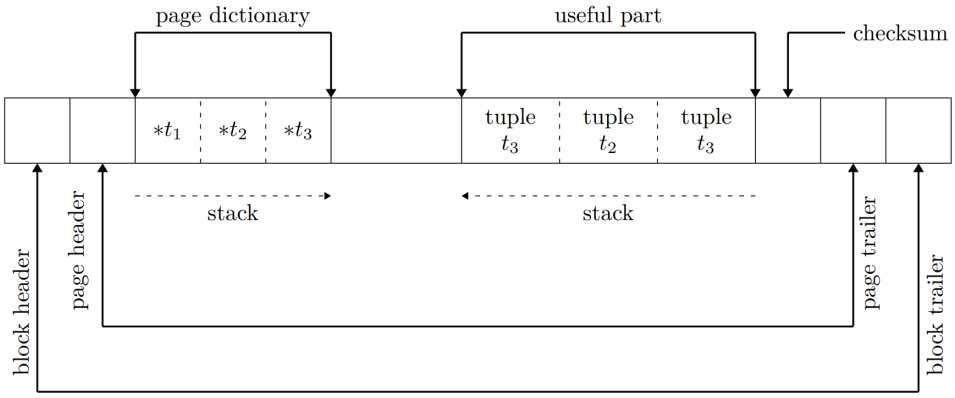


Figure 6.3: Structure of a block

The block factor (B) denotes the number of tuples within a block. To assess query costs, the following information is required:

- SR: the average size of a record (or tuple).
- SB: the average size of a block.

The block factor is defined as:

$$B = \left\lfloor \frac{\text{SB}}{\text{SR}} \right\rfloor$$

The remaining space can either be used, if the records are spanned between blocks, or not used if the records are unspanned.

Buffer manager Operations are executed in main memory and impact pages. In our cost model we assume pages of equal size and organization as blocks. The buffer operations encompass:

- *Insertion* and update of a tuple: may require a reorganization of the page or usage of a new page, also update to a field may require reorganization.
- *Deletion* of a tuple: typically accomplished by marking the tuple as invalid, with subsequent reorganization of the page performed asynchronously.
- *Access* to a field of a particular tuple: identified according to an offset with respect to the beginning of the tuple and the length of the field itself.

6.2.1 Sequential structures

In the context of sequential structures, we encounter an ordered arrangement of tuples in secondary memory, allowing for efficient access to the next element. Blocks can be either contiguous on disk or sparse. Two main possibilities arise:

- *Entry-sequenced organization*: sequence of tuples dictated by their order of entry.
- *Sequentially-ordered organization*: tuples ordered according to the value of a key.

Entry-sequenced organization Entry-sequenced organization, also known as a heap, represents the simplest and most common type of sequential organization. This organization proves to be efficient for:

- Insertion: this operation does not require shifting.
- Space occupancy: it utilizes all available blocks for data and all the space within a block.
- Sequential reading and writing: especially advantageous if the blocks are contiguous.
- Query like `SELECT * FROM table`.

However, it is inefficient for:

- Searching specific data units: this may require scanning the entire structure, a challenge mitigated by using indexes.
- Updates that increase tuple size: this may necessitate shifting and writing to another block. Deleting old versions of tuples and inserting new ones can address this issue.
- Query like `SELECT * FROM table WHERE <condition>`.

Sequentially-ordered sequential structure In this organization, tuples are sorted based on the value of a key field. It is efficient for:

- Range queries: retrieving tuples with the key in a specified interval.
- `ORDER BY` and `GROUP BY` queries: exploiting the key for sorting and grouping.

However, it is inefficient for reordering tuples within a block, especially challenging if space is limited. To address the global reordering issue, various techniques can be employed:

- Differential files and periodic merging.
- Local reordering operation within a block.
- Creation of an overflow file that contains tuples that do not fit in the current block.

Summary In real-world applications, the entry-sequenced organization is the most common solution only if paired with secondary access structures.

	Entry sequenced	**Sequentially ordered**
`INSERT`	Efficient	Not efficient
`UPDATE`	Efficient	Not efficient
`DELETE`	Invalid	Invalid
Tuple size	Fixed or variable	Fixed or variable
`SELECT * FROM T WHERE <condition>`	Not efficient	Efficient

6.2.2 Hash-based structures

Hash-based access structures provide efficient associative access to data based on the value of a key. The structure comprises N_B buckets ($N_B \ll$ number of data items), with each bucket being a unit of storage, typically equivalent to the size of one block. These buckets are often stored adjacently in the file.

A hash function maps the key field to a value between 0 and $N_B - 1$, interpreted as the index of a bucket in the hash structure (hash table). This organization proves efficient for:

- Tables with small size and (almost) static content.
- Point queries: queries with equality predicates on the key.

However, it is inefficient for:

- Range queries and full table queries.
- Tables with highly dynamic content.

Hash function The implementation of the hash function consists of two parts:

1. *Folding*: transforms key values into positive integer values, uniformly distributed over a large range.
2. *Hashing*: transforms the positive number into a value between 0 and $N_B - 1$ to identify the appropriate bucket for the tuple.

Collisions Collisions occur when two keys (tuples) are associated with the same bucket. When the maximum number of tuples per block is exceeded, collision resolution techniques are applied:

- *Closed hashing (open addressing)*: tries to find a slot in another bucket in the hash table. Linear probing is a simple technique, but not commonly used in databases.
- *Open hashing (separate chaining)*: allocates a new bucket for the same hash result, linked to the previous one.

Finding a tuple corresponding to a given key typically requires one access, although occasionally more.

Overflow chain The cost of accessing the tuple can be estimated by considering the average length of the overflow chain, which is influenced by:

- The load factor, that is the ratio of the occupied slots over the available slots:

$$\frac{T}{B \cdot N_B}$$

- The block factor B.

Here, T is the number of tuples, N_B is the number of buckets, and B is the number of tuples within a block. The average of accesses to the overflow list is summarized in the following table:

		Block factor				
		1	*2*	*3*	*4*	*5*
Load factor	*50%*	0.5	0.177	0.087	0.031	0.005
	60%	0.75	0.293	0.158	0.066	0.015
	70%	1.167	0.494	0.286	0.136	0.042
	80%	2.0	0.903	0.554	0.289	0.110
	90%	4.495	2.146	1.377	0.777	0.345

Example:
For an operation with a block factor of three and a load factor of seventy percent, the table indicates an average of 0.286 accesses to the overflow list. Consequently, the number of I/O operations is approximately 1.3.

Indexing Hash-based structures can be employed for secondary indexes, shaped and managed like a hash-based primary structure. Instead of tuples, the buckets only contain key values and pointers. Indexing without an overflow chain incurs a minimum cost of 2 I/O operations.

Summary In a large database, searching all index values to reach the desired data is inefficient. Hash-based structures exhibit good performance for equality predicates on the key field but prove inefficient for access based on interval predicates or the value of non-search-key attributes.

6.2.3 Indexes

Indexes are data structures designed to efficiently retrieve tuples based on specific criteria, known as a search key. Index entries are sorted with respect to the search key and typically contain records in the form

$$\langle \text{search key}, \text{pointer to block} \rangle$$

Primary and search keys It's crucial to distinguish between the primary key, used for uniquely identifying a tuple, and the search key, employed for retrieving tuples based on specific criteria.

	Primary key	**Search key**
Access path defined	✗	✓
Constraint defined	✓	✗
Unique	✓	✗
Implementation by index	✓	✗

Index classification Indexes are smaller than primary data structures and can be loaded into a file in the main memory. They efficiently support point queries, range queries, and sorted scans. However, adding indexes to tables requires the DBMS to update each index after an insert, update, or delete operation, which can be costly and may slow down operations.

Definition (*Dense index*)**.** An index is considered dense when it has an index entry for each search-key value in the file (one index per tuple).

This type of index provides excellent performance since a search on the index is sufficient to access the correct tuple. Dense indexing is applicable to entry-sequenced primary structures.

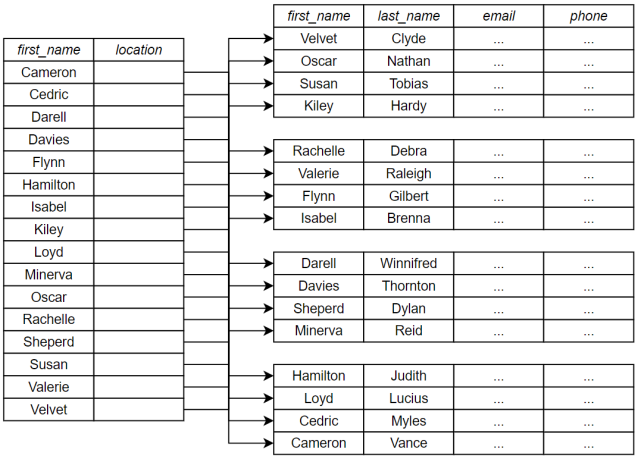


Figure 6.4: Example of dense indexing

Definition (*Sparse index*)**.** An index is termed sparse when it has index entries only for some search-key values in the file (one index per block).

While this requires less space, the search process is slower as it involves scanning an entire block to find the tuple. Sparse indexing necessitates ordered data structures and primary indexing, providing a favorable trade-off with one index entry for each block in the file.

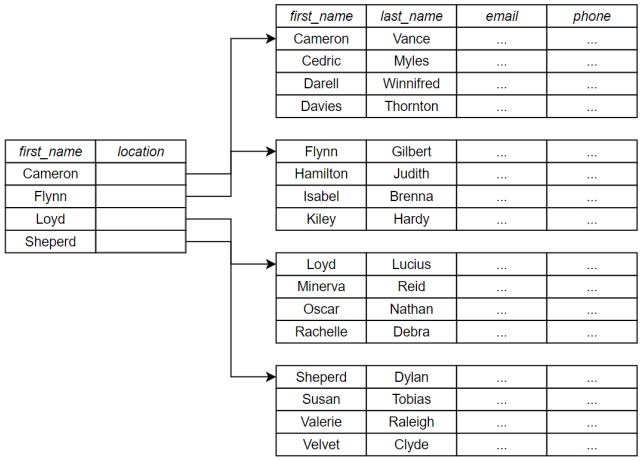


Figure 6.5: Example of sparse indexing

Primary index A primary index is defined on sequentially ordered structures. The search key (SK) is unique and coincides with the attribute according to which the structure is ordered (search key = ordering key). Only one primary index per table can be defined, typically on the primary key, but not necessarily. The primary index can be dense or sparse.

Clustering index A clustering index is a generalization of the primary index where the ordering key may not be unique. The pointer of key X refers to the block containing the first tuple of key X. It can be sparse or dense, but typically it's sparse.

Secondary index A secondary index has a search key specifying an order different from the sequential order of the file. Multiple secondary indexes can be defined for the same table, each on different search keys. It is necessarily dense because tuples with contiguous values of the key can be distributed in different blocks.

Summary The indexes are smaller than primary data structures, so they can be loaded into primary memory. They efficiently support point queries, range queries, and sorted scans. However, they are less efficient than hash structures for point queries. Adding indexes to tables requires the DBMS to update each index after an insert, update, or delete operation, incurring potential costs and slowing down data-changing operations.

Indexes in SQL To create an SQL index, every table should have:

- A suitable primary storage, possibly sequentially ordered.
- Several secondary indexes, both unique and non-unique, on the attributes most used for selections and joins. Secondary structures are progressively added, checking that they are used by the system.

Guidelines for choosing indexes:

1. Do not index small tables.
2. Index the primary key of a table only if it is not a key of the primary file organization.
3. Add a secondary index to any column heavily used as a secondary key.

4. Add secondary index structures on columns involved in `SELECT` or `JOIN` criteria, `ORDER BY`, `GROUP BY`, and other operations involving sorting.
5. Avoid indexing a column or table that is frequently updated.
6. Avoid indexing a column if the query will retrieve a significant number of rows.
7. Avoid indexing columns that consists of long strings.

The SQL commands used to create and remove the indexes are as follows:

```
CREATE [UNIQUE] INDEX <index_name> ON <table_name>[(<column_name>, ...)]
DROP INDEX <index_name>
```

6.2.4 Tree-based structures

Frequently used in relational Database Management Systems (DBMS) for secondary index structures, balanced trees support associative access based on the value of a key search field. These trees are specifically balanced in that the lengths of the paths from the root node to the leaf nodes are all equal, contributing to enhanced performance. There are two main types of balanced trees used in this context:

- *B trees*: key values are stored in both internal and leaf nodes.
- *B+ trees*: all key values are stored only in leaf nodes.

B+ trees The B+ tree is an evolution from the B-tree, with each node stored in a block. Key values are exclusively stored in the leaf nodes, making B+ trees more efficient than B-trees, especially when a majority of the nodes are leaves. The fan-out of B+ trees depends on the size of the block, the key, and the pointer.

Internal node's structure Each node contains F keys, sorted lexicographically, and $F+1$ pointers to child nodes. Each key K_j, $1 \leq j \leq F$ is followed by a pointer P_j, with K_1 preceded by a pointer P_0. Each pointer addresses a subtree, with P_0 pointing to the subtree containing keys less than K_1, P_F pointing to the subtree containing keys greater or equal to K_F, and intermediate pointers addressing subtrees containing keys K within the interval $K_j \leq K < K_{j+1}$. The value of F depends on the size of the page and the amount of space occupied by the key values and the pointers.

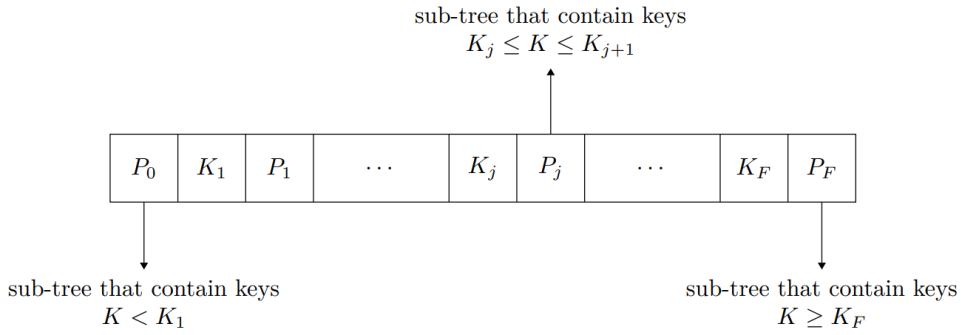


Figure 6.6: Structure of an internal node

Leaf node's structure The leaf node's structure resembles that of an internal node but contains only pointers to data tuples or the data tuples themselves. Leaf nodes can be structured in two ways:

1. The leaf node contains the entire tuple, making it key-sequenced.
2. The leaf node contains pointers to blocks of the database that contain tuples with the same key value, making it indirect.

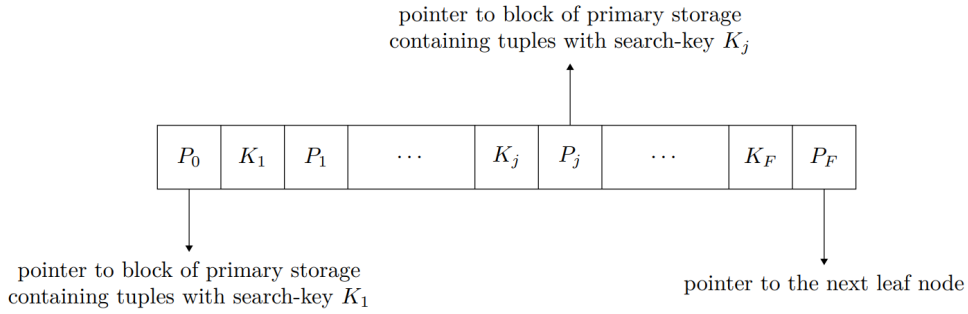


Figure 6.7: Structure of a leaf node

Search Mechanism The search mechanism involves following pointers starting from the root. At each intermediate node:

- If $V < K_1$, follow the pointer P_0.
- If $V \geq K_F$, follow the pointer P_F.
- Otherwise, follow the pointer P_j such that $K_j \leq V < K_{j+1}$.

The search continues until a leaf node is found. In a key-sequenced leaf node, the search is complete as the tuple is found. In an indirect leaf node, it's necessary to access the memory block pointed to by the pointer P_j, $0 \leq j \leq F$.

Differences B+ trees have linked leaf nodes through a chain of pointers, ordered by key values. This chain enables efficient execution of range queries, allowing sequential scanning of leaves for values within a given range. This ordered scan is not possible in B trees, where accessing tuples in a specific range requires a search for each tuple. B trees use two pointers for each value k_i in intermediate nodes, saving space in the index pages and terminating the search when a key value is found without continuing the search in subtrees.

6.3 Query optimization

The database management system's architecture includes a crucial component known as the optimizer. This optimizer plays a key role in processing SQL queries: it takes a query, conducts lexical, syntactic, and semantic analyses, and then generates an internal program using data access methods. ince there can be multiple ways to execute the same query, the optimizer is responsible for selecting the most efficient approach. The optimization process involves several steps:

1. Lexical, syntactic and semantic analysis of the query.
2. Translation into an internal representation (similar to algebraic expressions).
3. Algebraic optimization.
4. Cost-based optimization.
5. Code generation.

6.3.1 Relation profiles

Every commercial DBMS possesses quantitative information about the relations in the database, known as relation profiles. These profiles include:

- The cardinality (number of tuples) card(T) of each table T.
- The dimension in bytes size(T) of each tuple in table T.
- The dimension in bytes size(A_j, T) of each attribute A_j in table T.
- The number of distinct values val(A_j, T) of each attribute A_j in table T.
- The minimum and maximum values min(A_j, T) and max(A_j, T) of each attribute A_j in table T.

Relation profiles are derived from the data stored in tables and are periodically updated using system primitives. Cost-based optimization relies on the approximate values obtained from these relation profiles. The selectivity of a predicate, representing the probability that any row satisfies a given predicate, is a crucial metric. If val(A) = N and the values are uniformly distributed, the selectivity of the predicate $A = v$ is $\frac{1}{N}$. If no data on distribution is available, then the distribution is always assumed uniform.

Example:
Consider a memory with block size of 8 KB (8192 bytes). Consider the following schemas:

1. STUDENT (<u>ID</u>, Name, LastName, Email, City, Birthdate, Sex)
2. EXAM (<u>SID</u>, <u>CourseID</u>, Date, Grade)

In the first table we have 150000 tuples. Every tuple has a size of 95 bytes. Each block contains:

$$\frac{8192}{95} \approx 87 \text{ tuples}$$

As a result the occupied blocks are:

$$\frac{150000}{340} \approx 1700 \text{ blocks}$$

In the second table we have 1,8 million tuples. Every tuple has a size of 24 bytes. Each block contains:

$$\frac{8192}{24} \approx 340 \text{ tuples}$$

As a result the occupied blocks are:

$$\frac{1800000}{340} \approx 5300 \text{ blocks}$$

6.3.2 Internal representation of queries

The optimizer creates a representation for a query that accounts for the physical structure used to implement tables and any present indexes. This internal representation utilizes a tree structure, where leaves correspond to physical data structures and intermediate nodes represent data access operations. Operations on intermediate nodes include sequential scans, orderings, indexed accesses, and joins.

Scan operation A scan operation sequentially accesses all tuples of a table, performing algebraic or extra algebraic operations such as projection, selection, ordering, insertions, deletions, and modifications.

Indexed access operation As already seen, indexes are structures of the database that allow to access the tuples of a table in a more efficient way than a sequential scan. The DBMS chooses the corresponding index when a query involves a single supported predicate. For queries with conjunctions or disjunctions of predicates, the DBMS optimizes the use of indexes.

Join The `JOIN` operation is acknowledged as the most resource-intensive task for a DBMS due to the potential for a substantial increase in the number of tuples in the result. To address this challenge, the optimizer employs one of three techniques:

1. *Nested-loop*, based on scanning:
    - A scan is executed on a table (referred to as the external table), and for each tuple, a scan is performed on the other table (referred to as the internal table).
    - The matching process is efficient when there is an index on the join attribute of the internal table.
2. *Merge-Scan*, based on ordering:
    - This technique necessitates that both tables are sorted based on the join attribute.
    - Two synchronized scans traverse the tuples of each table in parallel, and the matching is carried out.
3. *Hash-join*, based on hashing
    - This technique requires that both tables are hashed using the join attribute.
    - The tuples of the internal table are hashed and stored in a hash table, while the tuples of the external table are scanned, and the matching is performed.

Equality The expense associated with an equality lookup is contingent upon the type of structure that represents the table:

- *Sequential structures with no index*:
    - Equality lookups are not supported, resulting in a cost equivalent to a full scan.

  - Sequentially ordered structures might incur a reduced cost.

- *Hash or Tree structures*:

  - Equality lookups are supported if A is the search key attribute of the structure.
  - The cost depends on the storage type (primary or secondary) and the search key type (unique or not).

Range The cost associated with a range lookup is contingent upon the type of structure representing the table:

- *Sequential structures* (primary):

  - Range lookups are not supported, resulting in a cost equivalent to a full scan.
  - Sequentially ordered structures may incur a reduced cost.

- *Hash structures* (primary and secondary):

  - Range lookups are not supported, leading to a cost equivalent to a full scan.

- *Tree structures* (primary and secondary):

  - Range lookups are supported if A serves as the search key attribute of the structure.
  - The cost depends on the storage type (primary or secondary) and the search key type (unique or non-unique).

- *B+ tree structures* (primary and secondary):

  - Range lookups are supported if A is the search key attribute of the structure.

Conjunction In the presence of indexes, the DBMS opts for the most selectively supported predicate for data access. Subsequently, the remaining predicates are assessed later in main memory through a sequential scan.

Disjunction If any of the predicates lack index support, a sequential scan is executed. However, if indexes are available for all predicates, the DBMS utilizes them to assess each predicate individually. The results are then merged, and duplicates are subsequently removed.

Sort Various methods facilitate the optimal sorting of a table's tuples. However, a significant challenge arises from the necessity for the DBMS to load the entire data into the buffer, a task that may exceed the available memory capacity. Sorting can occur either in main memory (utilizing ad-hoc algorithms like quick sort or merge sort) or on disk (employing external sorting algorithms). The latter approach is chosen when the data surpasses the main memory's capacity, following this procedure:

1. Split the data into chunks, each of a size equal to the main memory.

2. Load a chunk into the main memory.
3. Sort the data in the main memory.
4. Store the sorted data back on disk.
5. Merge sorted chunks using at least three pages:
    - Two for progressively loading data from two sorted chunks.
    - One as an output buffer to store the sorted data.
6. Save the result on disk.
7. Repeat from step two until all chunks are merged.

6.3.3 Cost-based Optimization

Cost-based optimization poses a challenge in decision-making within an optimization problem, involving:

- Determining which data access operations to execute.
- Deciding in which order to perform the operations.
- Choosing which option to select if there are multiple choices for a given operation.
- Establishing how to sort the data if sorting is required in the query.

To address this challenge, the DBMS leverages data profiles and approximate cost formulas. A decision tree is constructed, where:

- Each internal node represents a decision point (a choice between two or more options).
- Each leaf node represents a specific plan (a sequence of operations).

Assigning a cost to each plan allows the identification of the optimal one using operations research techniques like branch and bound. Optimizers must efficiently handle such problems within a reasonable timeframe.

6.3.4 Approaches to Query Evaluation

The query is evaluated by the DBMS according to two techniques.

Compile and store The query undergoes compilation and is stored in the DBMS for later execution. The internal code is retained in the DBMS along with indications of dependencies on specific versions of the catalog used during compilation. Upon relevant changes in the catalog, the compilation of the query is invalidated, leading to the recompilation of the query.

Compile and go The query is compiled and executed immediately, without storing the compiled code. Although the code is not stored permanently, it may persist in the DBMS for a period to be potentially reused by subsequent executions.

CHAPTER 7

Reliability

7.1 Introduction

Definition (*Reliability*)**.** Reliability is defined as the ability of an item to perform a required function under stated conditions for a stated time length.

In the realm of databases, reliability control ensures the foundational attributes of transactions:

- *Atomicity*: guaranteeing the all-or-nothing execution of a transaction.
- *Durability*: ensuring that once a transaction is committed, its effects are permanent.

The Database Management System (DBMS) implements a specific architecture for reliability control, with key components residing in stable memory and log management.

Reliability manager Within the DBMS, the reliability manager executes transactional commands such as `commit` and `abort`, which are carried out by the transaction manager. Additionally, it coordinates read and write access to data and log pages, managing recovery after system failures.

Memory persistence Durability implies a form of memory whose content endures indefinitely, abstracted over existing storage technology levels:

- *Main memory*: non-persistent.
- *Mass memory*: persistent but subject be loss.
- *Stable memory*: incapable of loss, with a failure probability of zero.

While achieving a zero probability of failure is impractical, the goal is to minimize the failure probability to a negligible value. Techniques such as replication and write protocols are employed for this purpose. The discipline addressing stable memory failure is known as disaster recovery. Stable memory can be guaranteed via:

- *On-line replication*: the data is replicated on multiple disks (e.g., RAID disk architecture).
- *Off-line replication*: the data is replicated on backup units

7.1.1 Main memory management

The objective of main memory management is to reduce data access time without compromising memory stability. This is achieved through buffers that cache data in faster memory and deferred writing onto secondary storage. A buffer page can contain multiple rows and includes:

- *Transaction counter*: indicates the number of transactions accessing it.
- *Dirty flag*: indicates whether the page has been modified and needs alignment with secondary storage.

On dedicated DBMS servers, up to 80% of the memory is allocated to the buffer.

Buffer management primitives The primitives used for buffer management are:

- `fix`: responds to a transaction's request to load a page into the buffer, returning a reference to the page and incrementing the transaction counter.
- `unfix`: unloads a page from the buffer, decrementing the transaction counter.
- `force`: moves a page from the buffer to secondary storage.
- `set_dirty`: sets the dirty flag of a page.
- `flush`: moves pages from the buffer to secondary storage when they are no longer needed.

Buffer management policies The policies used for buffer management are:

- *Write policies*: asynchronous page writing to disk concerning transactions:
  - `force`: pages are always transferred at commit.
  - `no_force`: transfer of pages can be delayed by the buffer manager.
- *De-allocation policies*:
  - `steal`: discards and flushes a page in an active transaction to disk.
  - `no_steal`: puts the transaction on a waitlist, managing the request when the page is no longer needed.
- *Pre-fetch policies*: loads pages likely to be read into the buffer before they are needed.
- *Pre-flushing policies*: de-allocates pages likely to be written from the buffer before they are needed.

Fix primitive The execution of a `fix` primitive involves the following steps:

1. If the page is in the buffer, increment the transaction counter and return a reference to the page.
2. Select a free page in the buffer (FIFO or LRU policy).
3. If found, increment the transaction counter and return a reference to the page.
4. If the dirty flag is true, flush the current page to disk before loading the new page.
5. If no page is found, select a page to be discarded, with one of two policies:
   - `steal` policy: load the new page, increment the transaction counter, and return a reference to the page.
   - `no_steal` put the transaction in a waitlist.

7.2 Failure Handling

A transaction constitutes an indivisible transformation from an initial state to a final state. The potential outcomes of a transaction include: `commit`, `rollback`, or `fault`.

In the event of a failure occurring between commit and the flushing of buffers to secondary storage, the reliability manager ensures the rolling forward of transactions and the subsequent flushing of buffers to disk. If a failure occurs before commit, the reliability manager rolls back the transactions, discarding any modifications made to the buffers.

Example:
Assuming the DBMS initiates at time T_0 and experiences a failure at T_f, with data for transactions T_2, T_3 committed and permanently stored, transactions T_1, T_6 need to be undone. In the absence of information on whether modified pages have been flushed, the reliability manager must redo T_4 and T_5.

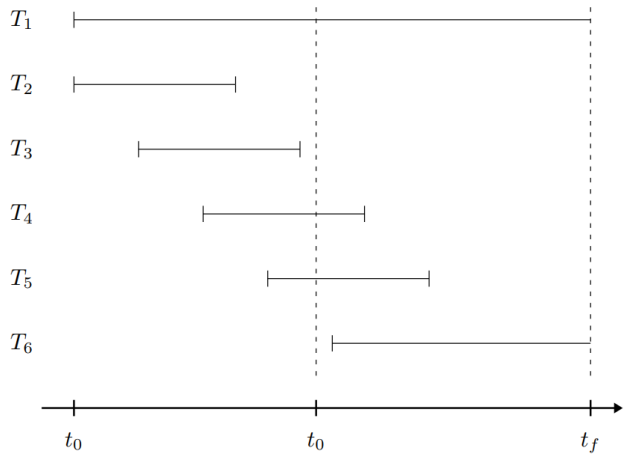


Figure 7.1: Architecture of a DBMS

7.2.1 Transaction Log

The transaction log is a sequentially written file containing records that describe the actions undertaken by various transactions. It extends up to the current instant.

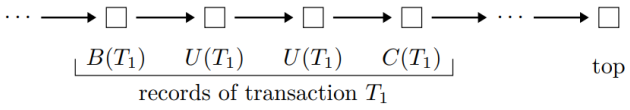


Figure 7.2: Architecture of a DBMS

The log, recorded in stable memory, takes the form of state transitions detailing the actions carried out by transactions. For an update U operation transforming object O from value v_1 to v_2, the log contains:

- `before-state(U)` $= v_1$
- `after-state(U)` $= v_2$

Logging `INSERT` and `DELETE` operations is the same as logging `UPDATE` operations, but:

- `INSERT` log record does not have a `before-state`.
- `DELETE` log record does not have an `after-state`.

Example:
Consider a transaction T_1 updating object O from value O_1 to O_2. To rollback the transaction or rectify a pre-commit failure, the log is utilized to recover the prior state:

$$\texttt{undo}\ \ T_1 : O = O_1$$

To address a post-commit failure, the log is used to recover the new state:

$$\texttt{redo}\ \ T_1 : O = O_2$$

The idempotence property applies to `undo` and `redo`:

- `undo`(T) = `undo(undo(T))`.
- `redo`(T) = `redo(redo(T))`.

Idempotence is crucial for the reliability manager, as it may need to apply these operations multiple times. For instance, if a failure occurs after the commit of T_1 and the log is not flushed, the reliability manager applies `redo` to T_1 and then T_2. This property ensures repeated application without altering the database state.

Log types The logs concerning transactional commands are:

- `B(T)`: begin transaction `T`.
- `C(T)`: commit transaction `T`.
- `A(T)`: abort transaction `T`.

The logs concerning `UPDATE`, `DELETE` and `INSERT` operations are:

- `U(T, O, v_1, v_2)`: update object `O` from value `v_1` to value `v_2` in transaction `T`.
- `D(T, O, v)`: delete object `O` with value `v` in transaction `T`.
- `I(T, O, v)`: insert object `O` with value `v` in transaction `T`.

Transactional rules Log management must ensure that transactions implement `write` operations reliably. To achieve this, the reliability manager adheres to the following rules:

1. A commit log record must be written synchronously (with a force operation).
2. The before-state must be written in the log before executing the corresponding operation on the database (write-ahead logging).
3. The after-state must be written in the log before executing the commit (commit rule).

The second rule ensures that actions can be undone in case of failure, while the third rule ensures that actions can be redone in case of failure.

7.2.2 Failures types

There are three types of failures:

- Soft failure: loss of part or all of the main memory and requires a warm restart. To replay the transactions it is possible to exploit the log.
- Hard failure: failure or loss of part or all of the secondary memory devices and requires a cold restart. To replay the transactions it is possible to exploit the dump.
- Disaster: loss of stable memory (of the log and the dump).

7.2.3 Checkpointing

Checkpoints serve to minimize the amount of work needed in case of failure by capturing snapshots of the database and log at a specific time. Periodically, the Reliability Manager initiates checkpoints, where committed transactions flush their data from the buffer to the disk, and all active transactions are recorded in the log. A straightforward checkpointing strategy involves the following steps:

1. Acceptance of all transactions that have committed their work.
2. Suspension of all abort requests.
3. Forceful transfer of all dirty buffer pages modified by committed transactions to mass storage.
4. Recording the identifiers of transactions still in progress in the checkpoint log record; no new transaction can start while this record is being written.
5. Resumption of the acceptance of operations.

This ensures that all data related to committed transactions resides on mass storage, while transactions still in progress are listed in the checkpoint log record (in stable memory).

Dump A dump is a comprehensive backup of the database at a specific time. Dumps are typically created during periods of low database activity, such as at night or over the weekend. The availability of the dump is recorded in the log, while the dump's content itself is stored in stable memory.

7.2.4 Database restarting

Warm restart The warm restart is performed according to the following steps: The warm restart involves the following steps:

1. Finding the most recent checkpoint.
2. Building the `UNDO` and `REDO` sets:

```
UNDO := active transactions in the checkpoint
REDO := {}
for log records from the checkpoint to the top of the log do:
    if B(T_i) then
        UNDO := UNDO $\cup$ {T_i} // started, maybe undone
    else if C(T_i) then
        UNDO := UNDO $\setminus$ {T_i}
        REDO := REDO $\cup$ {T_i} // ended, to redo
    end if
end for
```

3. Undoing the transactions in the `UNDO` set by applying the `UNDO` operations in reverse order.
4. Redoing the transactions in the `REDO` set by applying the `REDO` operations.

Cold restart The cold restart involves the following steps:

1. Restoring data starting from the last backup (dump).
2. Replaying operations recorded into the log before the failure time.
3. Restoring data on disk to the state existing at the failure time.
4. Performing a warm restart.

During the cold restart, uncertain transactions are undone.