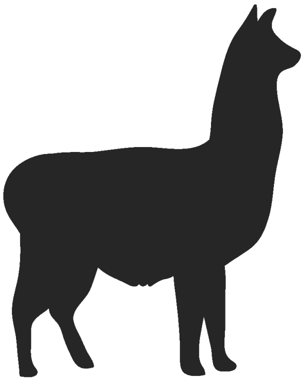
Notes

Lorenzo Rossi and everyone who kindly helped!

2023/2024

Last update: 2023-01-09



no alpaca has been harmed while writing these notes

Contents

Data bases 22022/2023

1 Introduction

1.1 Architecture of the *DBMS*

A ***DBMS*** *(short for Data Base Management System)* is a system *(or software product)* capable of managing data collections that can be:

- **Large**
  → much larger than the central memory available on the computer that runs the software
  → often data must be stored on secondary storage devices
- **Persistent**
  → its lifespan is longer than the lifespan of the software that accesses it
- **Shared**
  → used by several applications at the same time
  → various users must be able to gain access to the same data
- **Reliable**
  → ensures tolerance to hardware and software failures
  → the DBMS provides backup and recovery capabilities
- **Data-ownership respectful**
  → the data access is disciplined and controlled by the *DBMS*
  → users can only access the data they are authorized to

Capabilities of the *DBMS*:

- **Transaction** management
  → *ACID* properties make sure that a set of operations is performed as a single unit
- **Concurrency** control
  → pessimistic and optimistic locking prevent data corruption in presence of concurrent accesses
  → also known as *CC* theory
- **Reliability** control
  → log and recover protocols prevent data loss in case of failures
- **Buffer** and secondary memory management
  → paging and caching techniques improve performance by reducing the number of disk accesses
- **Physical data structures** and **access** structures
  → sequential, hash-based and tree-based structures are some of the low-level data structures used by the *DBMS*
- **Query** management
  → cost-based query optimization techniques are used to find the best execution plan for a given query

1.2 Database-Application integration

- **Impedance mismatch** handling
  → differences between database and application models are solved with code-level procedures and object-relational mapping
- Database **communication**
  → *DBMS* provides call level interfaces, `ODBC-JDBC`, and `JPA` persistence provider

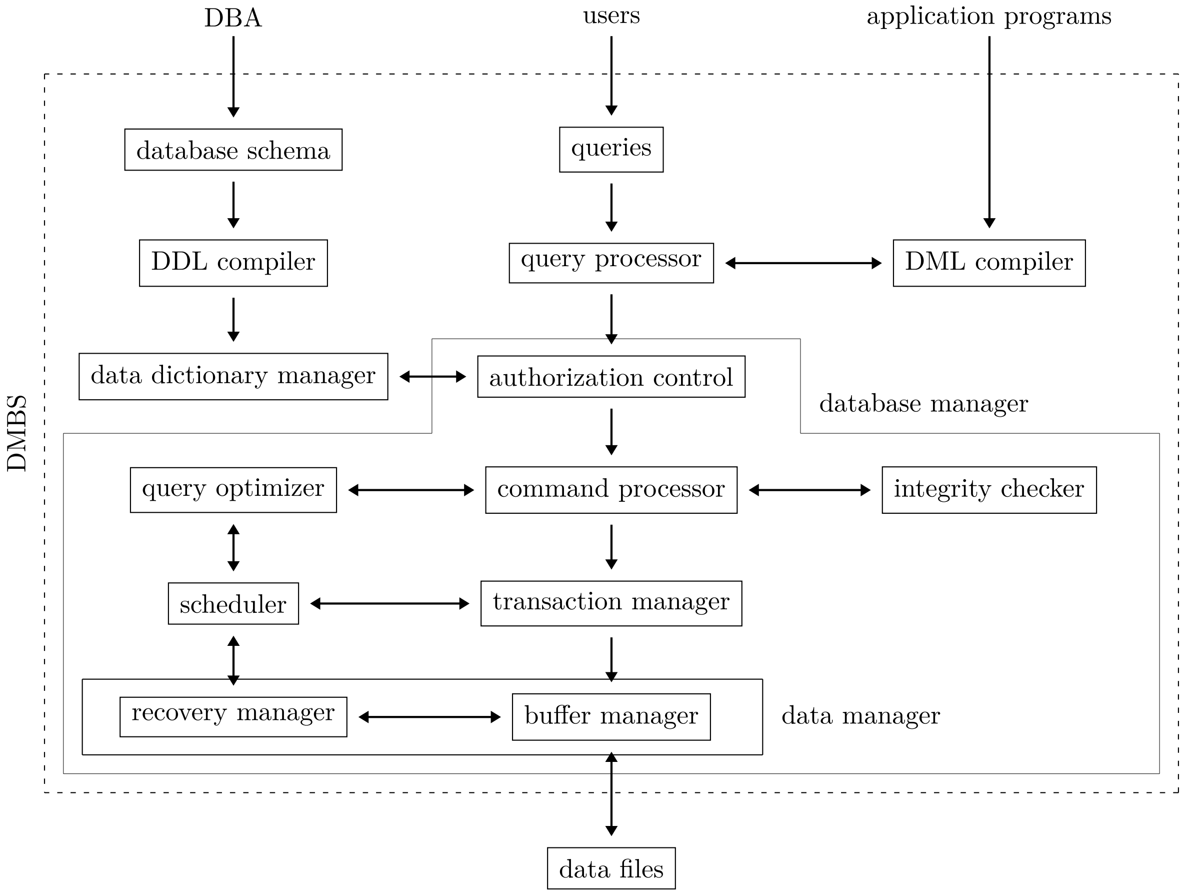


Figure 1: Architecture of the *DBMS*

- → the state of an object and the state of the persistent data that corresponds to it is synchronized by the *DBMS* via `JPA` manage entities

- Data **ranking**
  - → questions regarding all kinds of data preference and ranking are solved by the simultaneous optimization of several criteria

2 Transactions

A **transaction** is an elementary, atomic unit of work performed by an application. The need for a transaction arises when multiple operations must be performed in a single step or when the data in the application can be manipulated between multiple users at the same time *(properties of reliability and isolation)*.
Each transaction is encapsulated in a **transaction boundary**, defined by the commands:

1. `begin transaction` or `bot`
2. `end transaction` or `eot`

Within a transaction, one of two commands **is executed exactly once** to signal the end of the transaction:

1. `commit-work`, the transaction is committed
2. `rollback-work`, the transaction is aborted and rolled back

A transaction is defined as **well formed** if it fulfils the following conditions:

1. It **begins** its execution with a `begin transaction` command
2. It **ends** its execution with a `commit-work` or `rollback-work` command
3. It **includes** only one command between `commit-work` and `rollback-work`

An application is normally composed of multiple transactions, which are executed in a sequence.

2.1 ACID properties

A transaction must possess 4 peculiar properties *(called **ACID**)*:

- **Atomicity**
  - a transaction is an indivisible unit of execution: it either **succeeds** or **fails** completely
    - if it fails, the data is **rolled back** to the state it was before the transaction started
    - an error after the end does not alter the effect of the transaction
  - if a transaction fails, the *DBMS* must restore the database to its state before the transaction started
- **Consistency**
  - the carrying out of the transaction **does not violate any integrity constraint** defined on the database
    - if that happens, the transaction itself is aborted by the *DBMS*
  - immediate constraints can be checked by the *DBMS* before the transaction is committed, while deferred constraints can be checked only after
  - if the initial state S_0 is consistent then the final state S_f is also consistent, while intermediate state S_i may not be consistent
- **Isolation**
  - the execution of a transaction is independent of the simultaneous execution of other transactions
  - the parallel execution of a set of transactions gives the result that the same transaction would obtain by carrying them out singularly
  - isolation impacts performance and trade-offs can be defined between isolation and performance
- **Durability**
  - the effects of a correctly committed transaction are permanent
  - no piece of data is ever lost, for any reason

2.2 *DBMS* and ACID properties

The mechanisms provided by the *DBMS*, conform to the *ACID* properties, are:

- **Atomicity**
  - **abort**: the transaction is aborted and rolled back

  - **rollback**: the transaction is rolled back to the state it was before the transaction started
  - **restart**: the transaction is restarted
  - **reliability manager**: the *DBMS* provides a reliability manager that manages the execution of the transaction
- **Consistency**
  - **integrity checking of the *DBMS***: the *DBMS* checks the integrity of the database before the transaction is committed
  - **integrity control system at query execution time**: the *DBMS* checks the integrity of the database at query execution time
- **Isolation**
  - **concurrency control**: the *DBMS* provides a concurrency control system that manages the execution of the transaction
- **Durability**
  - **recovery management**: the *DBMS* provides a reliability manager that manages the execution of the transaction

3 Concurrency

Since multiple applications can access the same data at the same time, the *DBMS* must provide a mechanism to control the concurrent access to the data. The application load of a *DBMS* can be measured using the number of transactions per second *(TPS)*; by exploiting the parallelism, the *TPS* can be increased.
The **Concurrency Control System** *(or CC system)* manages the execution of transactions, avoiding the insurgence of anomalies while ensuring performances. The anomalies can be:

- **Update loss**
  - two transactions try to modify the same data, resulting in the loss of one of the updates
- **Dirty read**
  - a transaction reads data that has been modified by another transaction that aborts
  - this is a problem with a difficult solution
- **Inconsistent read** - *(phantom read)*
  - a transaction reads data that has been modified by another transaction that commits
- **Phantom insert** - *(phantom update)*
  - a transaction writes data that has been read by another transaction that commits

3.1 Concurrency control theory

Model: an abstraction of a system, an object of process, which purposely ignores some details to focus on the relevant aspects.
The concurrency theory builds upon a model of transaction and concurrency control principles that helps understand real-world systems; they exploit implementation-level mechanisms *(like locks and snapshots)* that help achieve some of the desirable properties postulated by the theory.
For the sake of simplicity, the concurrency theory is based on the following assumptions:

- A **transaction** is a **syntactical object**, of which only the *input* and *output* actions are known
- All transactions are **initiated** by the `begin-transaction` command
- All transactions are **terminated** by the `end-transaction` command
- The concurrency control system **accepts** or **refuses** concurrent executions during the evolution of the transactions, without knowing their outcome *(either with a `commit` or `abort` command)*
- An **operation** is a **read** or **write** of a specific datum by a specific transaction
- A **schedule** is a **sequence of operations** performed by concurrent transactions that respect the order of operations of each transaction

Transactions notation

- A **transaction** is denoted by T_i, where i is a number
- A `READ` operation on data x is denoted by $r_i(x)$
- A `WRITE` operation on data x is denoted by $w_i(x)$
- A **schedule** is denoted by the letter S

3.1.1 Schedules

Let N_S and N_D be respectively the number of **serial schedules** and **distinct schedules** for n transactions $\langle T_1, \dots, T_n \rangle$ each with k_i operations. Then:

$$N_S = n! \qquad \text{number of permutations of } n \text{ transactions}$$

$$N_D = \frac{\left(\sum_{i=1}^{n}\right)!}{\prod_{i=1}^{n}(k_i!)} \qquad \text{number of permutations of all operations}$$

3.1.2 Principles of Concurrency Control

The goal of the **Concurrency Control** is to **reject schedules that cause anomalies**. To enable Concurrency Control, two components are needed:

1. **scheduler**, a component that accepts or rejects the operations requested by the transactions
2. **serial schedule**, a schedule in which the actions of each transaction occur in a contiguous sequence

A **serializable schedule** is a schedule that leaves the database in the same state as some serial schedule of the same transactions; this property is commonly accepted as a notion of **schedule correctness**.
To identify classes of schedules that ensure serializability, it's required to establish a notion of **schedule equivalence**. However, there's a difference between real life and theory:

- **in theory**, all transactions are observed *a posteriori* and limited to those that have committed
  - this technique is called **commit projection**
  - the observed schedule is admissible if the transactions lead to a valid state
- **in practice**, all schedulers must make decisions while the transactions are still running

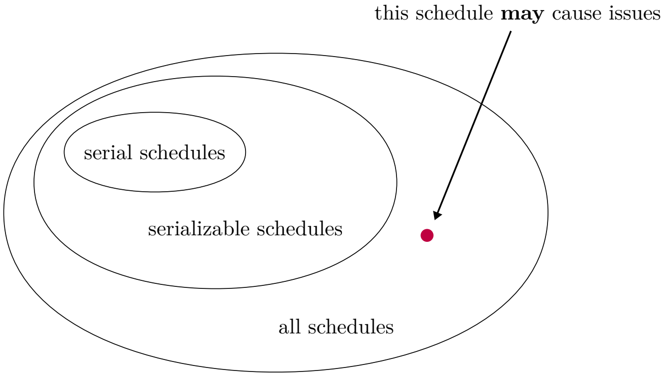


Figure 2: Schedules equivalence

And finally:

- **Reads-from** relation: $r_i(x)$ reads from $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ in S and there's no `WRITE` operation between them
  - this relation is independent of the time at which the commit T_j occurs
- **Final write**: $w_i(x)$ in a schedule S is a final write if it is the last write on x that occurs in S
- **Blind write**: $w_i(x)$ in a schedule S is not a final on x and the following operation on x is also a write
- **View equivalence**: two schedules S_i and S_j are view equivalent *($S_i \approx_v S_j$)* if they have the same operations, the same final writes and the same final reads
- **View serializability**: a schedule is view serializable if it is view-equivalent to a serial schedule of the same transactions
  - the class of view-serializable schedules is called *VSR*

3.1.2.1 Complexity of View Serializability

Deciding whether two given schedules are view equivalent is done in polynomial time and space; deciding whether a generic schedule is in *VSR* is a $\mathcal{NP}$*-complete* problem, as it requires considering the *reads-from* and *final writes* of all possible serial schedules with the same operations *(a combinatorial problem)*.
However, by giving up some accuracy, it's possible to increase the performance: a stricter definition of view equivalence is introduced. This simplification may lead to the rejection of the schedules that are view-serializable under the broader definition.

3.1.3 Conflict Serializability

First, the notion of **conflict** is introduced:

- Two operations o_i and o_j, with $i \neq j$, are **in conflict** if they address the same resource and at least one of them is a `WRITE`
  - *read-write* conflicts R-W or W-R
  - *write-write* conflicts W-W

Then, the notion of **conflict serializability** is defined:

- Two schedules S_i and S_j are **conflict-equivalent** *($S_i \approx_C S_k$)* if they contain the same operations and in all the conflicting pairs the transactions occur in the same order
- A schedule is conflict-serializable if it is **conflict-equivalent to a serial schedule** of the same transactions
- The class of conflict-serializable schedules is called *CSR*

3.1.3.1 Relation between VSR and CSR

First of all, it's immediate to establish that *VSR* $\subseteq$ *CSR*

- **Proof**: there are *VSR* schedules that are not *CSR* because they contain operations that are not in conflict.
- **Counter example**: consider the schedule $r_1(x)w_2(x)w_1(x)w_3(x)$
  - it's view-serializable, as it's view-equivalent to the schedule $T_1T_2T_3 = r_1(x)w_1(x)w_2(x)w_3(x)$
  - it's not conflict-serializable, as it contains R-W and W-W conflicts
  - there is no conflict-equivalent serial schedule

Therefore it can be deducted that *CSR* $\Rightarrow$ *VSR*: conflict-equivalence $\approx_C$ implies view-equivalence $\approx_v$, by assuming that $S_1 \approx_C S_2$ and $S_2 \approx_v S_3$.
To achieve this assumption, S_1 and S_2 must have:

- The **same final writes**
  - if they didn't, there would be at least two writes in a different order, and therefore they would not be conflict-equivalent
- The **same reads-from** relations
  - if they didn't, there would be at least two reads in a different order, and therefore they would not be conflict-equivalent

3.1.3.2 Testing *CSR*

As already said, determining the conflict serializability of two generic schedules is a $\mathcal{NP}$-*complete* problem. To test the conflict-serializability of a schedule, it's necessary to build a **conflict graph** *(or CG)* that has:

- One **node** for each transaction T_i
- One **arc** from T_i to T_j if 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

The schedule is conflict-serializable *(in CSR)* if and only if the conflict graph is acyclic.

Finally, each schedule $S \in$ *VSR* and $S \notin$ *CSR* has cycles in its *CG* due to the presence of blind writes. Such pairs can be swapped without affecting the reads-from and final-write relationships creating a new schedule S^{mod} that is view equivalent to S; its *CG* is acyclic and can be used to find serial schedules that are transitively view equivalent to S.

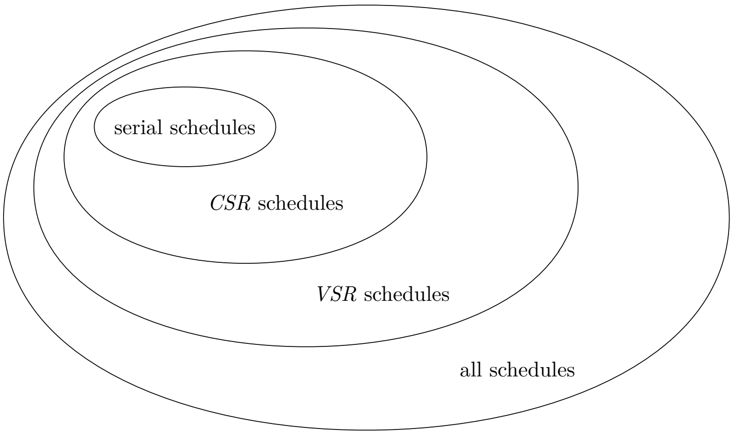


Figure 3: Relation between *VSR* and *CSR*

3.1.3.3 *CSR* implies acyclicity of the *CG*

This Paragraph is going to provide a simple explanation of the previous statement.
Consider a schedule S in *CSR*. As such, it is $\approx_c$ to a serial schedule S'. Without loss of generality, the transaction of S can be renamed such that their order is $T_1, T_2, \ldots, T_n$.
Since the serial schedule has all conflicting pairs in the same order as schedule S, the *CG* will have arcs only connecting the pairs (i, j) with $i < j$. As a direct consequence, the *CG* will be acyclic *(a cycle requires at least an arc (i, j) with $i > j$).*

3.1.3.4 Acyclicity of the *CG* implies *CSR*

Consider once again the schedule S already explored in the previous Paragraph.
If the *CG* is acyclic, then it induces a topological ordering on its nodes *(an ordering such that the graph only contains arcs from i to j if $i < j$).* The same partial order exists on the transactions of S.
Generally speaking, any serial schedule whose transactions are ordered according to the partial order is conflict-equivalent to S, because for all conflicting pairs i, j the transaction T_i precedes T_j in both schedules.

3.2 Concurrency control in practice

In the real world, the concurrency control methods explained so far are not used directly: *CSR* checking would be efficient if it was possible to know the graph in the beginning, but it's not the case. Additionally, as stated, the problem is *$\mathcal{NP}$-complete*, so it's not possible to solve it in a reasonable time.
A scheduler must work *"online" (or "on the fly")*: it must be able to decide for each operation if it can be executed or not, without knowing the whole schedule in advance; if it's not possible to maintain the conflict graph, then it has to be updated and its acyclicity checked after each operation.
The assumption that concurrency control can work only with the commit-projection of the schedule is not realistic as transactions can be aborted at any time. To solve this issue, a simple decision criterion is required for the scheduler. It must:

- **Avoid** as many **anomalies** as possible
- Have **negligible overhead**

So far, the notation $r_1(x)w_2(x)w_1(x)w_3(x)$ has been used to represent a schedule, or a posteriori view of the execution of concurrent transactions in the *DBMS (also sometimes called history).* A schedule represents *"what*

has happened" or, with more detail, *"which operations have been executed by which transactions in which order"* . They can be further restricted by the commit-projection hypothesis so operations are executed by committed transactions.
When dealing with concurrency control, it's important to consider **arrival sequences**: sequences of operation requests emitted in order by transactions. With abuse of notation, the arrival schedule will be denoted in the same way as the a posteriori schedule. The distinction will be clear from the context.
The *CC* system maps an arrival sequence to an a posteriori schedule, and it must guarantee that the a posteriori schedule is conflict-serializable.

To implement a real *CC* system, two main approaches are used in the real world:

- **Pessimistic**
  - based on **locks** or resource access control
  - if a resource is being used, no other transaction can access it
  - **prevents the errors from happening**
- **Optimistic**
  - based on **timestamping** and versioning
  - serve as many requests as possible, possibly using out-of-date data
  - **solves the error after it has happened**

Both families of approaches will be compared later.

3.2.1 Locking

The concurrency control mechanism implemented by most *DBMS* is called **locking**. It works on a simple principle: a transaction can access a data item *(either through a `WRITE` or a `READ` operation)* only if it has acquired a lock on it.
Three primitive operations are defined:

> `r_lock(`x`)`, used to acquire a read lock on x
> `w_lock(`x`)`, used to acquire a write lock on x
> `unlock(`x`)`, used to release the lock on x

The **scheduler** *(also called lock manager in this context)* receives those requests and decides if they can be executed or not by checking an adequate data structure with minimal computational cost and overhead.
During the execution of `READ` and `WRITE` operations, the following rules must be respected:

- Each `READ` operation should be preceded by an `r_lock` and followed by an `unlock`
  - this type of lock is **shared**, as many transactions can acquire it at the same time
  - this lock can be upgraded into a `w_lock` via lock escalation
- Each `WRITE` operation should be preceded by a `w_lock` and followed by an `unlock`
  - this type of lock is **exclusive**, as only one transaction can acquire it at the same time

When a transaction follows these rules, it's called **well formed with regard to locking**. The object can then be in one of 3 possible states:

1. `free` or `unlocked`: no transaction has acquired a lock on it
2. `r-locked`: at least one transaction has acquired a `READ` lock on it
3. `w-locked`: exactly one transaction has acquired a `WRITE` lock on it

The lock manager receives the primitives from the transaction and grants resources according to the conflict table (shown in Table 1).

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	✗	✓/✗ depends on n	✓ free

Table 1: Conflict table for locking: `n` is the number of concurrent readers on the object, incremented by `r_lock` and decremented by `unlock`.

3.2.1.1 Predicate locks

To prevent phantom reads and inserts, a lock should also be placed on *"future"* data *(inserted data that would satisfy previously issued queries)*; to achieve this goal, a new type of lock is introduced: the **predicate lock**, represented by the symbol ϕ.
Predicate locks extend the notion of data locks to future data:

- the **lock manager** can **acquire** a predicate lock on a predicate ϕ
- Other **transactions cannot insert, delete or upgrade** any tuple satisfying ϕ
- if the predicate lock is not supported, the transaction must acquire a lock on all the tuples satisfying ϕ; otherwise, the lock is managed via the help of **indexes**

3.2.1.2 Locks implementation

Typically, locks are implemented via lock tables: hash tables index the lockable items via hashing. Each locked item has a linked list of transactions that have required a lock on it. Every new lock request for the data item is appended as a new node to the list; locks can be applied to both data and index items. An illustration of the lock table is shown in Figure 4.

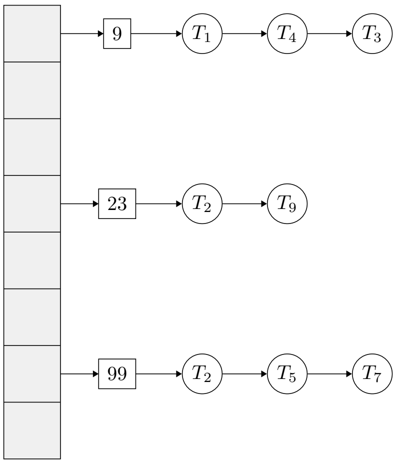


Figure 4: Illustration of a lock table

Resources can be *free*, *read-locked*, or *write-locked*. To keep track of the number of readers, a counter is used. Transactions requesting locks are either granted the lock or suspended and queued *(handled via a `FIFO` policy)*. This technique may cause:

- **Deadlocks**
  - two or more transactions are stuck in endless **mutual wait**
  - typically occurs due to transactions **waiting for each other** to release a lock

    - explained in Section 3.5
- **Starvation**
    - a transaction is waiting for a lock **for a long time**
    - typically occurs due to write transactions **waiting for resources** that are continuously *read-locked* by other transactions

3.3 Two-Phase Locking - *2PL*

The locking method ensures that the writing actions are exclusive while reading actions can occur concurrently; however, it doesn't guarantee that the reading actions are consistent with the writing actions.
The schedule mapped by the *CC* system can be **inconsistent** if the following conditions are met:

- a transaction T_1 reads a data item x and then writes it
- a transaction T_2 reads x before T_1 writes it

To avoid this situation, the locking method can be extended with a **two-phase locking** mechanism *(also called 2PL for brevity).* This method introduces a new restriction: **a transaction, after having released a lock, cannot acquire another lock**.
As a consequence of this principle, two phases can be distinguished:

1. **Growing phase**
    - the transaction **acquires** all the locks it needs to execute its operations
    - the transfer of an `r_lock` into a `w_lock` can only appear in this phase
2. **Shrinking phase**
    - the transaction **releases** all the locks it has acquired

An illustration of the resources used in the two phases is shown in Figure 5.

Figure 5: Illustration of the resources used in the two phases of the *2PL*

This extension is sufficient to prevent non-repeatable reads and phantom reads, but it doesn't prevent dirty reads; however, it does ensure serializability.

Finally, a **scheduler** that:

- only processes **well-formed transactions**
- grants locks according to the **conflict table**

- checks that all transactions apply the **two-phase locking**

generates schedules in the *2PL* class; as soon will be shown, those schedules are both **view-serializable** and **conflict-serializable**.
It can be noted that:

$$VSR \subset CSR \subset 2PL$$

3.3.1 *2PL* implies *CSR*

If $2PL \subseteq CSR$, then **every *2PL* schedule is also conflict serializable**.

First, let's suppose that a *2PL* schedule is not *CSR*. Then, there exists a transaction T_i that has acquired a lock on a data item x and then released it (a cycle $T_i \rightarrow T_j \rightarrow T_i$).
Therefore there must be a pair of conflicting operations in reverse order, such that:

1. $\mathrm{OP}_i^h(x), \mathrm{OP}_j^k(x) \dots \mathrm{OP}_j^u, \dots \mathrm{OP}_i^w(x)$ where:
    - at least one of $\mathrm{OP}_i^h, \mathrm{OP}_j^k$ is a `WRITE` operation
    - at least one of $\mathrm{OP}_j^u, \mathrm{OP}_i^v$ is a `READ` operation
2. when the instructions $\mathrm{OP}_i^h, \mathrm{OP}_j^k$ are executed, the transaction T_i has acquired a lock on x
3. later in the schedule, when the instructions $\mathrm{OP}_j^u, \mathrm{OP}_i^v$ are executed,
    - for a conflict to occur, the transaction must have acquired a lock on x
    - this is a contradiction on the *2PL*

The inclusion of *2PL* in *CSR* is therefore proved; this proves also that all *2PL* schedules are view-serializable too.
In the same way, the inclusion of *2PL* in *VSR* can be proved *(showing that $CSR \subset VSR$).*

3.3.1.1 Alternate proof

An alternate proof of the same relation can be found in the following steps.

- Consider a generic conflict $o_i \rightarrow o_j$ in S' with $o_i \in T_i, o_j \in T_J, i < j$
    - by definition, o_i and o_j address the same resource r and at least one of them is a `WRITE`
- Is it possible that o_i and o_j execute in the reverse order?
    - no, because then T_j would have released the lock on r before T_i acquired it
    - this would be a contradiction of the ordering criterion in the *2PL*

This also proves that all *2PL* schedules are view-serializable too and that they can be also checked with negligible overhead.

3.3.1.2 *2PL* is smaller than *CSR*

It can be proven that $2PL \subset CSR$: every *2PL* schedule is also conflict serializable, while the opposite is not always true.

Figure 6 shows the relation between the three classes of schedules.

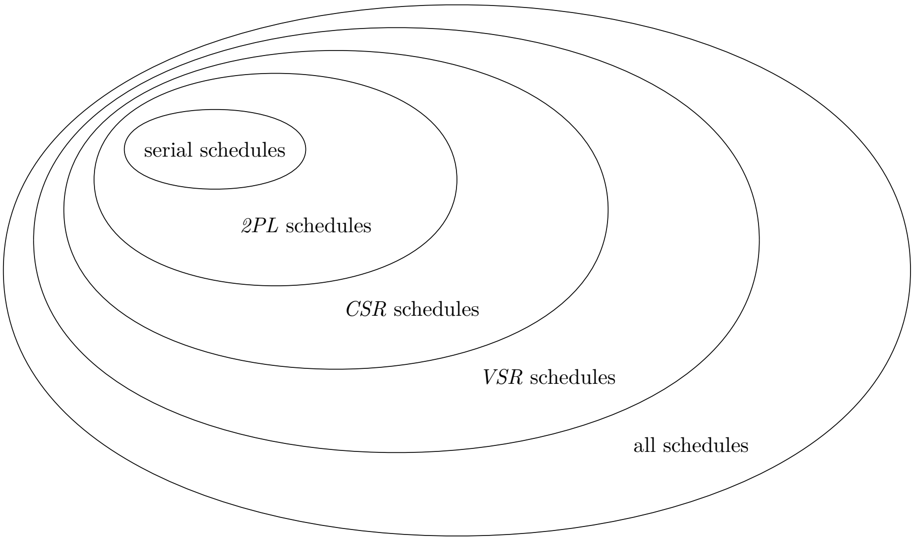


Figure 6: Relation between *2PL*, *CSR* and *VSR* schedules

3.4 Strict *2PL*

In the previous Sections, the hypothesis of commit-projection *(no transactions in the schedule are aborted)* was used; however, the *2PL* does not protect against dirty reads *(caused by uncommitted transactions)*, as releasing locks before rollbacks expose dirty data. In the same way, neither *VSR* nor *CSR* protects against dirty reads.
To remove this hypothesis, an additional constraint to *2PL* is added, therefore obtaining the ***Strict 2PL***: **locks held by a transaction can be released only after commit or rollback.**
This version of *2PL* is used in most of the modern *DBMS* whenever a level of higher isolation is required.

In practice:

- *Strict 2PL* locks are also called **long duration locks**, while *2PL* locks are called **short duration locks**
- Real systems may apply *2PL* policies differently, depending on the type of the lock:
  - **write locks** are usually released after commit or rollback *(long duration)*
  - **read locks** are usually implemented with mixed policies *(both short and long duration)*
- Long-duration read locks are costly in terms of performance and real systems replace them with more complex mechanisms

3.4.1 Use of long duration write locks

Consider the following schedule *(admissible if write locks are of short duration)* of two transactions T_1, T_2 without the hypothesis that aborts are not possible:

1. $w_1(x), \ldots, w_2(x), \ldots, ((c_1 \vee a_1) \wedge (c_2 \vee a_2)$ in any order)

   → T_2 is allowed to write over the same object updated by T_1 which has not yet completed
2. Transaction T_1 aborts
   - the schedule looks like $w_1(x), \ldots, w_2(x), \ldots, a_1, (c_2 \vee a_2)$
   - there are 2 ways to process a_1:
     1. if x is restored to the state before T_1, then T_2's update is lost; if it commits, x has a stale value

2. if x is not restored and T_2 aborts, then its previous state cannot be reinstalled

To solve this problem, write locks are held until the completion of the transaction to enable proper processing of aborts. The anomalies of the above non commit-projection schedule are called **dirty write** *(or dirty write anomaly).*

3.4.2 Isolation levels in `SQL`

The `SQL` standard defines transaction isolation levels which specify the anomalies that should be prevented by the *DBMS*; however, the level does not affect write locks. A transaction is always able to get *(and hold)* an exclusive lock on any data it modifies until its commit point, regardless of the isolation level. For read operations, levels define the degree of protection against modifications made by other transactions. Furthermore, different systems offer different guarantees about lost updates.
The different isolation levels are defined in Table 2.

`SQL` isolation levels may be implemented with the appropriate use of locks: Table 3 shows the locks that are held by the *DBMS* for each level. Normally, commercial systems use locks and timestamp-based concurrency control mechanisms.

error →	*dirty read*	*non repeatable read*	*phantoms*
isolation level ↓			
read uncommitted	✓	✓	✓
read committed	✗	✓	✓
repeatable read	✗	✗	✓
snapshot	✗	✗	✗
serializable	✗	✗	✗

Table 2: `SQL` isolation levels

	no isolation	*read uncommitted*	*read committed*	*repeatable read*	*serializable*
`READ`	*none*	*none*	*RL*	*RL*	*FS*
`WRITE - UPDATE`	*none or RU*	*RU*	*RU*	*RU*	*FU or IX*
`SELECT`	-	-	-	-	*IX*

Table 3: `SQL` isolation levels and locks. Legend: *RL* = record lock; *FS* = file lock; *RU* = record update lock; *FU* = file update lock; *IX* = intent lock

It's important to notice that serializable transactions don't necessarily execute serially: the requirement is that transactions can only commit if the result would be as if they had been executed serially in any order.
The locking requirements to achieve this level of isolation can frequently lead to deadlocks where one of the transactions needs to be rolled back. Because of this problem, the *serializable* level is used sparingly and it's not the default in most *DBMS*.

The `SQL` code to set the isolation level is shown in Code 1.

```
SET TRANSACTION ISOLATION LEVEL <isolation level>;
<set transaction statement> ::=
  SET [LOCAL] TRANSACTION <transaction characteristics>

<transaction characteristics> ::=
  [ <transaction mode> [{, <transaction mode>}...]]

<transaction mode> ::=
  <isolation level> | <transaction access mode> | <diagnostics size>

<transaction access mode> ::=
  READ WRITE | READ ONLY

<isolation level> ::=
  ISOLATION LEVEL <isolation level name>

<isolation level name> ::=
  READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

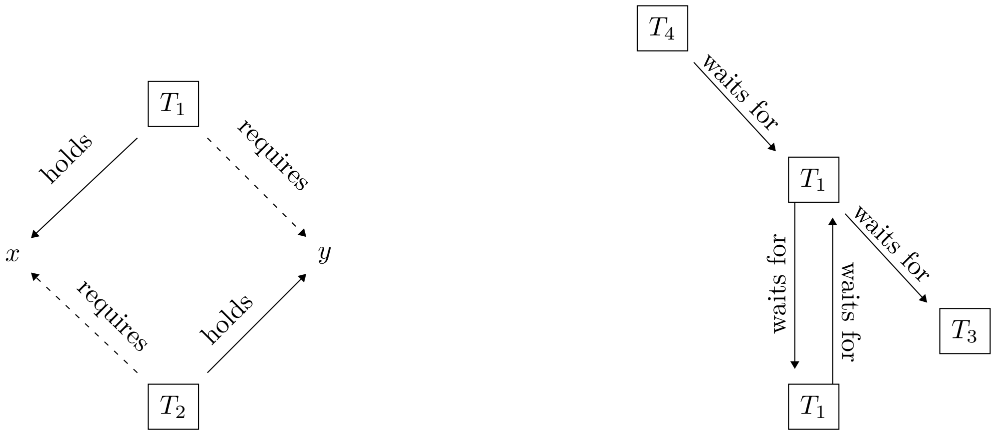
Code 1: SQL statement to set the isolation level of a transaction

3.5 Deadlocks

A **deadlock** is a situation where two or more transactions are waiting for each other to complete. The presence of a deadlock in a set of transactions can be shown via:

- **Lock graphs**: a bipartite graph in which nodes are resources or transactions and arcs represent lock assignments or requests
- **Wait-for graphs**: a directed graph in which nodes are transactions and arcs represent requests for locks

In both cases, a cycle in the graph represents a deadlock. Representation of a deadlock via a lock graph and a wait-for graph is shown respectively in Figure 7a and Figure 7b.



(a) Lock graph

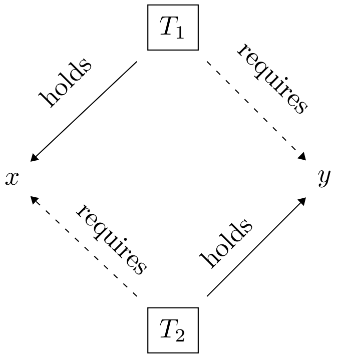
(b) Wait for graph

Figure 7: Representations of a deadlock

A deadlock can be created in the following way:

- $T_1: \ r_1(x), w_1(x), c_1$
- $T_2: \ r_2(y) w_2(x), c_2$

$\Rightarrow$ T_1 is waiting for T_2 to release the lock on x
$\Rightarrow$ a deadlock occurs



3.6 Deadlock Resolution

Many different techniques exist to solve deadlocks, such as:

- **Timeout**
- Deadlock **prevention**
- Deadlock **detection**

The deadlock resolution techniques will be discussed in more detail in the next Sections.

3.6.1 Timeout

A transaction is killed and restarted if it has been **waiting for a lock for a certain amount of time**, as it's assumed that the wait it's caused by a deadlock. Determining the maximum time interval is a difficult task, as it can vary greatly; as such, it must be determined for each system and sometimes can be altered by the database administrator.
The timeout value must be chosen carefully, keeping in mind the following tradeoff:

- an **high value** can lead to **long delays** whenever a deadlock occurs
- a **low value** can lead to **frequent and unneeded restarts** of transactions

This is the most used technique to solve deadlocks, as it's the simplest to implement and it's the one that requires the least amount of resources.

3.6.2 Deadlock Prevention

The deadlock prevention techniques work by **killing transactions** that are likely to cause a deadlock. This solution is implemented mainly in 2 ways:

- **Resource-based** prevention
  - introduces **restrictions on lock requests**
  - resources are globally sorted and must be **locked in a specific order**
  - since not all transactions know beforehand which resources they will need, **this technique is not very effective**
- **Transaction-based** prevention
  - introduces **restrictions** based on the transactions themselves
  - to each transaction an ID is assigned incrementally, **giving the transaction a priority**
    - the ID assignment can be implemented via **timestamps**
  - *"older"* transaction don't have to wait for *"younger"* transactions
  - there are two ways of determining which transaction to kill:
    - *preemptive* - the holding transaction is killed *(wound-wait)*
    - *non-preemptive* - the requesting transaction is killed *(wait-die)*
  - the problem with this technique is that **too many transactions get killed**

3.6.3 Deadlock Detection

This technique requires **controlling the contents of the lock tables** to reveal possible concurrency issues. The discovery of a deadlock requires the analysis of the *wait-for graph*, determining if it contains a cycle; to do so, the **Obermarck algorithm** is used.
Assumptions of the algorithm:

- transactions execute on a **single main node** *(one locus of control)*
- transactions may be **decomposed in sub-transactions** running on other nodes
- when a transaction spawns a sub-transaction, it **waits** for the latter to complete
- two kinds of wait-for relationships exist:
  - T_i waits for T_j to release a resource on the same node
  - a sub transaction of T_i waits for another sub transaction of T_i running on a different node

This is proven to be a **feasible solution** and it's implemented by most *DBMS*.

3.6.3.1 Obermarck Algorithm

Goal **Detection of a potential deadlock** by looking only at the local view of a node.

Method Establishment of a **communication protocol** whereby each node has a local projection of the global dependencies. Nodes exchange information and update their local graph based on the received information; communication has to be further optimized to avoid situations in which multiple nodes detect the same *(potential)* deadlock.

Node `A` **sends** its local info to a node `B` if:

- `A` **contains a transaction** T_i that is waited from another remote transaction and waits for a transaction T_j active on `B`
- $i > j$ *(the transaction with the highest ID is the oldest)*, ensuring a **message forwarding along a node path** where node `A` precedes node `B`

The algorithm runs periodically at each node and consists of 4 steps:

1. **get graph info** *(wait for dependencies among transactions and external calls)* from the previous nodes
   → sequences contain only node and top-level transaction identifiers
2. **update the local graph** by adding the received info
3. **check the existence of cycles** among transactions; if found, select one transaction in the cycle and kill it
4. **send updated graph** info to the next nodes

The algorithm contains the **arbitrary** choice of:

- **sending** the message to the **following** node ($i > j$)
- **sending** the message to the **previous** node ($i < j$)

Therefore 4 variants of the algorithm exist, each one with a different behaviour; each of them sends messages in different orders, while every one of them can detect the same deadlocks.

3.6.3.2 Distributed Deadlock Detection

To implement the aforementioned algorithm in the context of a distributed system, a distributed dependency graph is created: external call nodes represent a sub-transaction activating another sub-transaction at a different node. An illustration of such a graph is shown in Figure 8.
Description of the graph:

- Representation of the status:

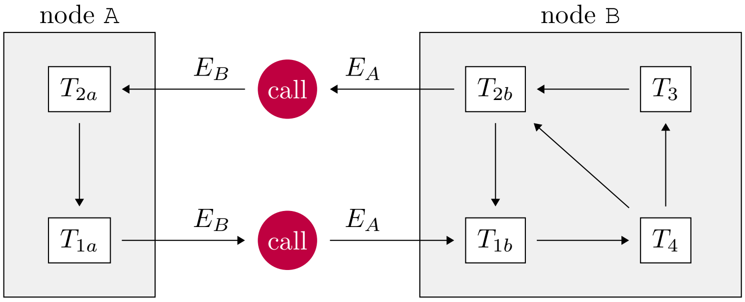


Figure 8: Distributed dependency graph

  - at node A: $E_b \to T_{2a} \to T_{1a} \to E_b$
  - at node B: $E_a \to T_{1b} \to T_{2b} \to E_a$
- E_a and E_b are external call nodes
- The symbol $\to$ represents the wait for relation between local transactions
- If one term is an external call, either the source:
  - is being waited for by a remote transaction
  - the sink waits for a remote transaction
- **Potential deadlock**: T_{2a} waits for T_{1a} (data lock) that waits for T_{1b} (call) that waits for T_{2b} *(data lock)* that waits for T_{2a}

Application of the Obermarck Algorithm to this graph:

- Node A
  - activation/wait sequence: $E_b \to T_2 \to T_1 \to E_b$
  - $i = 2, j = 1$
  - A can dispatch its local info to B
- Node B
  - activation/wait sequence: $E_a \to T_1 \to T_2 \to E_a$
  - $i = 1, j = 2$
  - B can't dispatch its local info to A

3.7 Deadlocks in practice

In real-world applications, deadlocks are not a common problem, as their probability is much lower than the conflict probability.
Consider a file with n records and 2 transactions doing each 2 accesses to their records *(assuming a uniform distribution of accesses)*. Then, the probability of:

- a **conflict** is $\displaystyle\sum_{i=1}^{n}\left(\prod_{j=1}^{n}\frac{1}{n}\right) = n \cdot \frac{1}{n} \cdot \frac{1}{n} = \mathcal{O}\left(\frac{1}{n}\right)$
- a **deadlock** is $\mathcal{O}\left(\dfrac{1}{n^2}\right)$, as it requires mutual conflicts between transactions

While deadlocks are a rare occurrence, the probability of them happening increases linearly with the number of concurrent transactions in the system but it increases quadratically with the number of data accesses each transaction performs. There are more advanced techniques to avoid deadlocks, such as update locks and hierarchical locks. Those techniques are covered in the following Sections.

3.7.1 Update lock

The most frequent deadlock occurs when 2 concurrent transactions start by reading the same resource *(SL)* and then try to write via updating their lock to `XL` *(exclusive)*. To avoid such situations, systems offer the update lock (`UL`) that allows a transaction to update *(read followed by a write)* a resource without having to acquire an exclusive lock.

Table 4 shows the escalation of the update lock.

status → request ↓	free	SL	UL	XL
SL	✓	✓	✓	✗
UL	✓	✓	✗	✗
XL	✓	✗	✗	✗

Table 4: Update lock compatibility

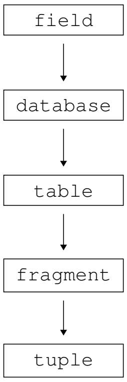
It must be noted that deadlocks are still possible in the presence of `UL`: while this technique makes them less likely, deadlock due to **update patterns** *(two transactions updating the same resource)* is still possible.
An example of this situation would be the schedule $r_1(x)r_2(x)w_1(x)w_2(x)$.

3.7.2 Hierarchical locks

To increase the granularity of the locks, the traditional lock protocol is extended via the use of hierarchical locks. This technique allows the transactions to lock items at a given level of the hierarchy; the objective is to lock the minimum amount of data possible while recognizing conflicts as soon as possible.

Locks can be specified with different granularities: database, table, fragment, tuple, or field.
This introduces a tradeoff since:

- a **coarser** granularity guarantees the probability of deadlocks is lower but the concurrency is reduced
- a **finer** granularity increases the concurrency but increases the probability of deadlocks



The technique works by:

- **Requesting** resources **top-down** until the right level is obtained

  ↓ locks are requested starting from the **root** and going down to the **leaf**
- **Releasing** the locks **bottom-up**

  ↑ locks are released starting from the **leaf** and going up to the **root**

The technique provides a richer set of primitives for lock requests; they are:

- `XL` *(exclusive lock)*: the transaction is the only one that can access the resource

  → corresponds to the write lock of the *2PL*

- `SL` *(shared lock)*: the transaction can read the resource

  → corresponds to the read lock of the *2PL*
- `ISL` *(intent shared lock)*: expresses the intention of locking in a shared manner one of the nodes that descend from the current node
- `IXL` *(intent exclusive lock)*: expresses the intention of locking exclusively one of the nodes that descend from the current node
- `SIXL` *(shared intent exclusive lock)*: expresses the intention of locking in a shared manner one of the nodes that descend from the current node and in an exclusive manner one of the nodes that descend from the current node

In order to:

- request a `SL` or a `ISL` on a node, a transaction must already hold an `ISL` or `IXL` lock on the parent node
- request a `IXL`, `XL` or `SIXL` on a node, a transaction must already hold a `SIXL` or `IXL` lock on the parent node

The rules of compatibility used by the lock manager to decide whether to accept the lock are shown in Table 5

`resource →`
`request ↓`
	`free`	`ISL`	`IXL`	`SL`	`SIXL`	`XL`

`ISL`	✓	✓	✓	✓	✓	✗
`IXL`	✓	✓	✓	✗	✗	✗
`SL`	✓	✓	✗	✓	✗	✗
`SIXL`	✓	✓	✗	✗	✗	✗
`XL`	✓	✗	✗	✗	✗	✗

Table 5: Hierarchical lock compatibility

3.8 Concurrency Control based on timestamps

The **timestamp based** concurrency control *(or TS)* makes use of timestamps or identifiers that define a total ordering of temporal events within a system. In centralized systems, the timestamp is generated by reading the value of the system clock at the time at which the event occurs.
The concurrency control with timestamps works as follows:

- every transaction is assigned a **timestamp** at the beginning of its execution
- a schedule is **accepted** if and only if it reflects the serial ordering of the transaction based on the value of the timestamp of each transaction

This method is the easiest to implement, but it's less efficient than then *2PL*; compared to the former *(which is described as pessimistic)*, **this method is optimistic** as it does not assume that collisions will arise.

3.8.1 Timestamps in distributed systems

Concurrency control in a distributed environment causes theoretical difficulties because the system clock is not synchronized among the different nodes: the timestamp is not an indicator of global time. To solve this issue, a system function must give out timestamps on requests
This technique is called **Lamport Method**. Using this method, a timestamp is a number characterized by two groups of digits:

- the **least significant digits** represent the **node** at which the event occurs

- the **most significant digits** represent the **time** at which the event occurs

The syntax of a timestamp is then `timestamp = <event-id>.<node-id>`

The most significant digits can be obtained through a local counter, incremented at each event; in this way, each event is assigned a unique timestamp. Each time two nodes communicate *(via a message exchange)* the timestamp becomes synchronized: given that the sending event precedes the receiving event, the timestamp of the latter must be greater than the timestamp of the former. Otherwise, the timestamp of the receiving event is *"bumped"* to the value of the timestamp of the sending event plus one.

Each object `x` has two indicators, `RTM(x)` and `WTM(x)`, which represent the timestamp of the last transaction that has respectively read or written the resource `x`.
The scheduler receives requests from access to objects of the type `read(x, ts)` or `write(x, ts)`, where `ts` is the timestamp of the transaction that requests the access; the scheduler accepts or rejects the requests according to the following policies:

- `read(x, ts)`
  - ✗ the request is **rejected** if `ts < WTM(x)`; the transaction is killed
  - ✓ the request is **accepted** if `ts ≥ WTM(x)`; the transaction is allowed to read the object and the value of `RTM(x)` is set equal to the greater between `ts` and `RTM(x)`
- `write(x, ts)`
  - ✗ the request is **rejected** if `ts < WTM(x)` or `ts < RTM(x)`; the transaction is killed
  - ✓ the request is **accepted** if `ts ≥ WTM(x)` and `ts ≥ RTM(x)`; the transaction is allowed to write the object and the value of `WTM(x)` is set equal to `ts`

Basic *TS*-based control considers only committed transactions in the schedule, while aborted transactions are not considered *(commit-projection hypothesis)*; however, if aborts occur, dirty reads may happen.
To cope with dirty reads, a variant of basic *TS* has been introduced: a transaction T_i that issues a $r_{\texttt{ts}}(x)$ or a $w_{\texttt{ts}}(x)$ such that `ts > WTM(X)` is delayed until the transaction T_j that has written the object x commits or aborts.
This method works similarly to long-duration write locks, but buffering operations might introduce big delays.

3.8.2 Thomas Rule

In order to reduce the kill rate, the *Thomas Rule* has been introduced. It makes a slight change to the `write(x, ts)` request:

- if `ts < RTM(x)` the request is **rejected** and the transaction is **killed**
- else, if `ts < WTM(x)` the write request is **obsolete** and the transaction is **skipped**
- else, access is **granted** and the value of `WTM(x)` is set equal to `ts`

This rule then allows the scheduler to skip a `WRITE` on an object that has already been written by a younger transaction, without killing it. It works only if the transaction issues a write without requiring a previous read on the object; instructions like `SET X = X + 1` would fail.

3.9 Comparison between *VSR*, *CSR*, *2PL* and *TS*

Figure 9 illustrates the taxonomy of the methods *VSR*, *CSR*, *2PL* and *TS*:

- *VSR* is the most general class, strictly including *CSR*
- *CSR* strictly includes both *2PL* and *TS*
- *2PL* and *TS* are neither mutually exclusive nor mutually inclusive, as they share some schedules that are accepted by both methods but other schedules that are accepted by one method but not by the other exists

The relation between *TS* and *2PL* can be summarized by 3 schedules:

- $r_1(x)w_1(x)r_2(x)w_2(x)r_0(y)w_1(y)$ is accepted by *TS* but not by *2PL*
- $r_2(x)w_2(x)r_1(x)w_1(x)$ is accepted by *2PL* but not by *TS*
- $r_1(x)w_1(x)r_2(x)w_2(x)$ is accepted by both *TS* and *2PL*

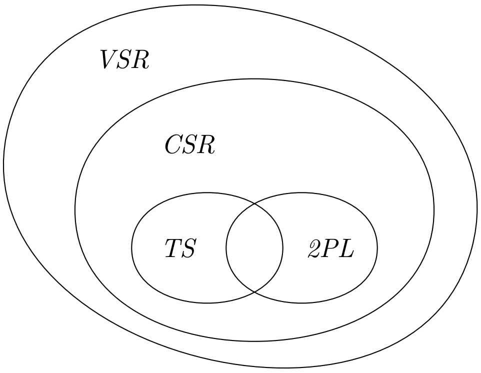


Figure 9: Taxonomy of concurrency control methods

	2PL	*TS*
transactions	can be actively waiting	killed and restarted
serialization	imposed by conflicts	imposed by timestamps
delays	caused by commit wait in strict *2PL*	none
deadlocks	can be caused	prevented via the wound-wait and wait-die methods

Table 6: Differences between *2PL* and *TS*

3.9.1 *TS* implies *CSR*

The objective of this Section is to illustrate the relation between *TS* and *CSR*. To do so, consider the following:

- Let S be a **schedule** accepted by *TS* and containing two **transactions** T_1 and T_2
- Suppose that S is not accepted by *CSR*, which implies that there is a cycle between T_1 and T_2
  - S contains $\mathrm{OP}_1(x)$ and $\mathrm{OP}_2(x)$ where at least one of the two is a `WRITE`
  - S also contains $\mathrm{OP}_2(y)$ and $\mathrm{OP}_1(y)$ where at least one of the two is a `WRITE`
- When $\mathrm{OP}_1(y)$ arrives, the following statements are true:
  - if $\mathrm{OP}_1(y)$ is a `READ`, then T_1 is killed by *TS* because it tries to read a value written by a younger transaction. This is a contradiction.
  - If $\mathrm{OP}_1(y)$ is a `WRITE`, then T_1 is killed by *TS* because it tries to write a value written by a younger transaction. This is a contradiction.

3.9.2 *2PL* and *TS*

The main differences between *2PL* and *TS* are shown in Table 6.
However, normally restarting a transaction costs more than waiting; for this reason, *2PL* is normally preferred. Commercial systems implement a mix of optimistic and pessimistic concurrency control.

3.10 Multi version Concurrency Control

Idea: while write operations generate a new version, read operations access the latest *(right)* version.

In this technique, write operations generate new copies, each one with a new `WTM`. Each object `x` always has $n \geq 1$ active versions, each of them having a timestamp $\texttt{WTM}_\texttt{i}$ associated with it, while there is only one global timestamp `RTM`.
Old versions of the object are kept in the database until there are no transactions that need their values.

Mechanism of `READ` and `WRITE` operations:

- $r_{\texttt{ts}}(x)$ is always accepted. A copy x_k is selected for reading such that:
  - if $\texttt{ts} > \texttt{WTM}_N(x)$, then $k = N$
  - otherwise, k is taken such that $\texttt{WTM}_k(x) \leq \texttt{ts} < \texttt{WTM}_{k+1}(x)$
- $w_{\texttt{ts}}(x)$:
  - if $ts < \texttt{RTM}(x)$ then the request is **rejected**
  - otherwise, a new version is created *(N is incremented)* with $\texttt{WTM}_N(x) = \texttt{ts}$

3.10.1 Snapshot Isolation - *SI*

The realization of multi-*TS* allows introducing into *DBMS* another level of isolation, called **snapshot isolation** *(or SI).*
At this level:

- no `RTM` is used on the objects *(the `WTM` is the only timestamp)*
- every transaction reads the version consistent with its timestamp and defers `WRITE` operations to the end
  → the read version is the one that existed when the transaction started *(the snapshot)*
- If the scheduler detects that the writes of a transaction conflict with writes of other current transactions after the snapshot timestamps, it aborts
  → this is a form of optimistic concurrency control

However, this method introduces a **write skew anomaly**, since:

- *SI* does not guarantee serializability
- an execution under *SI* in which the two transactions T_1 and T_2 are executed in parallel is not equivalent to an execution in which T_1 is executed before T_2

4 Triggers

Triggers are a set of actions that are executed automatically by the *DBMS* when an `INSERT`, `UPDATE`, or `DELETE` operation is performed on a specific table. They are performed in addition to integrity constraints, allowing the *DBMS* to perform supplementary, more complex, checks; they are stored in the *DBMS* similarly to stored procedures.
Triggers define the **Event-Condition-Action** *(ECA)* paradigm:

- whenever **event** E occurs
  → normally a modification of the database status: `INSERT`, `UPDATE` or `DELETE`
- if a **condition** C is true
  - a predicate that identifies those situations in which the execution of the trigger's action is required
- then **action** A is executed
  → a generic update statement in or a stored procedure
  → usually composed of database updates *(`INSERT, UPDATE, DELETE`)*
  → can include also error notifications

It's important to note that all triggers **interact with referential integrity constraints**.

Triggers add powerful data management capabilities in a transparent and reusable manner: databases can be enriched with *"business and management rules"* that would otherwise be distributed over all applications. Understanding the interactions between triggers and transactions is however rather complex.

4.1 Cascading and recursive cascading

The action of a trigger may cause another trigger to be executed. Let `T1` and `T2` be two triggers on the same table `t` and let `T1` be executed before `T2`. Then:

- **cascading** is the condition in which the action of `T1` triggers `T2` *(also called nesting)*
- **recursive cascading** is the condition in which the action of `T1` triggers `T1` again *(also called looping)*

The termination condition ensures that, for any initial state and any sequence of modifications, a final state is always produced and infinite activation cycles are not possible.

When multiple triggers are activated by the same event the *DBMS* establishes a precedence criterion to decide which one to execute first; typically the delayed triggers will be executed at a later moment.

Termination analysis The termination analysis is the process of verifying that a trigger does not cause undesired effects. The simplest way to achieve this goal is by analysing the triggering graph:

- A **node** i is associated with each **trigger** $\mathtt{T}_i$
- An **arc** from a node i to a node j exists if the **execution** of trigger $\mathtt{T}_i$ causes the execution of trigger $\mathtt{T}_j$

Such a graph is built via a simple syntactical analysis of the trigger's action. If the graph is acyclic, then the system is guaranteed to terminate; otherwise, triggers may terminate or not.
As such, the **acyclicity condition is sufficient for termination**, but not necessary.

4.2 Triggers in SQL

```
CREATE TRIGGER <trigger_name>
  [BEFORE | AFTER] -- EXECUTION MODE
  [INSERT | UPDATE | DELETE [ OF <column> ]] ON <table_name> -- EVENT
  REFERENCING {
    [OLD TABLE [AS] <old_table_alias>] [NEW TABLE [AS] <new_table_alias>] |
    [OLD ROW [AS] <old_row_alias>] [NEW ROW [AS] <new_row_alias>]
  }
  [FOR EACH [ROW | STATEMENT] ]
  [WHEN <condition>] -- CONDITION
  <trigger_action> -- ACTION
```

Code 2: Trigger Syntax

Multiple triggers on the same event `SQL 1999` specifies the ordered execution of multiple triggers on the same event; if multiple triggers on the same level exist, then the order of execution is implementation-dependent (normally either alphabetically ordered or in the order of creation). The execution sequence is:

1. **`before`** statement-level triggers
2. **`before`** row-level triggers
3. modification is applied and integrity checks are performed
4. **`after`** row-level triggers
5. **`after`** statement-level triggers

4.2.1 Execution mode

Before The action of the trigger is executed **before the database status changes** *(if the conditions hold)*; the trigger is used to check the validity of the operation before it takes place, possibly conditioning its effects.

Safeness constraint Before triggers cannot update the database directly but can affect the transition variables in row-level granularity.

After The action of the trigger is executed **after the database status change**s *(if the conditions hold)*; it is used to update the database in response to the operation, and it's the most common execution mode.

4.2.2 Granularity of Events

Row level The trigger is considered and possibly executed **once for each tuple affected** by the activating statement. Writing row-level triggers is simpler, but the triggers are potentially less efficient.

Statement level The trigger is considered and possibly executed only **once for each activating statement**, independently of the number of affected tuples in the target table *(even if the statement does not affect any tuple)*. This is close to the traditional approach of `SQL` statements, which are normally set-oriented.

4.2.3 Transition Variables

Special variables denoting the before and after the state of the database are available for the trigger action; their syntax depends on the granularity of the event.

- **row level**: tuple variables `old` and `new` represent the value of the tuple before and **after the update**
- **statement level**: **table variables** `old` and `new` represent the set of tuples **before** and **after the update**

Furthermore:

- variables `old` and `old table` are **undefined** in triggers whose event is **`insert`**
- variables `new` and `new table` are **undefined** in triggers whose event is **`delete`**

4.3 Materialized Views

A **view** is a virtual table defined with a query stored in the database catalogue and then used in queries as if it were a total table. When a view is mentioned in a `SELECT` query, the query processor of the *DBMS* rewrites the query via the view definition: the executed query only uses the base tables to the view.
When the queries to a view are more frequent than the updates on the base tables that change the view content, then the view can be materialized: the results of the query that defines the view are stored in a table, and the view is defined as a `SELECT` query on that table.
Some systems support the `CREATE MATERIALIZED VIEW` command, which makes the view automatically materialized by the *DBMS*; an alternative is to implement the materialization via triggers.

Incremental View Maintenance Incremental View Maintenance is the process of updating a materialized view when the base tables are updated.
Suppose that view `V` is defined by a query `Q` over a set of base relations `D`. When `D` changes to:

```
D′ := D + dD
```

its corresponding new state is:

```
V′ := Q(D′) = Q(D) + dQ
```

The *DBMS* computes the new view state over the increment of the base relations.

4.4 Design Principles

- Use triggers to **guarantee** that when a specific operation is performed, all the related actions are performed
- Do not define triggers that **duplicate features** already built into the *DBMS*

  → does not define triggers that reject bad data if the same can be done with `SQL CHECK` constraints
- Limit the **size** of the trigger action

  → if the logic of a trigger requires more than 60 lines of code, then it is probably better to implement it as a stored procedure and call it from the trigger
- Use triggers only for **centralized, global operations** that should be fired for the triggering statement, regardless of which user or database application issues the statement
- Avoid recursive triggers if not strictly necessary

  → recursive triggers may cause infinite loops, thus preventing the *DBMS* from terminating
- Use triggers **judiciously**, as they are executed for every user every time the event occurs on which they are defined

5 Physical Databases

A simple illustration of the **Physical Database** is shown in Figure 10.

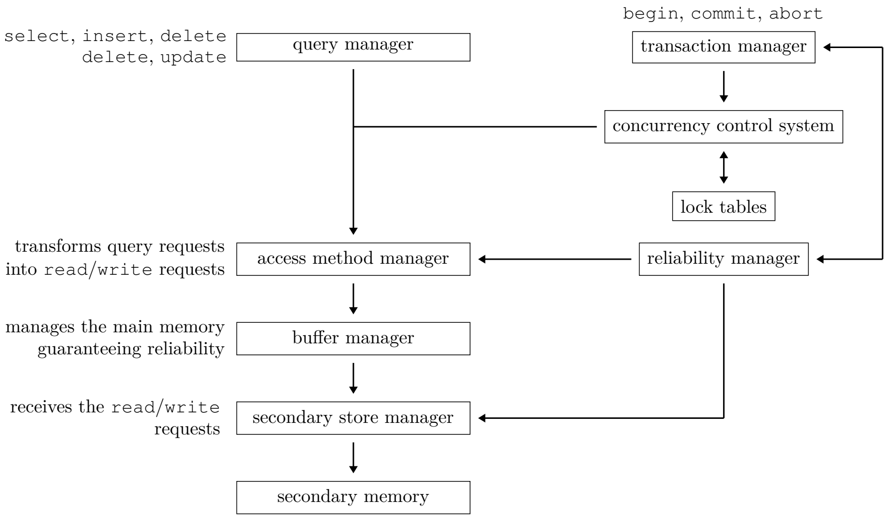


Figure 10: Physical database structure

Databases data is stored between a **main** and a **secondary** memory, due to size and persistence constraints; data stored in secondary memory can only be used if it is loaded into the main memory via an **I/O operation**. The units of storage are called:

- **Block**, if in the secondary memory
  - normally, blocks have a fixed size *(≈ 4 KB)*
- **Page**, if in the main memory
  - normally, pages have the same size of blocks
  - in some systems, a page can contain multiple blocks

The duration of an I/O operation varies greatly, depending on the technology used to store data: *SSD* are faster, yet more expensive, than *HDD*; for such reasons, large datasets are usually stored on *HDD* and small datasets on *SSD*.

Mechanical Hard Drive A mechanical hard drive is a device that stores data on a rotating platter, which is divided into tracks and sectors; an illustration of an HDD is shown in Figure 11.

- Several **disks** are **piled** and **rotating** at a constant angular speed around a central axis *(the spindle)*
- A **head stack** mounted onto an arm moves radially to reach the tracks at various distances from the centre
- A particular **sector** is reached by waiting for it to pass under one of the heads
- Several **blocks** can be reached at the same time *(one per head)*

Access Time The access time of the secondary memory can be measured as the sum of:

- **seek** time $8 - 12\,ms$, head positioning time on the correct track
- **latency** time $2 - 8\,ms$, disc rotation on the correct sector
- **transfer** time $0.1 - 1\,ms$, data transfer from the disk to the buffer

The cost of access to secondary memory is about 4 orders of magnitude (10^4) higher than the one of access to main memory. In I/O bound applications, the cost exclusively depends on the number of accesses to secondary memory.
The cost of a query is closely related to the number of numbers that are read (moved) from secondary memory.

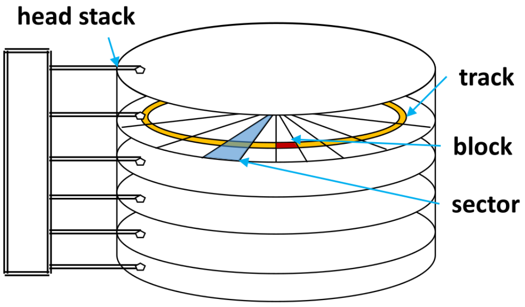


Figure 11: Structure of a mechanical hard drive

***DBMS* and file systems** The File System *(FS)* is the layer of the operating system that manages the secondary memory: *DBMS* make limited use of FS functionalities *(such as creating or deleting files, reading or writing blocks, etc.)*, since they manage directly the file organization, both in terms of the distribution of data within the blocks and with respect to the internal structures of the blocks. Furthermore, a *DBMS* may also control the physical allocation of blocks onto the disk, to optimize the access time or increase the reliability.

Data dictionary Each relational *DBMS* maintains a **data dictionary** that contains information about the database structure and the database itself. The data dictionary is a centralized repository of information about data such as meaning, relationships to other data, origin, usage and format.

5.1 Blocks and tuples

The files can be split into their physical and logical components:

- **blocks**: the physical components
- **tuples**: the logical components

While the size of a block is typically fixed, depending on the file system and on how the disk is formatted, the size of a tuple depends on the database design and is typically variable within a file.

Organization of tuples in blocks A block for sequential and hash-based methods is divided in:

- **block header** and **trailer** with control information used by the File System
- **page header** and **trailer** with control information about the access method
- a **page dictionary** containing pointers to each elementary item of useful data contained in the page
- a **useful part** containing the data
    - normally, page dictionaries and useful data grow as a stack in opposite directions
- a **checksum** to verify the integrity of the block

An illustration of a block is shown in Figure 12.

The block factor B represents the number of tuples within a block. It's fundamental to estimate the cost of queries:

- the average **size** of a **record** *(or tuple)* is called **SR**
- the average **size** of a **block** is called **SB**

The block factor is defined as:

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

The rest of the space can be:

- **used** if the records are **spanned** between block
- **not used**, if the records are **unspanned**

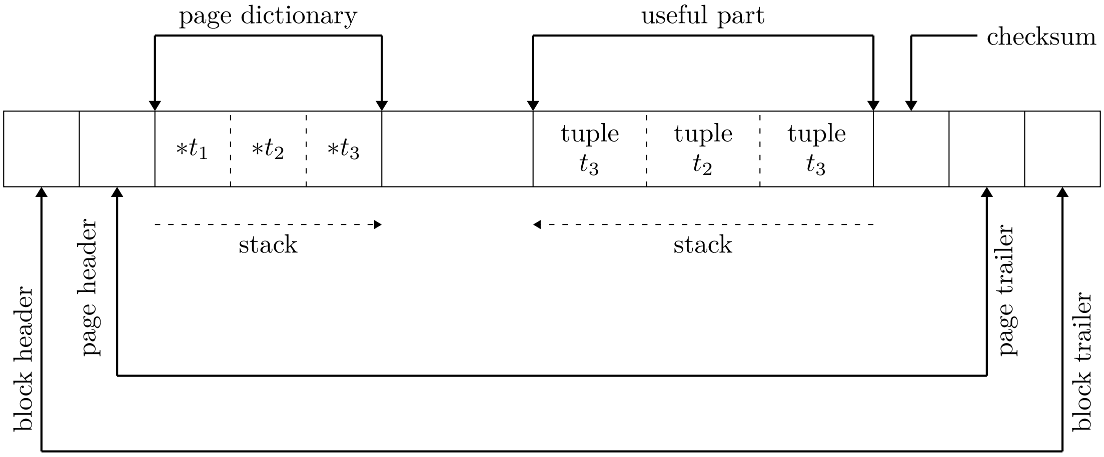


Figure 12: Structure of a block

Buffer Operations Operations are performed in the main memory and affect the pages: in the cost model, it's always assumed that all the pages have equal size and are organized in blocks.
Operations are:

- **insertion** an update of a tuple
  - may require a reorganization of the page or the usage of a new page
  - an update to a file may require a reorganization of the page
- **deletion** of a tuple
  - often carried out by marking the tuple as invalid and triggering later a reorganization of the page
- **access** to a field of a tuple
  - identified by a pointer to the beginning of the tuple and the length of the field itself

5.2 Indexes

Indexes are data structures that help efficiently retrieve tuples according to certain criteria, called **search key** (or *SK*); index entries are sorted with regard to the search key.
They contain records in the form `<search key pointer to block>`.
It's important to highlight the difference between **primary key** and **search key**: the former is the key used to uniquely identify a tuple, while the latter is the key used to retrieve tuples according to certain criteria.
Furthermore, the search key defines a **search domain**: each tuple is associated with one or more pointers; each of them may not be unique.
A comparison of the two entities is shown in Table 7.

primary key	*search key*
does not imply access path	defines a common access path
defines a constraint	defines a common access path
unique	not necessarily unique
implemented by an index	-

Table 7: Comparison between primary key and search key

Indexes are smaller than primary data structures, and they can be loaded in a file in the main memory; they support point queries, range queries, and sorted scans efficiently. However, adding indexes to tables means that the *DBMS* has to update each index after an insert, update, or delete operation: this operation may be costly and may slow it down.

Indexes can be categorized into **sparse** and **dense**: while the former associate a search key to a single tuple, the latter associate a search key to a block of tuples.

Sparse indexes

- an entry index is associated with **every** search-key value in the file
- **high performances**: to access a tuple, the index is accessed first, and then the block is accessed
- can be used on **entry-sequenced** primary structures

Dense indexes

- an entry index is associated only with **some** search-key values in the file
- requires **less space**, but it's generally slower in locating the tuple
- requires **sequentially ordered** primary structures
  → the search key corresponds to the ordering key of the tuple
- **low performances**: to access a tuple, it's necessary to scan each block until the search key is found; then the block must be scanned to locate the tuple itself
- good **tradeoff** between memory and performance as only one index entry is necessary for each block in the file

5.2.1 Indexing techniques

The index can be further classified according to the indexing used, creating the following indexes: **primary**, **secondary**, and **clustering**.

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:

$$\begin{aligned} \text{search key} &= \text{ordering key} \\ \text{SK} &= \text{OK} \end{aligned}$$

- Only one primary index can be defined for each table
  - usually the primary index corresponds to the primary key of the table
- The index can be either **sparse** or **dense**

Secondary index A secondary index specifies an order different from the sequential order of the file:

$$\begin{aligned} \text{search key} &\neq \text{ordering key} \\ \text{SK} &\neq \text{OK} \end{aligned}$$

- Multiple secondary index structures can be defined for the same table on multiple search keys
- It is always **dense** because tuples with contiguous values of the key can be distributed in different blocks

Clustering index A clustering index is a generalization of the primary index in which the ordering key *(equal to the search key)* can not be unique.

- The pointer of key x refers to the block containing the first tuple with key x
- The index can be either **sparse** or **dense**, but it is usually **sparse**

5.2.2 Indexes in SQL

To create an SQL index, every table should have:

- a suitable **primary storage**, possibly sequentially ordered *(normally on unique values)*
- several **secondary indexes**, both unique and non, 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 **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 that is heavily used as a **secondary key**
4. add secondary index structures on columns that are 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 syntax for creating and deleting an index is shown in Code 3.

```
-- creation
CREATE [UNIQUE] INDEX <index_name> on <table_name>[(<column_name>, ...)]
-- deletion
DROP INDEX <index_name>
```

Code 3: SQL syntax for creating and deleting an index

5.3 Physical Access Structures

Each *DBMS* has a distinctive and limited set of access methods and software modules that provide data access and manipulation (STORE and RETRIEVE) primitives for each physical data structure.
Access methods have their own data structures to organize data:

- Each table is stored in exactly **one primal** physical data structure
- Each table may have **zero or more secondary** access structures

Structures are divided in two **categories**:

- **Primary**: contains all the tuples of a table; its main purpose is store the table content
- **Secondary**: used to index primary structures, and only contains the values of some fields, interleaved with pointers to the block of the primary structure; its main purpose is to speed up the search for specific tuples, according to some criteria

To access each type of structure, the *DBMS* uses a different set of access methods:

- **Sequential** access methods
- **Hash-based** access methods
- **Tree-based** access methods

Not all types of structures are equally suited for implementing the primary or secondary access methods, as shown in Table 8.

	primary	*secondary*
sequential	✓ (typical)	✗
hash-based	✓ (in some *DBMS*)	✓
tree-based	✗ (obsolete)	✓ (typical)

Table 8: Access structures

5.3.1 Sequential structures

Sequential structures arrange tuples in a sequence in the secondary memory; blocks can be contiguous on disk or sparse.
Two types of sequential structures are:

- **entry-sequenced** organization: the sequence of tuples is dictated by their **entry order**
- **sequentially-ordered** organization: the sequence of tuples is dictated by the **value of one or more keys**

Entry-sequenced organization Entry-sequenced organization *(also called heap)* is the simplest and most common type of sequential organization.

- **Efficient** for:
  - ✓ insertion, which does not require shifting
  - ✓ space occupancy, as it uses all the blocks available for data and all the space within a block
  - ✓ sequential reading and writing, especially if the blocks are contiguous
  - ✓ query like `SELECT * FROM table`
- **Inefficient** for:
  - ✗ searching specific data units, as it may require scanning the whole structure
    - → this issue can be solved via indexes
  - ✗ updates that increase the size of a tuple, as it may require shifting and writing on another block
    - → this issue can be solved by deleting old versions of the tuple and inserting new ones
  - ✗ query like `SELECT * FROM table WHERE ...`

Sequentially-ordered sequential structure Tuples are sorted according to the value of a key field.

- Efficient for:
  - ✓ range queries that retrieve tuples having the key in an interval
  - ✓ `order by` and `group by` queries exploiting the key
- Inefficient for:
  - ✗ reordering tuples within a block, if there is even enough space

To avoid the global reordering problem, the following techniques can be used:

- **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

5.3.1.1 Comparison of sequential structures

Table 9 shows the main differences between the main structures.
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`	✔	✘
`UPDATE`	✔	✘
`DELETE`	✘	✘
`SELECT * FROM T WHERE [...]`	✘	✔

Table 9: Comparison of sequential structures

5.3.2 Hash-based structures

Hash-based structures provide efficient associative access to data, based on the value of a key field.
A hash-based structure has N_B buckets, with $N_B \ll \#$ of data items: a bucket is a unit of storage, typically of the size of 1 block; often they are stored adjacently in the file. A hash function maps the key field to a value between 0 and $N_B - 1$; this value is interpreted as the index of a bucket in the hash structure (via a hash table).

- **Efficient** for:
  - ✓ tables with small size and almost static content
  - ✓ queries with equality predicates on the key field *(point queries)*
- **Inefficient** for:
  - × tables with large size and dynamic content
  - × queries with inequality predicates on the key field *(range queries)*

Implementation The implementation consists of two steps.

step 1. **folding**: transforms the key values to positive integers, uniformly distributed in a big range
step 2. **hashing**: transforms the positive number into an index in range $[0, N_B - 1]$, to be used as the index of a bucket

Collision When two or more keys are mapped to the same bucket, a **collision** occurs; when the number of tuples in a block is greater than the number of buckets, collisions are inevitable.
Techniques to solve collisions:

- **Closed hashing** *(open addressing)*, where the collision is resolved by probing the next bucket
  - → a simple technique is linear probing, where the following buckets are tried in sequence wrapping around the end of the hash table
- **Open hashing** *(separate chaining)*, where the collision is resolved by storing the tuple in a linked list
  - → a new bucket is allocated for the same hash result linked to the previous one

Overflow chains The average length of the overflow chain is a function of:

- the **load factor** $\alpha = \dfrac{T}{B \times N_B}$, representing the average occupancy of a block
- the **block factor** $\beta = \dfrac{B}{\#\text{ of items}} = \dfrac{1}{\alpha}$, representing the average number of items per block

Where:

- T is the **number** of **tuples**
- N_B is the **number** of **buckets**
- B is the **number** of **tuples within a block**

5.3.2.1 Hash-based indexes

Hash-based structures can be used for **secondary index structures**: they are shaped and managed exactly like a hash-based primary structure, but instead of the tuples, the buckets contain key values and pointers.
In a huge database it would be very inefficient to search all the index values to reach the desired data: a good performance equality query predicates on the key field. This technique is as such inefficient for access based on interval predicates or the value of non search-key attributes.

5.3.3 Tree-based structures

The tree-based structures are the most frequently used in relational *DBMS* for secondary index structures; for instance, `SQL` structures are implemented this way. They support associative access based on the value of a key search field.
A commonly used tree-based structure is the **B-tree**.

B-trees Balanced trees *(or B-trees)* are a generalization binary search trees; the length of all the possible paths from the root node to the leaves is the same, while a node can have any number of children in a predefined range. This constraint improves the performance of the database.
There are two types of B-trees:

1. **B trees**: the key values are stored in **leaf nodes and internal nodes**
2. **B+ trees**: the key values are stored in **leaf nodes** only

5.3.3.1 B+ trees

The **B+ tree** represents an evolution from the B-tree. Each node is stored in a block, and the key values are stored in the leaf nodes only; since the majority of the nodes are leaf nodes, the B+ tree is more efficient than the B-tree.
The fan-out depends on the size of the block, the size of the key and the size of the pointer.

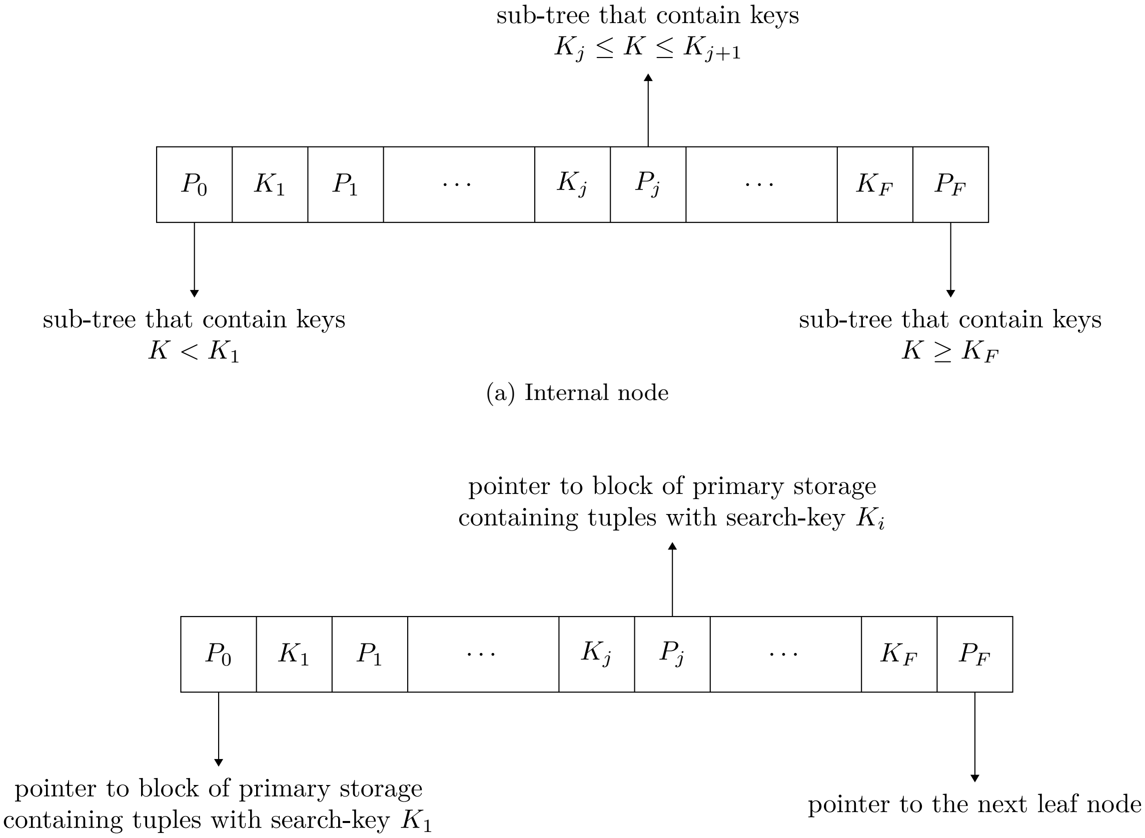
Structure of an internal node The structure of an internal node is shown in Figure 13a. Each node contains F keys, sorted lexicographically, and $F+1$ pointers to the child nodes. Each key $K_j,\ 1 \leq j \leq F$ is followed by a pointer P_j, while K_1 is preceded by a pointer P_0. Each pointer addresses a sub-tree:

- P_0 points to the subtree containing all the keys less than K_1
- P_F points to the subtree containing all the keys greater or equal to K_F
- each intermediate pointer addresses a sub-tree that contains all the information about the keys K included in 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.

Structure of a leaf node The structure of a leaf node is similar to the one of an internal node, but it contains only pointers to the data tuples or the data tuples themselves; the nodes can be structured in two ways:

1. the leaf node contains the entire tuple
    - the data structure is called **key-sequenced**
    - the position of a tuple is determined by the value of its key
    - it's simple to insert or change a tuple
2. the leaf node contains pointers to the blocks of the database that contain tuples with the same key value
    - the data structure is called **indirect**
    - the tuples can be anywhere in the file, allowing the access of a tuple allocated via any other primary mechanism
    - the structure of such a leaf node is shown in Figure 13b


sub-tree that contain keys
$K_j \leq K \leq K_{j+1}$
P_0
K_1
P_1
$\cdots$
K_j
P_j
$\cdots$
K_F
P_F
sub-tree that contain keys
$K < K_1$
sub-tree that contain keys
$K \geq K_F$
pointer to block of primary storage
containing tuples with search-key K_i
P_0
K_1
P_1
$\cdots$
K_j
P_j
$\cdots$
K_F
P_F
pointer to block of primary storage
containing tuples with search-key K_1
pointer to the next leaf node

(a) Internal node

(b) Leaf node

Figure 13: Structure of the nodes of a B+ tree

5.3.3.2 Search Mechanism

The search mechanism consists of the 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 case of a key-sequenced leaf node, the search is completed as the tuple is found
- in case of an indirect leaf node, it's necessary to access the memory block pointed by the pointer $P_j,\ 0 \leq j \leq F$

5.3.3.3 Difference between B and B+ trees

In B+ trees, leaf nodes are linked by a chain of pointers, connecting them in the order imposed by the key values. This chain allows efficient execution of range queries, as it's possible to access the lower bound of the interval via a normal search and then scan sequentially the leaves until a value greater than the upper bound is found. This data structure makes possible an ordered scan of the entire database, which is not possible in B trees.
In B trees, there is no such chain of pointers, and the only way to access the tuples in a given range is to perform a search for each tuple. In this case, intermediate nodes use two pointers for each value K_i: one of the two is used to point directly to the block containing the tuple corresponding to K_i, thus interrupting the search; the other one is used to continue the search in the subtree that includes the key values great than K_i and less than $K_i + 1$. The first pointer P_0 highlights the subtree corresponding to key values less than K_1, while the last pointer P_F highlights the subtree corresponding to key values greater than K_F. This technique saves space in the pages of the index and at the same time allows the termination of the search when a given key value is found on intermediate nodes, without having to continue the search in the subtrees.

5.4 Query Optimization

The **optimizer** is an important part of the architecture of a *DBMS*: it receives a query expressed in `SQL`, analyses the query to find mistakes, and finally generates a program in an internal format that uses the data access methods. There may be different ways to execute the same query, and the optimizer has to choose the best one.
Steps of this operation:

1. **lexical**, **syntactic** and **semantic analysis** of the query
2. translation into an **internal representation** *(similar to algebraic expressions)*
3. **algebraic** optimization
4. c**ost-based** optimization
5. **code** generation

5.4.1 Relation profiles

Each commercial *DBMS* possesses quantitative information about the relations in the database, called **relation profiles**.
The profile of a relationship contains the following information:

- the **cardinality** *(number of tuples)* $\text{CARD}(T)$ of each **table** T
- the **dimension** in bytes $\text{SIZE}(T)$ of each **tuple** in table T
- the **dimension** in bytes $\text{SIZE}(A_j, T)$ of each **attribute** A_j in **table** T
- the **number** of **distinct values** $\text{VAL}(A_j, T)$ of each **attribute** A_j in **table** T
- the **minimum** and **maximum values** $\text{MIN}(A_j, T)$ and $\text{MAX}(A_j, T)$ of each **attribute** A_j in **table** T

The relation profiles are calculated from the data stored in the tables and updated periodically via appropriate system primitives *(such as the command `UPDATE STATISTICS` in Oracle DBMS)*.

Cost-based optimization requires the knowledge of an approximate value of each of the quantities contained in the relation profiles.

The probability that any row will satisfy a given predicate is called the **selectivity** of the predicate.
If VAL(A) = N and the values are uniformly distributed, the selectivity of the predicate $A = v$ is $1/N$. If no data on distribution is available, then the distribution is always assumed uniform.

5.4.2 Internal representation of queries

The representation that the optimizer creates for a query must account for the physical structure used to implement the tables, as well as the indexes that may be present; the internal representation of a query uses a tree structure, where leaves correspond to the physical data structures and whose intermediate nodes represent data access operations.
Typically, the operations that can be performed on the intermediate nodes include sequential scans, orderings, indexed accesses and joins.

Scan operation A scan operation performs sequential access to all the tuples of a table while executing operations of algebraic or extra algebraic nature, such as:

- **projection** of a subset of the attributes
- **selection** of a simple predicate of type $A_i = v$
- **ordering**, insertions, deletions, and modification of the tuples when they are accessed by the scan

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; as a consequence, if the query presents only one supported predicate, it's convenient to use the corresponding index.
When a query presents a conjunction of predicates, the *DBMS* chooses the most selective one for the primary access via index, while the other one is evaluated later in main memory via a sequential scan; on the other hand, if the query presents a disjunction of predicates, the *DBMS* is forced to do a sequential scan if any of the predicates are not supported by an index.

Join operation The `JOIN` operation is considered to be the most expensive for a *DBMS*, as there is a risk of explosion of the number of tuples in the result; to solve this problem, the optimizer uses one of three techniques:

1. **nested-loop**, based on scanning
   - a scan is performed on a table *(termed external)* and for each tuple, a scan is performed on the other table *(termed internal)*
   - the matching is efficient if there is an index on the join attribute of the internal table
2. **merge-scan**, based on ordering
   - this technique requires that both the tables are sorted on the join attribute
   - two coordinated scans run through the tuples of each table, in parallel, and the matching is performed
3. **hash-join**, based on hashing
   - this technique requires that both tables are hashed on 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 cost of an equality lookup *(such as $A = v$)* depends on the type of structure representing the table.

- **Sequential structures** with no index:
  - ✗ equality lookups are **not supported**, so the cost is the one of a full scan
  - ✗ sequentially ordered structures may have 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 of a range lookup *(such as $v_1 \leq A \leq v_2$)* depends on the type of structure representing the table.

- **Sequential structure**s *(primary)*:
  - ✗ range lookups are **not supported**, so the cost is the one of a full scan
  - ✗ sequentially ordered structures may have reduced cost
- **Hash structures** *(primary and secondary)*:
  - ✗ range lookups are **not supported**, so the cost is the one of a full scan
- **Tree structure**s *(primary and secondary)*:
  - ✓ range 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)*
- B+ **tree** structures *(primary and secondary)*:
  - ✓ range lookups are **supported** if A is the **search key attribute** of the structure

Conjunction If supported by indexes, the *DBMS* chooses the most selective supported predicate for the data access, and the other predicates are evaluated later in the main memory via a sequential scan.

Disjunction If any of the predicates are not supported by indexes, then a sequential scan is performed; otherwise, indexes can be used to evaluate all predicates and then the results are merged and duplicates are removed.

Sort Various methods allow sorting the tuples of a table optimally; however, the problem lies in the fact that the *DBMS* must load the entire loaded data in the buffer, which may be too large for the available memory. Data can be sorted in the main memory *(in which case it's performed via ad-hoc algorithms such as quicksort or merge sort)* or in the disk *(in which case it's performed via external sorting algorithms).* The latter case is picked when the data is too big to fit on the main memory; the procedure is as follows:

1. split the data in chunks of size equal to the main memory
2. load a chunk in the main memory
3. sort the data in the main memory
4. store the sorted data back on disk
5. Merged sorted chunks parts using at least three pages:
   - two for progressively loading data from two sorted chunks
   - one as output buffer to store the sorted data
6. save the result on disk
7. go to step 2 until all chunks are merged

5.4.3 Cost-based Optimization

Cost-based optimization is an optimization problem whose decisions are:

- **which data** access operations to perform
- **in which order** to perform the operations
- if there are multiple options for a given operation, **which one** to choose
- if the query requires sorting, **how to sort** the data

In order to solve the problem, the *DBMS* makes use of data profiles and approximate cost formulas. A decision tree is built, in which:

- 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)*

By assigning a cost to each plan, it's possible to find the optimal one via techniques of operations research such as branch and bound. Optimizers should be able to handle these kinds of problems in a short enough time.

5.4.4 Approaches to Query Evaluation

The query is evaluated by the *DBMS* according to two techniques:

1. **compile and store**
   - the query is compiled and stored in the *DBMS* to be executed later
   - the internal code is stored in the *DBMS* together with an indication of the dependencies of the code on the particular versions of the catalogue used at compile time
   - On relevant changes in the catalogue, the compilation of the query is invalidated and the query is recompiled
2. **compile and go**
   - the query is compiled and executed immediately, without storing the compiled code
   - although not stored, the code may live for a while in the *DBMS* to be reused by other executions

6 Ranking

The objective of **ranking** is to find the best possible result for a given query. It is formulated as a multi-objective optimization problem, in which the objective is the simultaneous optimization of multiple criteria *(such as different attributes of objects in a dataset)*. A general formulation is:

Given N **objects** described by d **attributes** and a notion of **goodness** g_i for each object i,
find the k objects that maximize the goodness.

The two main approaches are:

1. **Ranking** *(or top-k)* Queries, in which the objective is to find the k objects that maximize g_i
2. **Skyline** Queries, in which the objective is to find the objects that maximize g_i and are not dominated by any other object

6.1 Rank Aggregation

The rank Aggregation problem regards finding the best possible entity among a set of n candidates based on the ranking of m voters. Once more, two different approaches are possible:

1. **Borda's** proposal: the entity with the lowest sum of ranks is the best one
2. **Cordocet's** proposal: a candidate who defeats every other candidate in a pairwise comparison is the best one

   ✗ if preferences graph contains a cycle, the winner may not exist

Since a unique solution does not exist nor does a unique way to find it, the problem is formulated as a multi-objective optimization problem; different approaches have been defined to solve it.

Axiomatic Approach Arrow stated a list of 5 axioms that a solution to the problem must satisfy:

axiom 1. **Non-dictatorship**: the winner cannot be chosen by a single voter
axiom 2. **Universality**: the winner should represent the opinion of all voters
axiom 3. **Transitivity**: if candidate x is preferred over candidate y and candidate y is preferred over candidate z, then candidate x should be preferred over candidate z
axiom 4. **Pareto-efficiency**: for every pair (x, y) of alternatives, if x is preferred over y by every voter, then x should be preferred over y by the winner
axiom 5. **Independence of irrelevant alternatives**: if candidate x is preferred over candidate y by every voter, then the winner should be indifferent between x and y

He then promptly proved the **Arrow's Impossibility Theorem**: when more than 2 or alternatives are present, no solution satisfies all the axioms; the axioms are then inconsistent with each other and as such, they stop being axioms.

Metric Approach The metric approach is based on the idea of defining a finding a new ranking R whose total distance from the original rankings $R_1, \dots R_n$ is minimized. Multiple distances have been defined, such as:

- **Kendall Tau distance** $K(R_1, R_2)$, defined as the number of exchanges needed in a bubble sort algorithm to convert R_1 into R_2

  ✗ finding a solution is computationally hard *(NP-complete)*
- **Spearman's Footrule distance** $F(R_1, R_2)$ which adds up the distance between the ranks of the same items in the two rankings

  ✓ finding a solution is computationally easy *(polynomial)*

These two distances are however related, since:

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

Furthermore, Spearman's Footrule distance admits efficient approximations, such as the median ranking.

6.2 Combining opaque rankings - MedRank

Opaque rankings are defined as rankings that have no visible score; as such, they cannot be compared directly. To handle this problem, techniques using only the position of a candidate in the ranking have been created: one of the most famous is the MedRank method. As an output, it provides an approximation of the Footrule distance, which is then used to find the best candidate.

← **Input**: integer k, ranked lists $R_1 \dots R_N$ of N elements
→ **Output**: the top k elements
- **Procedure**:

  step 1. used sorted accesses in each list, one element at a time, until there are k elements that occur in more than $m/2$ lists
  step 2. for each element that occurs in more than $m/2$ lists, compute the median of the positions in which it occurs
  step 3. return the top k elements with the lowest median

The maximum number of sorted accesses made on each list is also called the **depth** reached by the algorithm. When the median ranks are all distinct, the algorithm provides the best possible approximation of the Footrule distance.

6.2.1 Instance Optimality

Instance Optimality is a form of optimality applied when standard optimality is unachievable.
Let A be a family of algorithms, I a family of instances, and c a cost function that maps an algorithm and an instance to a cost. Then algorithm A^* is instance optimal with regards to A and I for the cost metric c if there exists constants $k_1,\ k_2$ such that, for all $A_i \in A$ and $I_j \in I$:

$$c\left(A_i, I_j\right) \leq k_1 \cdot c\left(A^*, I_j\right) \leq k_2 \cdot c\left(A_i, I_j\right)$$

If A^* is instance-optimal, then any algorithm can improve with respect to A^* by only a constant factor, called the **optimality ratio** of A^*.
Instance optimality is a much stronger notion than standard optimality in the average or worst case since it guarantees that the algorithm is optimal on every input instance.

6.2.1.1 Optimality of MedRank

An algorithm is defined **optimal** if its execution cost is never worse than any other algorithm on any input.
MedRank is **not optimal**; however, it's **instance optimal**: among the algorithms that access the lists in sorted order, this is the best possible one *(up to a constant factor)* on every input instance.

6.3 Ranking Queries - Top-k

The Ranking Queries technique, also called Top-k, aims to retrieve the best k answers from a (potentially large) set of n answers; the notion of best depends on the context and application.
The method requires being able to order objects according to their relevance.

Naive approach Assume that S is a scoring function that assigns to each tuple t a numerical score $S(t)$.

← **Input**: cardinality k, dataset R composed by a set of tuples t, scoring function S
→ **Output**: the top k tuples in R according to S
- **Procedure**:

  step 1. for all tuples t in R, compute $S(t)$
  step 2. sort tuples according to their score
  step 3. return the first k highest-scoring tuples

This approach is expensive for large datasets, as it requires sorting a large amount of data; furthermore, if the scoring function evaluates more than one relation, the sorting must be performed on the Cartesian product of the relations *(which is even more expensive)*.

Top-k in `SQL` To perform a Top-k query in `SQL`, the *DBMS* must be capable of:

1. **ordering** tuples according to a scoring function
   - performed via the `ORDER BY` clause
2. **limiting** the output to the first k tuples
   - limiting was not native in `SQL` until 2008
   - performed via `FETCH FIRST k ROWS ONLY`, while many non standard syntax are available

Only the first k tuples become part of the result; if more than one set of k tuples satisfies the `ORDER BY` directive, then any of such sets is a valid answer and the *DBMS* is free to choose one: the semantic is **non-deterministic**.

6.3.1 Evaluation of Top-k Queries

6.3.1.1 Single relation

In the simplest case, the Top-k query is evaluated via a single relation:

- if the tuples are sorted according to the scoring function, the query can be evaluated in $O(k)$ time
  - only the first k tuples are read and returned
  - there's no need to scan the entire input
- if the tuples are not sorted and k is not too large, the query can be evaluated in $O(n \log(n))$ time
  - the entire input is scanned and sorted

6.3.1.2 Multiple relations

The Top-k query can be evaluated on multiple relations: if the relations are 2, the attribute space can be visualized as a $2D$ plane, where each tuple is represented by a point with coordinates (A_1, A_2). A representation of the Top-k query on two relations is shown in Figure 14.

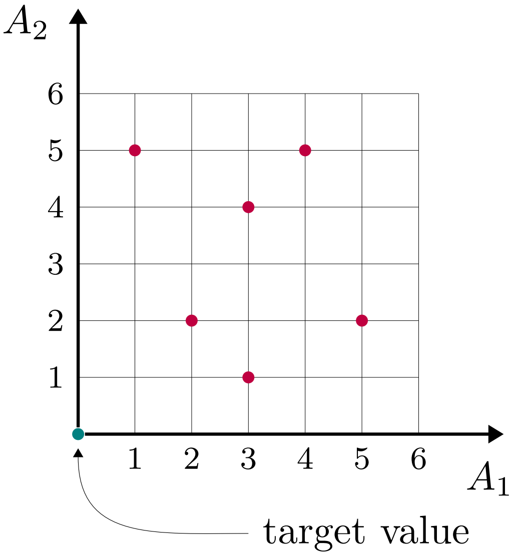
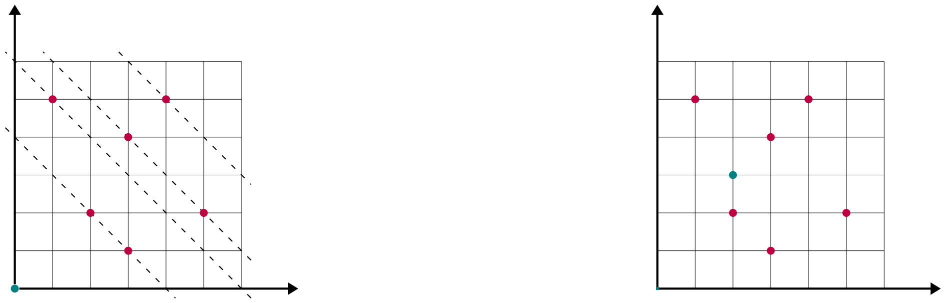


Figure 14: Top-k query on two relations

If S has form $S = k_1 \cdot A_1 + k_2 \cdot A_2$ and the objective is minimizing it, the target value lies in $q = (0, 0)$. By fixing a value of S, it's possible to calculate a slope; all the points that lie on the line are equally good answers. A representation of the weights of the Top-k query is shown in Figure 15a.
On the other hand, if the objective is getting the tuples that best match the target value q, the k best tuples are the ones that are closest to q: the distance between two tuples can be calculated in different ways. A representation of the weights of the Top-k query is shown in Figure 15b.



(a) Top-k query weights

(b) Top-k query target

For this reason, it is sometimes useful to consider distances rather than scores; therefore, the model is now a m-dimensional space of ranking attributes $A = (A_1, A_2, \dots A_m)$. A relation $R = (A_1, A_2, \dots A_m, B_1, B_2 \dots B_n)$ is a set of attributes, where $A_i,\ 1 \leq i \leq m$ are the attributes of the first tuple and $B_j,\ 1 \leq j \leq m$ are the attributes of the second tuple. The target tuple is $q = (q_1, q_2, \dots q_m)\,,\ q \in A$, while a function $d : A \times A \to \mathbb{R}$ is used to calculate the distance between two tuples in A.

Under this model, a Top-k query is transformed into a k-Nearest Neighbours *(k-NN)* query.

← **Input**: point q, relation R, integer $k \geq 1$, and a distance function d
→ **Output**: the k tuples in R that are closest to q according to d

6.3.2 Distance Functions

The most common class of distance functions is the **LP-norm** *(or Minkowski distance)*, defined as:

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

Some relevant cases:

- $p = 1 \to L_1(t, q) = \sum_{i=1}^{m} |t_i - q_i|$, **Manhattan** distance
- $p = 2 \to L_2(t, q) = \sqrt{\sum_{i=1}^{m} (t_i - q_i)^2}$, **Euclidean** distance
- $p = \infty \to L_\infty(t, q) = \max_{i=1}^{m} |t_i - q_i|$, **Chebyshev** distance

The different functions generate different shapes of the attribute space:

- Manhattan distance: the attribute space is a **hypercube**
- Euclidean distance: the attribute space is a **hypersphere**
- Chebyshev distance: the attribute space is a **hyper square**

Distance Functions with weights

The use of weights $W = (w_1, \dots, w_m)$ allows to give more importance to some attributes than others; as a consequence, the distance function is modified and the attribute space is modified as follows:

- **Manhattan** distance: $L_1(t, q, W) = \sum_{i=1}^{m} w_i\, |t_i - q_i|$, the relative space is a **hyper rhomboid**

- **Euclidean** distance: $L_2(t,q,W) = \sqrt{\sum_{i=1}^{m} w_i \left(t_i - q_i\right)^2}$, the relative space is a **hyper ellipsoid**
- **Chebyshev** distance: $L_\infty(t,q,W) = \max_{i=1}^{m} w_i \left|t_i - q_i\right|$, the relative space is a **hyper rectangle**

The elongation of the attribute space is proportional to the ratio of the weights.

6.3.3 Top-k Query in SQL

In a Top-k query, there are $n > 1$ input relations and a scoring function S defined on the result of the join, as shown in Code 4.

```
SELECT <attributes>
FROM <relation 1> R1, <relation 2> R2, ..., <relation n> Rn
WHERE <join and local condition>
ORDER BY <scoring function> DESC
FETCH FIRST k ROWS ONLY
```

Code 4: Top-k query in SQL

where the scoring function has the form $S(p_1, p_2, \ldots, p_n)$, and p_i is the weight relative to attribute A_i.
Each object o returned by the input L_j has an associated partial *(or local)* score $p_j(o) \in [0,1]$; for convenience, the scores are normalized in the range $[0,1]$, as only the worst and best possible scores of the objects are relevant.

- The **hypercube** $[0,1]^m$ is called the **score space**
- The **tuple** $p(o) = (p_1(0), p_2(0), \ldots, p_m(0)) \in [0,1]^m$ is called the **map of o into the score space** *(or score vector)*
- The **global score** $S(o)$ of o is computed via a **scoring function** S that combines the local scores of o:

$$S : [0,1]^m \to R \; S(o) = S\left(p(o)\right) = S\left(p_1(o), p_2(o), \ldots, p_m(o)\right)$$

6.3.3.1 Common sorting algorithms

The following SQL functions are commonly used to compute the global score $S(o)$:

- SUM (or AVG): weights the preferences **equally**

$$\texttt{SUM}(o) = \texttt{SUM}\left(p(o)\right) = \sum_{i=1}^{m} p_i(o)$$

$$\texttt{AVG}(o) = \texttt{AVG}\left(p(o)\right) = \frac{1}{m} \sum_{i=1}^{m} p_i(o)$$

- WSUM: weights the ranking attributes **differently**

$$\texttt{WSUM}(o) = \texttt{WSUM}\left(p(o)\right) = \sum_{i=1}^{m} w_i p_i(o)$$

- MIN: considers the **worst** partial score

$$\texttt{MIN}(o) = \texttt{MIN}\left(p(o)\right) = \min_{i=1}^{m} p_i(o)$$

- MAX: considers the **best** partial score

$$\texttt{MAX}(o) = \texttt{MAX}\left(p(o)\right) = \max_{i=1}^{m} p_i(o)$$

The objective is still finding the k objective with the highest global score.

6.3.3.2 Top-k 1-1 `JOIN` query

The Top-k 1-1 `JOIN` query is a special case of the Top-k query, where the `JOIN` operation is a 1-1 `JOIN`: all the joins are on common attributes. It's the simplest case of the Top-k query.
Assumptions that allow the use of the `SQL` Top-k 1-1 `JOIN` query:

- each input list supports **sorted access**: each access returns the id of the next best object and its partial score p_j
- each input list supports **random access**: each access returns the partial score of an object, given its id
- the **id** of an object is the **same** across all inputs
- each input consists of the **same set** of objects

6.3.3.3 Scoring Functions Model

Two different scenarios are possible:

1. an **index** is available on the **common attribute**
2. the **relation** is spread over **several sites**, each providing information only on a subset of the tuples *(middleware scenario)*

Efficient computation of `MAX` Top-k 1-1 join query When the scoring function is `MAX`, the Top-k query can be calculated efficiently via the B_0 algorithm.

← **Input**: integer $k \geq 1$, ranked lists $R_1, \ldots R_m$
→ **Output**: the k best objects according to the `MAX` scoring function
- **Procedure**:
    - step 1. Make k sorted access on each list and store objects and partial scores in a buffer B
    - step 2. For each object in B, compute the `MAX` of its available partial scores
    - step 3. Return the k objects with the highest `MAX` score

The algorithm does not need to obtain missing partial scores and does not need random access to the lists. Furthermore, B_0 is **instance-optimal**.

6.3.3.4 Fagin's Algorithm

Fagin's algorithm *(or FA)* is described as follows.

← **Input**: integer $k \geq 1$, function S combining ranked lists $R_1, \ldots R_m$
→ **Output**: the top k `<object, score>` pairs
- **Procedure**:
    - step 1. Extract the same number of objects via sorted accesses in each list until there are at least k objects in common
    - step 2. For each extracted object, compute its score by making random accesses to the lists
    - step 3. Among these, return the k objects with the highest score

The time complexity is sub-linear in the number of objects in the lists, as it's proportional to the square root of N when combining two lists: $\mathcal{O}\left(N^{m - 1/m} k^{1/m}\right)$. The stopping criterion is independent of the scoring function and the algorithm is not instance-optimal.
A visual representation of the algorithm is shown in Figure 16:

- the threshold point τ is the point with the smallest seen values on all lists in the sorted access phase
- the yellow region contains the points seen in all regions
- the grey regions contain the points seen in at least one ranking
- the blue region contains the points not seen in any ranking
- the algorithm stops when the yellow region contains at least k points

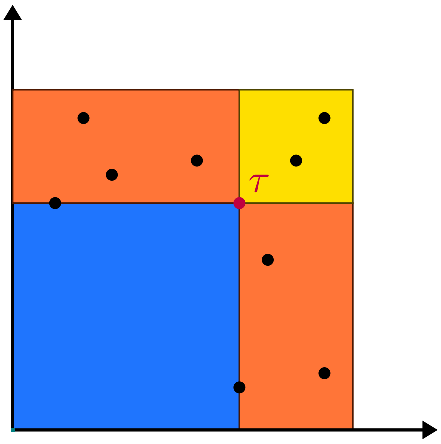


Figure 16: Visualization of Fagin's algorithm

Limits of the Fagin's Algorithm The limits of Fagin's algorithm are:

- Specific scoring functions are not exploited at all: the costs of the random and sorted accesses are not taken into account in the scoring function
- Memory requirements can become prohibitive: FA has to buffer all of the objects seen in the sorted access phase
- Small improvements are possible only by changing the algorithm itself
- Significant improvements require changing the stopping condition

6.3.3.5 Treshold Algorithm

The **Threshold Algorithm** *(or TA)* is a variant of Fagin's algorithm that exploits the scoring function to improve the performance.

← **Input**: integer $k \geq 1$, monotone function S combining ranked lists $R_1, \dots R_m$
→ **Output**: the top k `<object, score>` pairs

- **Procedure**:

```
step 1. Do a sorted access in parallel in each list $R_i$
step 2. For each object $o$, do random accesses in the other lists $R_j$, thus obtaining the score $s_j$
step 3. Compute the overall score $S(s_1, \dots s_m)$; if the value is among the $k$ highest, store it
step 4. Let $s_{li}$ be the minimum score (the last seen under sorted access) of ranked list $R_i$
step 5. Define the threshold $T = S(s_{l1}, \dots s_{lm})$
step 6. If the score of the $k$-th object is less than $T$, go back to step (1)
step 7. Return the current $k$ objects with the highest score
```

The Threshold Algorithm is instance optimal among all algorithms that use random and sorted accesses: the stopping criterion depends on the scoring function.
A visual representation of the algorithm is shown in Figure 17:

- the threshold point τ is the point with the smallest seen values on all lists in the sorted access phase
- the dashed red line separates the region of points with a higher score than τ from the rest
- the yellow region contains the points seen in all regions
- the blue region contains the points not seen in any ranking
- the algorithm stops when the yellow region contains at least k points with a score higher than τ

Generally speaking, the Threshold algorithm is more efficient than Fagin's algorithm, since it can adapt to the specific scoring function. To characterize the performance of the Threshold Algorithm, it's necessary to first define the **middleware cost**:

$$\text{cost} = S_A \cdot c_{S_A} + S_R \cdot c_{S_R}$$

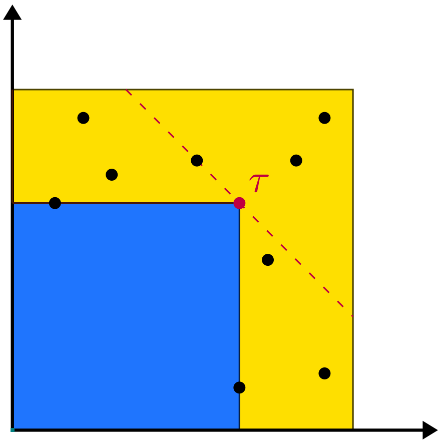


Figure 17: Visualization of the Threshold algorithm

where:

- S_A is the number of sorted accesses
- S_R is the number of random accesses
- c_{S_A} is the cost of a sorted access
- c_{S_R} is the cost of a random access

In an ideal scenario, $c_{S_A} = c_{S_R} = 1$, thus the cost is equal to the number of accesses; in other cases, the base cost may differ:

- for web sources, usually $c_{R_A} \gg c_{S_R}$
    - limit case: $c_{R_A} = \infty$, the random access is impossible
- some sources might not be accessible through sorted access
    - happens when there's no index to process an attribute p_j
    - $c_{S_A} = \infty$

6.3.3.6 No Random Access Algorithm

The **No Random Access Algorithm** *(or NORA)* is a variant of the Threshold Algorithm that does not use random accesses; it still returns the Top-k results, but the scores might be uncertain.
The main idea of the algorithm is to maintain a lower bound $S^-(o)$ and an upper bound $S^+(o)$ of the score for each retrieved object o; to achieve such an objective, a buffer B with unlimited capacity is used and it's kept sorted according to decreasing lower bounds $S^-(o)$. The algorithm terminates when no new object $\overline{o}$ can do better than any of the objects in the set.

← **Input**: integer $k \geq 1$, monotone function S combining ranked lists $R_1, \dots R_m$
→ **Output**: the top k objects according to S

- **Procedure**:

```
step 1. Do a sorted access in parallel in each list $R_i$
step 2. Store in $B$ each retrieved object $o$ and maintain $S^-(o)$ and $S^+(o)$ for each $o$ and a threshold $\tau$
step 3. Go to step (1) as long as there's a new object $\overline{o}$ such that $S^+(o) > \tau$
```

The *NRA* Algorithm is instance optimal among all algorithms that use only sorted accesses *(don't use random accesses)*; the optimality ratio is m. The cost of the algorithm does not grow monotonically with k: it might be cheaper to look for the top k objects than for the top $k-1$ objects.

6.3.3.7 Comparison of the Algorithms

Table 10 shows a comparison between the three algorithms.

algorithm	*scoring function*	*data access*	*notes*
`BO`	`MAX`	sorted	instance-optimal
`FA`	monotone	sorted and random	cost not dependent on scoring function
`TA`	monotone	sorted and random	instance-optimal
`NRA`	monotone	sorted	instance-optimal, no exact scores

Table 10: Comparison of the algorithms

6.4 Skyline Queries

The Skylines Queries technique aims to retrieve the best objects in a set according to several different criteria. Instead of specifying a score function *(as in the Top-k Queries)*, the algorithm is based on the notion of **dominance**.
Tuple t **dominates** tuple s *(denoted $t \prec s$)* if and only if two conditions are satisfied:

1. $\forall\, i\ 1 \leq i \leq m : t[A_i] \leq s[A_i]$ - tuple t is better than s on all attributes
2. $\exists\, k\ \mid t[A_k] < s[A_j]$ - tuple t is better than s on at least one attribute

Note that these conditions assume that a lower score is better; the same principle applies if the objective is to maximize the score function. This convention is the opposite of the one used in the Top-k Queries.
The **skyline** of a relation is the set of its non-dominated tuples:

- Maximal vectors problem *(in computational geometry)* - find the set of non-dominated points in a set of points
- Pareto-optimal solutions *(in multi-objective optimization)* - find the set of non-dominated solutions in a set of solutions

In $2D$, the shape of the contour of the dataset resembles a skyline *(Figure 18)*.

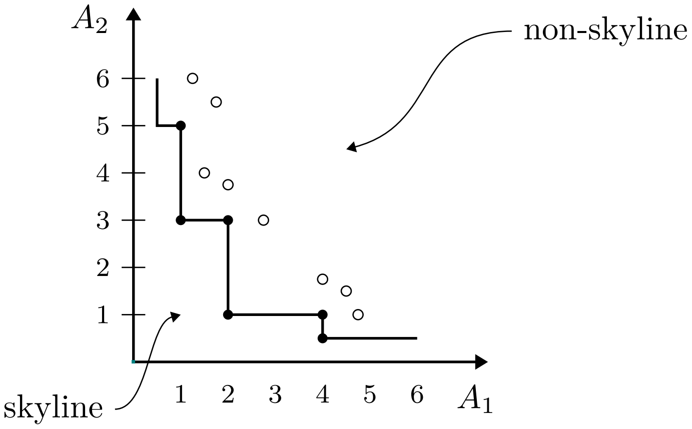


Figure 18: Skyline in 2D

Properties of the skyline

- A tuple t is in the skyline if and only if is the Top-1 tuple according to at least one monotone scoring function
- The skyline is not the Top-k query, since no scoring function yields in the first k positions the skyline points on all possible instances

Main aspects of skyline queries

- **Pros**:
  - ✓ **Effective** in identifying potentially interesting objects if nothing is known about the preference of a user
  - ✓ Very **simple** to use, as no parameter is needed
- **Cons**:
  - × **Too many objects** might be returned in large, non-correlated datasets
  - × Computation is **quadratic** in the size of the dataset *(and thus infeasible for large datasets)*
  - × **Agnostic** to the user's preferences
- **Extension**:
  - **k-skyband**: the set of tuples dominated by less than k tuples

6.4.1 Skyline Queries in `SQL`

There's no standard `SQL` syntax for skyline queries; a proposed syntax is shown in Code 5. A naively implemented skyline query would be very inefficient since it would require a full scan of the table; an example of such an implementation is shown in Code 6.

```
SELECT <attributes>
FROM <table>
GROUP BY <grouping attributes>
HAVING <condition>
SKYLINE OF [ALL | DISTINCT] <attribute> [MIN | MAX | DIFF]
  {, <attribute> [MIN | MAX | DIFF], ...}
ORDER BY <attribute> [ASC | DESC] {, <attribute> [ASC | DESC], ...}
```

Code 5: Proposed syntax for skyline queries

```
SELECT * FROM hotels AS h
WHERE h.city = 'Paris' AND NOT EXISTS (
  SELECT * FROM hotels AS h2
  WHERE h2.city = h1.city AND
    h1.distance <= h2.distance AND
    h1.price <= h2.price AND
    (
      h1.distance < h2.distance OR
      h1.price < h2.price
    )
)
```

Code 6: Naive implementation of skyline queries

6.4.1.1 Block Nested Loops Algorithm

The Block Nested Loops *(BNL)* Algorithm is a skyline query algorithm that uses a **block** of tuples to perform the skyline computation.

← **Input**: dataset D of multi dimensional points
→ **Output**: skyline of D
- **Procedure**: shown in Code 7

```
W := ∅
for every point p ∈ D
  if p is not dominated by any point in W
    W := W \ {points dominated by p}
    W := W ∪ {p}
  end
end
```

Code 7: Block Nested Loops algorithm

The computation of the skyline is $\mathcal{O}\left(n^2\right)$, with $n = |D|$; as such, it's very inefficient for large datasets.

6.4.1.2 Sort-Filter-Skyline Algorithm

The Sort-Filter-Skyline *(SFS)* Algorithm is a skyline query algorithm that uses a **sort** of the tuples to perform the skyline computation.

- ← **Input**: dataset D of multi dimensional points
- → **Output**: skyline of D
- **Procedure**: shown in Code 8

```
S := sort(D) // sort by a monotone function of attributes of D
W := ∅
for every point p ∈ S
  if p is not dominated by any point ∈ W
    W := W ∪ {p}
  end
end
```

Code 8: Sort-Filter-Skyline algorithm

Pre-sorting is very convenient for large datasets, so the *SFS* algorithm is more efficient than the *BNL* algorithm: if the input is sorted, a later tuple cannot dominate any previous one. No two non-skyline points will ever be compared and every point in W can be immediately added to the skyline.
The time complexity, however, is still $\mathcal{O}\left(n^2\right)$.

6.5 Comparison of Top-k and Skyline Queries

A simple comparison of the two techniques is shown in Table 11.

	ranking queries	*skyline queries*
simplicity	✗	✓
overall view of interesting results	✗	✓
control of cardinality	✓	✗
trade-off among attributes	✓	✗

Table 11: Comparison of Top-k and Skyline Queries

7 Java Persistence API - `JPA`

The end-point of many web applications is the *DBMS*; in many cases, it exists independently of the application that uses it. Moving data back and forth between the object model application is a harder task than it would seem: a lot of repetitive code is spent to convert row and column data into objects.
The technique involving conversion between the object model and the relational model is called **object-relational mapping** *(ORM)*. *ORM* techniques try to map the concepts from one model onto the other, overcoming the impedance mismatch between the two models: there's no logical equivalence between the two of them, so the mapping is not always straightforward. A mediator is needed to manage the automatic transformation of one onto the other.
The differences between Object Model and Relational Model are highlighted in Table 12.

Object Oriented Model - `Java`	*Relational Model -* `SQL`
classes, objects	tables, rows
attributes, properties	columns
identity	primary key
reference to other objects	foreign key
inheritance, polymorphism	**not supported**
methods	stored procedures, triggers
code is portable	**vendor-dependent**

Table 12: Object-Relational Mapping

The **`Java` Persistence API** *(`JPA`)* is a standard for object-relational mapping: it bridges the aforementioned gap between the object model and the relational model. It features:

- *POJO (Plain Old Java Object)* **Persistence**: any existing `Java` non-`final` object with default constructor can be persisted
- **Non-intrusiveness**: the persistence layer is completely transparent to the application, as the mapped objects are not aware of it
- **Object queries**: a query powerful is provided to query the database using the object model, without having to use `SQL`

`JPA` Main Concepts

- **Entity**: a `Java` class *(called `JavaBean`)* representing a collection of persistent objects mapped onto a relational table
- **Persistence Unit**: the set of all the classes that are persistently mapped to one database

  → analogous to the **schema** of the database
- **Persistence Context**: the set of all the managed objects of the entities defined in the persistence unit

  → analogous to the **instance** of the database
- **Managed Entity**: an entity part of a persistence context for which the changes of the state are tracked
- **Entity Manager**: the interface for interacting with a Persistence Context
- **Client**: a component that can interact with a Persistence Context via the Entity Manager

A *UML* diagram showing the entities and the relationships between them is shown in Figure 19.

7.1 Entity

A `JavaBean` *(or POJO)* is a `Java` class that gets associated with a tuple in a database:

- the class is mapped to the table schema

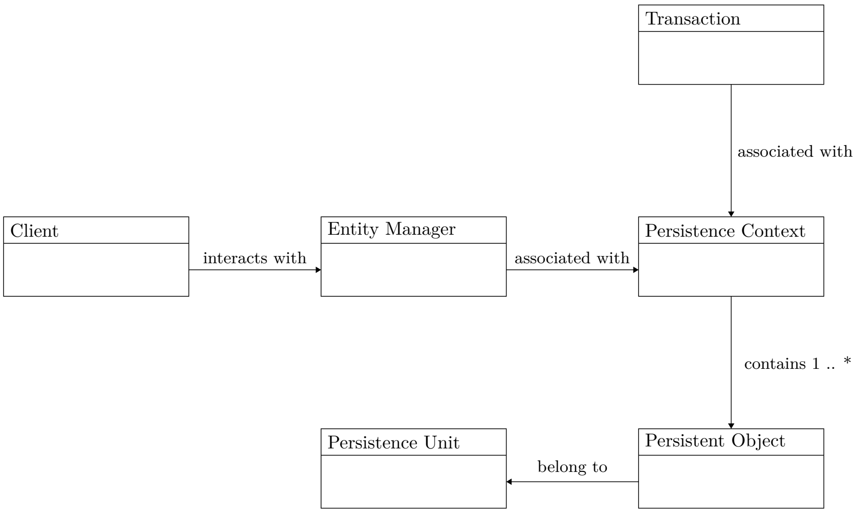


Figure 19: UML Diagram of `JPA`

- the instance of the class is mapped to a tuple

The persistent counterpart of an entity has a longer lifetime than the instance of the class: the instance of the class is created and destroyed during the execution of the application, while the persistent counterpart is stored in the database.
The entity class must be associated with the database table it represents by an operation called **persistence mapping**. An entity can enter a managed state: all the modifications to the entity are tracked by the persistence context and automatically synchronized with the database.

Properties of the Entity The properties of an entity are:

- **identification** *(the primary key)*
- **nesting**
- **relationship**
- **referential integrity** *(the foreign key)*
- **inheritance**

Constraints of an Entity Entities must respect the following **constraints**:

- the entity class must have a `public` or `protected` constructor with **no arguments**
- the entity class must not be `final`
- no method or persistent instance variables of the entity class may be `final`
- if an entity instance is to be passed by value, it must be `Serializable`
- The persistent state of an entity is represented by instance variables, which correspond to `JavaBean` properties

7.1.1 Entity Identification

In a database, objects and tuples have an identity *(the primary key)*: an entity assumes the identity of the tuple it represents; the key can be:

- **simple primary key**, which is a persistent field of the `JavaBean` used to represent its identity
- **composite primary key,** which is a set of persistent fields of the `JavaBean` used to represent its identity

Compared to *POJOs*, the persistent identity is a new and different concept: *POJOs* don't have a **durable identity**.
The syntax of the entity identification is shown in Code 9.

```
@Entity
public class <class name> implements Serializable {
  @Id
  private <type> <field name>;
  ...
}
```

Code 9: Entity Identification

where:

> `@Entity`: is an annotation indicating the **nature** of the entity class
> `@Id`: is an annotation indicating the **simple primary keys** of the entity
> `@EmbeddedId`: is an annotation indicating the **composite primary keys** of the entity

Identifier Generation Sometimes, applications do not want to manage the uniqueness of data values; in this case, the persistence provider can automatically generate an **identifier** *(primary key value)* for every instance of an entity of a class. This feature is called **identifier generation** and is specified by the `@GeneratedValue` annotation; there are 4 different strategies for identifier generation, shown in Table 13.

strategy	*description*
`AUTO`	the provider generates identifiers using whatever strategy is appropriate for the underlying database
`TABLE`	identifiers are generated according to a generator table
`SEQUENCE`	identifiers are generated via sequences if the underlying *DBMS* supports it
`IDENTITY`	identifiers are generated via primary key identity column if the underlying *DBMS* supports it

Table 13: Identifier Generation Strategies

The syntax of the identifier generation is shown in Code 10.

```
@Entity
public class <class name> implements Serializable {
  @Id
  @GeneratedValue(strategy = GenerationType.<strategy>)
  private <type> <field name>;
  ...
}
```

Code 10: Identifier Generation

7.1.2 Attribute Specification

Attributes can be properties that direct the mapping between the object and the relational database; the annotation `@Basic` can be placed before a field or property to mark it as a **persistent attribute**. This annotation is mostly used for documentation, as it is the **default behaviour of the persistence provider**. A non-persistent attribute can be marked with the `@Transient` annotation.

Persistent attributes can be:

- **Primitive** types
- Large **Objects** *(LOBs)*
- **Enumerated** Types
    - `Java` enumerations
    - strings
- **Temporal** Types
    - `Java Date` and `Calendar`
    - `JDBC` temporal types
- **Serializable** Types

Furthermore the fetch policy can be specified for the attributes of the entity, between the:

- **`LAZY`**: the attribute is loaded when it is **first accessed**
    - the attribute value remains empty when the object is retrieved until the attribute is accessed
    - this is best used for LOBs
- **`EAGER`**: the attribute is loaded when the **entity is loaded**
    - this is the default fetch policy

```
@Entity
public class <class name> implements Serializable {
  @Id
  @Basic(fetch = FetchType.<fetch policy>)
  private <type> <field name>;
  @Transient
  private <type> <field name>;
  ...
}
```

Code 11: Attribute Specification

Object to Table Mapping By default entities are mapped to tables with the same name and attributes are mapped to columns with the same name; default value can be overridden by using the `@Table` annotation. Some annotations *(for example `@Column`)* may have attributes used for generating the database schema from the entity classes.

```
@Entity @Table=(name = "<table name>")
public class <class name> implements Serializable {
  @Id
  @Column(name = "<column name>", nullable = <true|false>, length = <length>)
  private <type> <field name>;
}
```

Code 12: Object to Table Mapping

7.1.3 Entity and Relationships

If entities contained only a simple persistent state, the issuing of the *ORM* would be trivial; however, in most cases, entities are related to each other, and the *ORM* must be able to handle these relationships.
There is an ambiguity in the meaning of the word **relationship** in the context of *ORM*: from now on, the relationship will be referred to in the sense of the Object Model *(such as the one shown in Figure 20).*

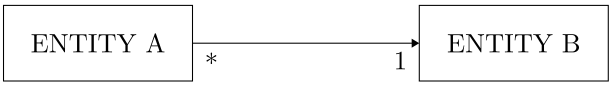


Figure 20: Object Model

Every *OMR* has 4 characteristics:

1. **Directionality**
   → each of the two entities may have an attribute that enables access to the other entity
2. **Role**
   → each entity is said to play a role with respect to the relationship direction of access
3. **Cardinality**
   → the number of entities that can exist on each side of the relationship
4. **Ownership**
   → one of the two entities is said to own the relationship

7.1.3.1 Explanation of the Characteristics

Directionality Each entity in the relationship may have a reference to the other entity:

- **unidirectional**: the relationship is accessible only from one entity
- **bidirectional**: the relationship is accessible from both entities

All relations in `JPA` are unidirectional; bidirectional relationships can be implemented via matched pairs of unidirectional relationships. Matching must be declared explicitly via the `MappedBy` attribute.

Role According to the direction, one entity plays the role of source and the other plays the role of target.
In a relation from `A` to `B`, `A` is the source and `B` is the target.

Cardinality Each role in the relationship has its cardinality, leading to four possible cardinalities:

- **one-to-one**: one source entity, one target entity
- **one-to-many**: one source entity, many target entities
- **many-to-one**: many source entities, one target entity
- **many-to-many**: many source entities, many target entities

A *-to-one relationship implies that the source entity has one reference to the target entity; a *-to-many relationship implies that the source entity has a collection of references to the target entities.

Ownership In the database all Relationships are implemented via a foreign key column that refers to the primary key of the referenced table; in `JPA`, this column is called the **join column**.
In **one-to-many** and **one-to-one** relationships, one of the two entities will have the foreign key column in its table: this entity is said to be the owner of the relationship and its side is called the **owning side**.
In **many-to-many** relationships, the relationship is implemented via a join table that contains the foreign key columns of both entities; in this case, neither entity is the owner of the relationship.
Ownership is important because the annotations that define the physical mapping of the relationship are defined on the owning side.

7.1.3.2 Relationship Mapping

The possible relationships between entities are:

- $1 : N$ - bidirectional, unidirectional one-to-many and many-to-one
- $1 : 1$ - bidirectional, unidirectional
- $N : M$ - bidirectional, unidirectional

The choice between uni- and bidirectional relationships is a matter of design and implementation; generally speaking, if the relationship is accessed from one side only, it is unidirectional, otherwise, it is bidirectional. Bi-directionality can be implemented with unidirectional relationship mapping and set-valued queries. Performance considerations may prompt the use of bidirectional mappings even if only one direction is used.

$1 : N$ **bidirectional** The $1 : N$ bidirectional relationship is expressed via the `@ManyToOne` and `@OneToMany` annotations, together with the `MappedBy` attribute.
The many-to-one mapping direction is defined by annotating the entity that participates with multiple instances with the `@ManyToOne` annotation; the entity that contains the `@ManyToOne` annotation is the **owning** side of the relationship. The `@JoinColumn` annotation is used to specify the **join column** that is used to **reference** the primary key of the target entity *(it defaults to the data member name)*.
To achieve the bidirectional relationship, the mapping direction must be specified too; this is done by including a `@OneToMany` annotation in the entity that participates with one instance. The `@OneToMany` annotation is placed on a collection data member and includes a `MappedBy` attribute that specifies the name of the data member in the target entity that is the owner of the relationship.
In a **one-to-many mapping**, the **owner** of the relationship is the entity that participates with **multiple instances**.

```
@Entity
public class Many {
  @Id
  private int id;
  @ManyToOne
  @JoinColumn(name="one_fk")
  private One one;
}

@Entity
public class One {
  @Id
  private int id;
  @OneToMany(MappedBy="one") // case insensitive
  private List<Many> many;
}
```

Code 13: Bidirectional `@ManyTooOne` annotation

1 : *N* unidirectional Sometimes applications require access to related entities to be unidirectional: in this case, bi-directional mapping is not needed. The same annotation as in bidirectional mapping is used, but the `MappedBy` attribute is omitted.

```
@Entity
public class Many {
  @Id
  private int id;
  @ManyToOne
  @JoinColumn(name="one_fk")
  private One one;
}

@Entity
public class One {
  @Id
  private int id;
  @OneToMany
  @JoinColumn(name="one_fk")
  private List<Many> many;
}
```

Code 14: Unidirectional `@ManyToOne` annotation

The same goes for the `@OneToMany` annotation; however, two alternatives are available performance-wise:

- Map the relationship as in the bi=directional case and use only the one-to-many direction
- Do not map the collection attribute in the entity that participates with one instance and use a query instead to retrieve the correlated instances, relying on the inverse *(many-to-one)* relationship direction mapping

1 : 1 unidirectional In a one-to-one relationship, the `@OneToOne` annotation is used to specify the relationship direction, as either entity can be the owner of the relationship. A one-to-one mapping is defined by annotating the owner entity (the one with the foreign key column) with the `@OneToOne` annotation.

```
@Entity
public class One {
  @Id
  private int id;
  @OneToOne
  @JoinColumn(name="one_fk")
  private Two two;
}

@Entity
public class Two {
  @Id
  private int id;
  @OneToOne(MappedBy="two")
  private One one;
}
```

Code 15: Unidirectional `@OneToOne` annotation

1 : 1 **bidirectional** The one-to-one bidirectional relationship is expressed like the unidirectional counterpart, but the other entity must be annotated with the `@OneToOne` annotation too and include the `MappedBy` attribute; the presence of the former tells `JPA` that the foreign key constraint is defined in the target entity.

```
@Entity
public class One {
  @Id
  private int id;
  @OneToOne
  @JoinColumn(name="one_fk")
  private Two two;
}

@Entity
public class Two {
  @Id
  private int id;
  @OneToOne(MappedBy="two")
  private One one;
}
```

Code 16: Bidirectional `@OneToOne` annotation

$N : M$ **unidirectional** In a many-to-many mapping, there's no foreign key column; such mapping is implemented via a bridge table *(also called join table)*. Either entity can therefore be the owner of the relationship.

```
@Entity
public class N {
  @Id
  private int id;
  @ManyToMany
  @JoinTable(name="N_M",
             joinColumns=@JoinColumn(name="n_fk"),
             inverseJoinColumns=@JoinColumn(name="m_fk"))
  private List<M> m;
}

@Entity
public class M {
  @Id
  private int id;
  @ManyToMany
  @JoinTable(name="N_M",
             joinColumns=@JoinColumn(name="m_fk"),
             inverseJoinColumns=@JoinColumn(name="n_fk"))
  private List<N> n;
}
```

Code 17: Unidirectional `@ManyToMany` annotation

$N : N$ **bidirectional** The many-to-many bidirectional relationship is expressed like the unidirectional counterpart, but the other entity must be annotated with the `@ManyToMany` annotation too and include the `MappedBy` attribute.

```
@Entity
public class N {
  @Id
  private int id;
  @ManyToMany
  @JoinTable(name="N_M",
             joinColumns=@JoinColumn(name="n_fk"),
             inverseJoinColumns=@JoinColumn(name="m_fk"))
  private List<M> m;
}

@Entity
public class M {
  @Id
  private int id;
  @ManyToMany(MappedBy="m")
  private List<N> n;
}
```

Code 18: Bidirectional `@ManyToMany` annotation

7.1.3.3 `@JoinColumn` and `MappedBy`

The annotation `@JoinColumn` is used to specify the join column that is used to reference the primary key of the target entity; such annotation is normally inserted in the entity owner of the relationship (the one mapped to the table that contains the foreign key column). It is used to drive the generation of the `SQL` code to extract the correlated instances.

The `MappedBy` attribute is used to specify the name of the data member in the target entity that is the owner of the relationship; the owner of the relationship resides in the related entity. It is used to specify bidirectional relationships.

In absence of the `MappedBy` attribute, the default `JPA` mapping uses a bridge table to implement the relationship. The purpose of this annotation is to instruct `JPA` to not create a bridge table as the relationship is already being mapped by the other side.

7.1.3.4 Relationship Fetch Mode

When loading an entity, it's not always necessary to fetch and load all related entities; performance can be optimized by deferring data loading until it's needed. This design pattern is called lazy loading *(opposed to eager loading)* and is supported by `JPA`. At the relation level, lazy loading can greatly enhance performance because it can reduce the amount of `SQL` code that is executed if correlated instances are seldom accessed by the application.

Loading policy can be expressed specifying the `FetchType` attribute of the `@ManyToOne`, `@OneToOne`, and `@OneToMany` annotations; when it's not specified, a single-valued relationship is fetched eagerly, while a collection-valued relationship is fetched lazily.

In the case of bidirectional relationships, the fetch mode might be lazy on one side but eager on the other: this situation might occur when the direction of navigation occurs from one side to the other, but not vice versa.

The directive to lazily fetch an attribute is meant to only be a hint to the persistence provider: it is not required to respect the request because the behaviour or the entity is not compromised if the provider decides to load eagerly. The converse however is not true: specifying that an attribute is to be fetched eagerly is a requirement that the provider must respect.

7.2 Entity Manager

Since entity instances are plain `Java` objects, they do not become managed until the application invokes an `API` to initiate the process. The Entity Manager is the central authority for all operations involving entities: it manages the mapping between a fixed set of entity instances and an underlying data source while providing an `API` for the application to access and manipulate the data.

7.2.1 Entity Manager Interface

The Entity Manager exposes all the operations needed to synchronize the managed entities in the persistence context to the database; the methods signatures and their descriptions are shown in Table 14.

signature	*description*
`public void persist(Object entity);`	persists an entity instance in the database
`public <T> T find (Class<T> entityClass, Object primaryKey);`	finds an entity instance by its primary key
`public void remove(Object entity);`	removes an entity instance from the database
`public void refresh(Object entity);`	resets the entity instance from the database
`public void flush();`	writes the state of entities to the database **immediately**

Table 14: Entity Manager Interface

7.2.2 Cascading Operations

By default, every Entity Manager operation applies to the only entity supplied as an argument to the operation: the operation will not cascade to other entities that have a relationship with the entity supplied as an argument. For some operations *(such as `remove`)*, this normally is the desired behaviour; for some other operations *(such as `persist`)*, however, it is not since they require propagating to other entities that are related to the entity supplied as an argument. If an entity has a relationship with another dependent entity, normally the child must be persisted together with the father.

The `cascade` attribute is used to define when operations should be automatically cascaded across the relationship; it accepts several possible values specified in the `CascateType` enum:

- `PERSIST`
- `REFRESH`
- `REMOVE`
- `MERGE`
- `DETACH`
- `ALL`
  → this is a shorthand to specify that all operations should be cascaded

Like in relations, the cascade settings are unidirectional: they must be specified explicitly on both sides of the relationship if the default behaviour is not desired.

Unless otherwise specified, no cascade operation is performed.

7.2.2.1 Orphan Removal

JPA offers an additional feature to automatically remove orphan entities via the orphanRemoval attribute in the @OneToOne and @OneToMany annotations.
This attribute is appropriate for privately owned part-child relationships, where every child entity is associated only with one parent entity through just one relationship (weak entity in the ER model). The relationship is broken either by removing the parent entity or by setting the child entity's relationship to the parent to null; in the one-to-many case, the child entity is removed from the collection.
With the cascade attribute, no element is deleted from the database when the parent entity is removed; the orphanRemoval automatically deletes them.

7.2.3 Persistence Context

The persistence context is a fundamental component of JPA: it's a kind of main memory database that holds the objects that are being manipulated by the application *(in the managed state)*. A managed object is tracked and all the modifications to its state are monitored for automatic alignment to the database; each of them has two lives: one as a Java object and one as a relational tuple *(the ID of the POJO is the primary key of the relational tuple)*. Such bindings exist only inside the persistence context; when the POJO exits the persistence context, the binding is lost and the POJO is no longer managed.
Database writes occur asynchronously, at a time decided by the persistence provider; for the database writes to happen, the Persistence Context must hook up to a transaction, which is the only way to write to the database.

7.2.3.1 Creating a new *POJO*

Calling the new operator does not interact with some underlying database to create a new entity instance; when an entity instance is created, it is in the **transient** state since the Entity Manager has no knowledge of it. Transient entities are not part of the Persistence Context associated with the Entity Manager.

Managing an Entity Entity Manger's persist method is used to make an entity instance managed and persistent; it does not imply that the entity is immediately written to the database. A managed entity lives in a Persistence Context and is associated with an Entity Manager that makes sure that any change to its state is tracked for being persisted to the database: the managed entity and the corresponding tuple become associated until the entity exits the managed state. When the transaction associated with the Entity Manager's persistence context is committed, the Entity Manager writes the changes to the database.
Calling persist on an entity that is already managed is allowed and triggers the cascade operation.
The flush method can be called to ask the Persistence Provider to write the changes to the database immediately.

Finding an Entity The find method is used to find an entity instance by its primary key; when the call completes, the returned object will be managed and added to the Persistence Context. If the entity is not found, the method returns *null*.
The actual amount of data extracted and added to the persistence context depends on the Fetch Policy of attributes and relationships.

Removing an Entity The remove method is used to break the association between an entity and its Persistence Context; when the transaction associated with the Entity Manager's persistence commits of the flush method is called, the entity is scheduled for deletion from the database.
The entity still exists but its changes are no longer tracked for being synchronized to the database.

7.2.3.2 Entity Life Cycle

The life cycle of an entity is divided into five states:

> NEW

- unknown to the Entity Manager, not yet associated with a Persistence Context

> MANAGED

- associated with a Persistence Context, changes to its state are tracked for being synchronized to the database

> `DETACHED`

- its identity is potentially associated with a database tuple but changes are not automatically propagated to the database

> `REMOVED`

- scheduled for removal from the database

> `DELETED`

- erased from the database

7.3 JPA Application Architectures

The client exploits the services of a container (via `EJB` or `CDI`) to connect to the entity manager; the client then interacts with business components, that in turn interact with the Entity Manager. The container provides support to map the `JPA` entity method calls into transactions via the automatic creation of transactions.
Details of the implementation:

- an Entity Manager is associated with a persistence context, which tracks changes to entities
- changes are synchronized to the database via transactions
- the transactions exist outside the Entity Manager, which hooks the persistent context to it
- a transaction is the only channel to write to reflect the changes in the Persistence Context to the database
- a transaction can be associated with only one Persistence Context: it reads the changes in it and maps them into transactional operations onto the database

Furthermore:

- Entity managers are the only interface to the Persistence Manager
- Different components may use their Entity Managers
- Applications may need to work with different components, therefore with different Entity Managers

7.3.1 Transaction Management in JPA

Database application development requires understanding the transactional properties of the code, such as how the Entity Manager participates in transactions and when a Transaction starts or ends. Even if transactions are managed transparently by the container, it's important to understand how they work in order to predict the behaviour of the application.
Transactions exist at 3 abstraction levels: *DBMS* transactions, Resource-Local transactions and Container transactions.

- *DBMS*
  - they live inside the *DBMS*
  - they are described via `SQL` commands
  - they are managed by the *DBMS*
- Resource Local
  - they are described and issued via `JDBC` interface
  - they are mapped by the `JDBC` to *DBMS* transactions
  - they must be managed by the application
- Container (level used in the course)
  - they are defined through the `JTA` interface
  - they are mapped by the `JTA` Transaction Manager and `EJB` container to `JDBC` transactions
  - they are managed by the application or by the container

7.3.1.1 Container Managed Entity Manager

The steps under which the Contained Managed (CM) Entity Manager operates are:

1. the container injects the Entity Manager into the business object (the client component)
2. the container creates and destroys instances of the Entity Manager as needed, transparently to the application
3. the container provides the transaction needed for saving the modifications made to the entities of the Persistence Context associated with the Entity Manager into the database

This is the default behaviour of the Entity Manager unless otherwise specified.
The syntax of the CM Entity Manager is shown in Code 19.

```
@Stateless
public class <business-object-name> {
  @PersistenceContext(unitName = "<persistence-unit-name>")
  private EntityManager <entity-manager-name>;
}
```

Code 19: CM Entity Manager

7.3.1.2 Transacion Management at Container Level

The `JPA` Entity Manager:

- the transaction is created by the container externally to the Entity Manager
- in context-managed mode, the transaction demarcation can be delegated to the container
  - the transaction is created when a business method that requires it is called and there is no active transaction yet
  - the transaction is committed when the method that caused its creation terminates
- in bean-managed transactions (AM mode), the transaction is created by the component of the application that can implement the `EntityTransaction` (in `JPA`) or `UserTransaction` (in `JTA`) interfaces to manage the transaction and `begin` and `commit` it explicitly

7.3.1.3 Propagation

The process of sharing a Persistence Context between multiple Container Managed Entity Managers in a single `JTA` transaction is called **transaction propagation**. Thanks to propagation, methods of different components can share the same Persistence Context and thus the same database connection.
Propagation has two forms:

- **vertical** along the call stack, if the same transactions are executed by different components
  - the Transaction and the Persistence Context are shared between the components
  - the **`@Required`** annotation is needed
- **Horizontal**, if the Entity Managers of the two components are defined on the same Persistence Unit
  - the two components share the same Persistence Context

7.3.1.4 Transaction and Method Calls

When a client calls a method of a business object that exploits a Context Managed Entity Manager for persistence, the container (EJB) provides a transaction for saving the modifications made into the database, either by creating a new transaction or by joining an existing one. If the called method, in turn, calls another method of the same or a different business object, the same transaction is reused.

This is the most common and default behaviour, but the business methods can be annotated to specify a different way to use the transactions provided by the container.

When Container-Managed transactions are used, the container wraps the method calls of managed components and executes them in a transaction. A method can be annotated to obtain the desired transactional behaviour with the annotation:

```
@TransactionAttribute(TransactionAttributeType.<type>)
```

where `type` can be:

> `REQUIRED` - if no transaction is active, then one is started; otherwise, the method is executed in the active transaction. This is the default type.
> `MANDATORY` - a transaction is expected to have already been started and be active when the method is called; otherwise, an exception is thrown
> `REQUIRES_NEW` - a new transaction is started, and the method is executed in it; if a transaction is already active, it is suspended during the execution of the method
> `SUPPORTS` - the method is executed in the active transaction if any; otherwise, it is executed without a transaction
> `NOT_SUPPORTED` - the method is executed without a transaction, even if one is active; if a transaction is active, it is suspended during the execution of the method
> `NEVER` - the method is executed without a transaction, even if one is active; if a transaction is active, an exception is thrown

8 Reliability

Reliability is defined as the ability of an item to perform a required function under stated conditions for a stated time length. In databases, reliability control ensures fundamental properties of transactions:

- Atomicity: all-or-nothing execution of a transaction
- Durability: once a transaction is committed, its effects are permanent

The *DBMS* implements a specific architecture for reliability control; the keys components are found in **stable memory** and **log management**.

8.1 Reliability Manager

The Reliability Manager of the *DBMS* realizes the transactional commands `COMMIT` and `ABORT`, which are executed by the `Transaction Manager`. Furthermore, it orchestrates read and write access to pages *(of data and log)* and handles recovery after failures.

8.1.1 Persistence of Memory

Durability implies a memory whose content lasts forever, which is an abstraction built on top of existing storage technology levels.

- Main Memory: non-persistent
- Mass Memory: persistent but can be lost
- Stable Memory: cannot be lost, the probability of failure is zero

Since achieving zero probability of failure is impossible, the objective is to reduce the probability of failure to a negligible value; the techniques employed are replication and write protocols. The failure of stable memory is the subject of the discipline called disaster recovery.

Stable memory can be guaranteed via:

- On-line replication
    - the data is replicated on multiple disks
    - the RAID disk architecture is an example of online replication
- Off-line replication
    - the data is replicated on backup units

8.1.2 Main Memory Management

The objective of Main Memory Management is to reduce data access time without compromising memory stability, via buffers to cache data in faster memory and deferred writing onto the secondary storage.
A page of the buffer can contain multiple rows and have a:

- transaction counter, stating how many transactions are accessing it
- dirty flag, stating whether the page has been modified and must be aligned to the secondary storage

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

Buffer Management Primitives

> `fix`: responds to the request of a transaction to load a page into the buffer; returns a reference to the page and increments transaction counter
> `unfix`: unloads a page from the buffer; decrements transaction counter
> `force`: moves a page from the buffer to the secondary storage
> `set_dirty`: sets the dirty flag of a page
> `flush`: moves pages from the buffer to the secondary storage, when they are no longer needed

Buffer Management Policies

- Write Policies: page writing to disk is normally asynchronous with respect to the transactions
  - > `force`: pages are always transferred at commit
  - > `no_force`: transfer of pages can be delayed by the buffer manager
- De-allocation Policies
  - > `steal`: a page in an active transaction is discarded and flushed to disk
  - > `no_steal`: the transaction is put on a waitlist and the request is managed when the page is no longer needed
- Pre-fetch Policies: pages that are likely to be read are loaded in the buffer before they are needed
- Pre-flushing Policies: pages that are likely to be written are de-allocated from the buffer before they are needed

8.1.2.1 Execution of a `fix` primitive

The execution of a `fix` primitive is performed in 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**: the new page is loaded, the transaction counter is incremented and a reference to the page is returned
   - > **`no_steal` policy**: the transaction is put in a waitlist

8.2 Failure Handling

Recall on transactions A transaction is an atomic transformation from an initial state into a final state. The possible outcomes of a transaction are:

- commit: success
- rollback or fault before commit: undo
- fault after commit: redo

Implications for recovery after failure If a failure occurs between commit and buffers flush in secondary storage, then the transactions are rolled forward by the Reliability Manager and the buffers are flushed to disk; if a failure occurs before commit, the Reliability Manager rolls back the transactions, discarding any changes made to the buffers.

8.2.1 Transactions and Recovery

Refer to Figure 21.
Suppose that *DBMS* starts at time T_0 and fails at T_f. Assume that data for transactions T_2, T_3 has been written to secondary storage (committed and permanently stored): transactions T_1, T_6 have to be undone.
In absence of information on whether modified pages have been flushed, Reliability Manager has to red T_4 and T_5.

8.2.2 Transaction Log

The transaction log is a file containing records describing the actions carried out by the various transactions. It's written sequentially up to the block top (the current instant).
The log is recorded on sable memory in the form of state transitions (the actions carried out by the transactions). If an update (`U`) operation transforms object O from value v_1 to value v_2, the log contains the following record:

- `BEFORE-STATE(U)` $= v_1$

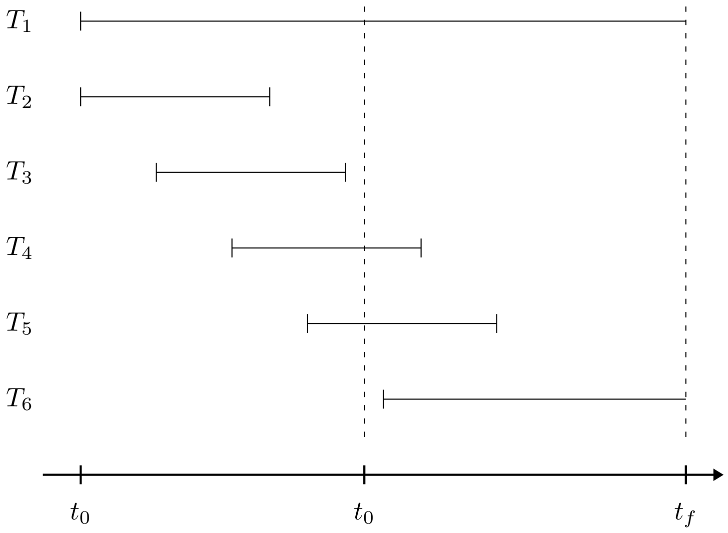


Figure 21: Transactions and Recovery

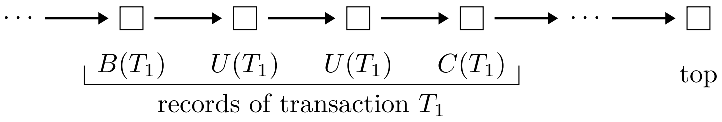


Figure 22: Transaction Log

- `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`

Using the log Assume a transaction T_1 that updates object O from value O_1 to value O_2.
To rollback the transaction or to fix a failure that occurred before the commit, the log is used to recover the previous state of the object:

$$\texttt{UNDO}\ T_1\text{: } O = O_1$$

To fix a failure that occurred after the commit, the log is used to recover the new state of the object:

$$\texttt{REDO}\ T_1\text{: } O = O_2$$

Idempotence of `UNDO` and `REDO`

- `UNDO`(T) = `UNDO`(`UNDO(T)`)
- `REDO`(T) = `REDO`(`REDO(T)`)

Idempotence is an important property of the two operations because the Reliability Manager may have to apply them multiple times; for example, if a failure occurs after the commit of a transaction T_1 and the log is not flushed, the Reliability Manager has to apply `REDO` to T_1 and then to T_2. This property ensures that the Reliability Manager can apply the operations multiple times without changing the state of the database.

Types of Log Records

- Records concerning a transactional command:
  - > `B(T)` - begin transaction `T`
  - > `C(T)` - commit transaction `T`
  - > `A(T)` - abort transaction `T`
- Records concerning UPDATE, DELETE and INSERT operations:
  - `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`

8.2.2.1 Transactional Rules

Log management must ensure that transactions implement `WRITE` operations reliably; to do so, the Reliability Manager has to apply 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 carrying out the corresponding operation on the database (write-ahead logging)
3. the after-state must be written in the log before carrying out the commit (commit rule)

Rule 2 ensures that actions can be undone in case of failure; rule 3 ensures that actions can be redone in case of failure.

8.2.3 Types of Failures

There are 3 types of failures:

- Soft failure
  - loss of part or all of the main memory
  - requires a warm restart
  - log is exploited to replay the transactions
- Hard failure
  - failure or loss of part or all of the secondary memory devices
  - requires a cold restart
  - dump is exploited to replay the transactions
- Disaster
  - loss of stable memory (of the log and the dump)
  - not treated in this course

8.2.3.1 Checkpointing

Checkpoints are used to reduce the amount of work that has to be done in case of failure; they are snapshots of the database and the log at a given time.
They are performed periodically by the Reliability Manager: all transactions that committed flush their data from the buffer to the disk, while all active transactions are recorded in the log.
A simple checkpointing strategy consists of the following steps:

1. acceptance of all transactions that have committed their work
2. suspension of all abort requests
3. all dirty buffer pages modified by committed transactions rea transferred to mass storage forcefully
4. the identifiers of the transactions still in progress are recorded in the checkpoint log record; no new transaction can start while this record is being written
5. acceptance of the operations is resumed

In this way, it's ensured that all the data relative to committed transactions are on mass storage, while transactions that are still ongoing are listed in the checkpoint log record (in stable memory).

8.2.3.2 Dump

The dump is a complete backup of the database at a given time.
Dumps are normally created when there's low activity on the database; usually at night or during the weekend. The availability of the dump is recorded in the log, while the content of the dump itself is stored in stable memory.

8.2.3.3 Restarting the Database

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

1. find the most recent checkpoint
2. build 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 ∪ {T\_i} // started, maybe undone
  else if C(T_i) then
    UNDO := UNDO \ {T\_i}
    REDO := REDO ∪ {T\_i} // ended, to redo
  end
end
```

3. undo the transactions in the `UNDO` set by applying the `UNDO` operations in reverse order
4. redo the transactions in the `REDO` set by applying the `REDO` operations

Cold Restart The cold restart is performed according to the following steps:

1. data is restored starting from the last backup (dump)
2. operations recorded into the log before the failure time are replayed
3. data on dick is restored in the status existing at failure time
4. a warm restart is performed

During the cold restart, uncertain transactions are undone.

9 Tricky Exercises and how to slay them

9.1 Concurrency Control

9.1.1 Schedule classification

Due to the presence of blind writes, the *CG* of a schedule is cyclic; as such, it's not in *CSR*. By swapping two of the nodes in the *CG*, it may be possible to obtain a schedule that is in *VSR*; if it's not possible, then the schedule is not in *VSR* either.
In some schedules, the blind writes may appear multiple times involving the same transactions with different objects in different orders; in such cases the schedule is not in *CSR (and thus not in VSR).*

9.1.1.1 *2PL*

It's necessary to impose temporal constraints on the lock and unlock requests, based on two principles:

- **same resource** R - each lock can be acquired if R is free
$$U_i^{rR} < L_j^{wR} \quad U_i^{wR} < L_j^{rR} \quad U_i^{wR} < L_j^{wR} \qquad i \neq j$$
- **same transaction** i - all releases must follow all acquisitions
$$L_i^{r/w^{R_n}} < L_i^{r/w^{R_m}} \qquad n \neq m$$

The instants in which the actions take place, in to write inequalities. This approach is needed because the non-strict *2PL* does not allow showing when a transaction has released a lock.

9.1.2 Update lock

A common notation to avoid ambiguity between Unlock and Update is to use the notation $\texttt{SL}_i(x)$ for a `SL` request on resource x by transaction T_i and $\texttt{rel}(\texttt{SL}_i(x))$ for the corresponding release.
In order to avoid deadlocks due to interleaved lock upgrades, any upgrade `SL` → `XL` is not allowed; the transaction needs to acquire a `UL` lock first and then upgrade it to a `XL` lock.

9.2 Structures and Indexes

A comparison of indexing methods related to the data structures is shown in Table 15.

type of index	*type of structure*	*search key*	*ordering*	*allowed number of indexes*
primary	sequential with $SK = OK$	unique	dense or sparse	one per table
secondary	sequential with $SK \neq OK$	unique or not	dense	many per table
clustering	sequential with $SK = OK$	not unique	normally sparse	one per table

Table 15: Comparison of the three indexing techniques