Data bases 2

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2022-12-04
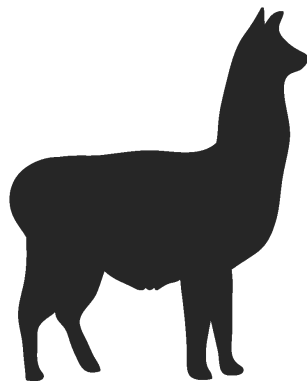
no alpaca has been harmed while writing these notes

Contents

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
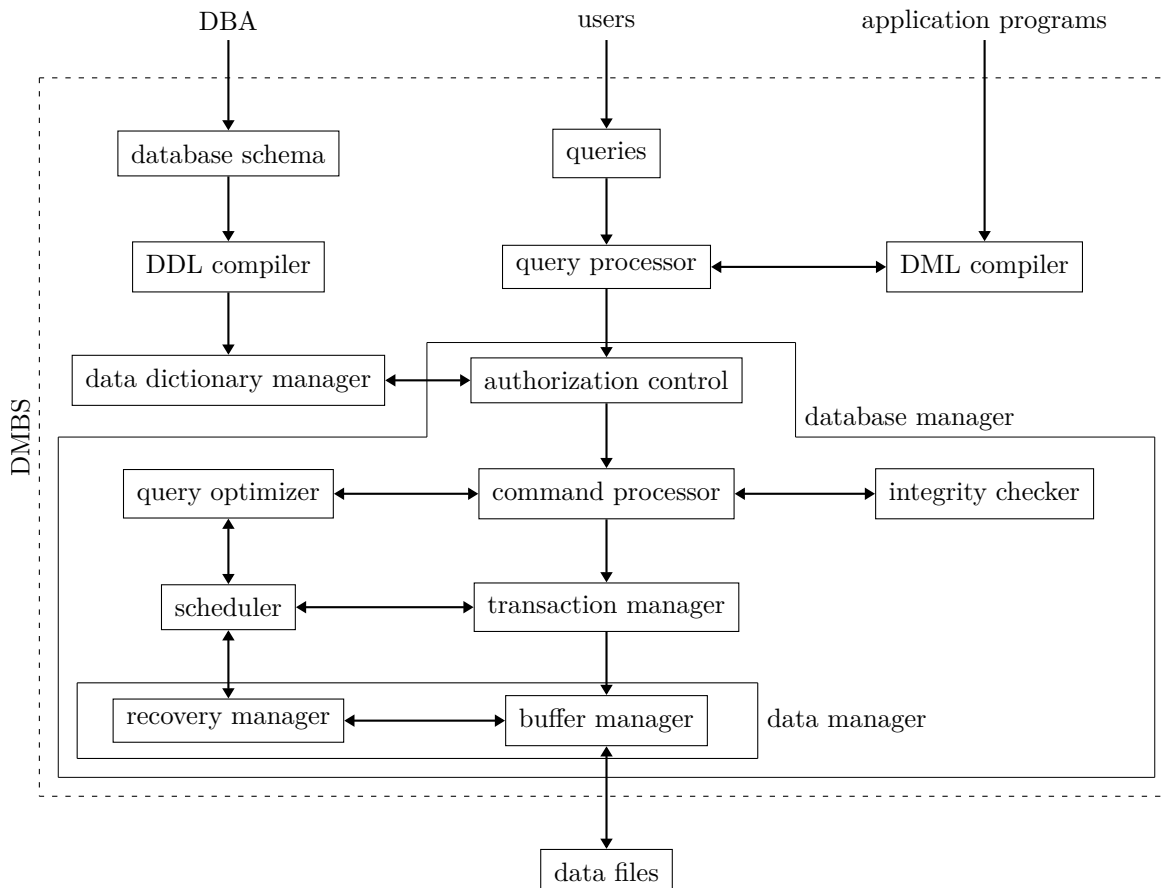
Figure 1: Architecture of the *DBMS*

Capabilities of the *DBMS*:

- **Transaction** management
  - → ACID properties make sure that a set of operations is performed as a single unit
- **Concurrency** control
  - → *CC* theory, pessimistic and optimistic locking prevent data corruption in presence of concurrent accesses
- **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
  - → 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

The mechanisms provided by the *DBMS* are shown in Table 1.

Atomicity	Consistency	Isolation	Durability
abort-rollback-restart reliability manager	integrity checking of the *DBMS* integrity control system at query execution time	concurrency control Concurrency Control System	recovery management reliability manager

Table 1: ACID properties and the *DBMS* mechanisms

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

3.1.1 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

3.1.2 Schedules

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

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

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

3.1.3 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
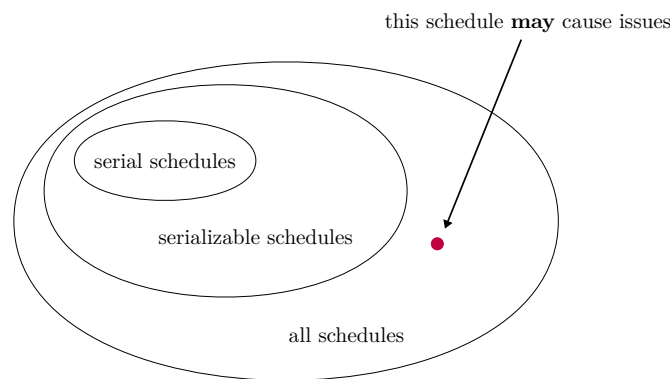
Figure 2: Schedules equivalence

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*

3.1.3.1 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.

3.1.4 Conflict serializability

First, let's define the notion of **conflict**:

- 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*

3.1.4.1 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

3.1.4.2 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.
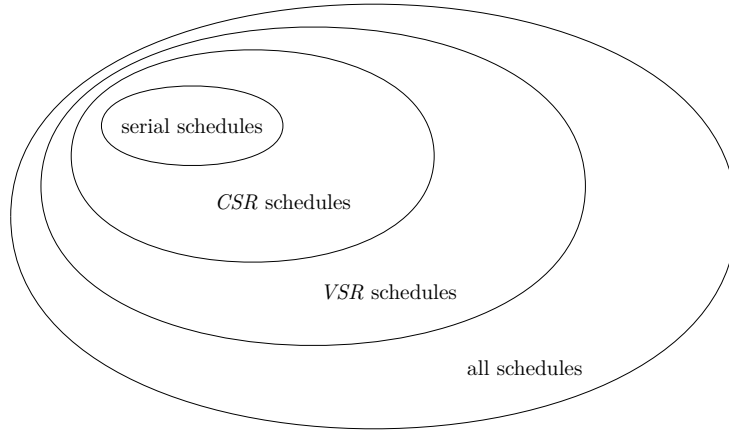
Figure 3: Relation between *VSR* and *CSR*

3.1.4.3 *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 $i > j)$.*

3.1.4.4 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($x$)`, to acquire a read lock on x
- `w_lock($x$)`, to acquire a write lock on x
- `unlock($x$)`, 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 2).

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 2: 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**, written 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 transaction 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.
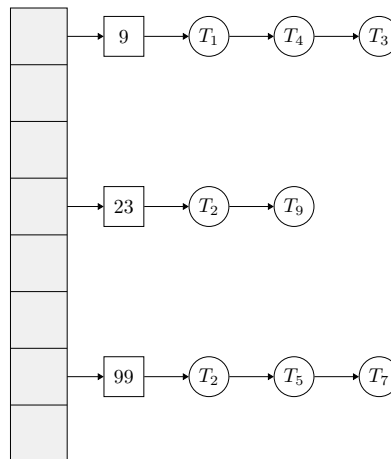

Figure 4: Illustration of a lock table

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
- explored 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. 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. $\mathrm{OP}_i^h(x), \mathrm{OP}_j^k(x) \ldots \mathrm{OP}_j^u, \ldots \mathrm{OP}_i^w(x)$ 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.

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.
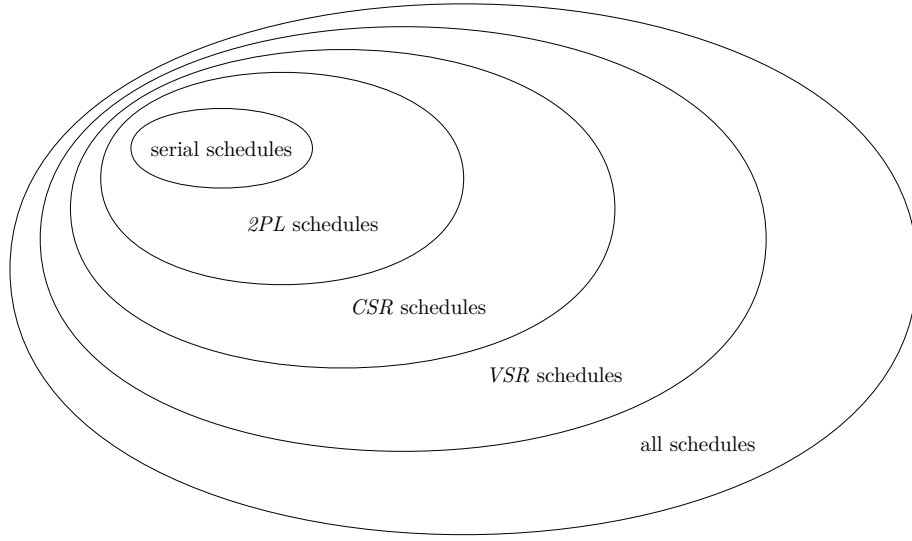
In practice:

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

- *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)

    $\rightarrow T_2$ 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 3.

`SQL` isolation levels may be implemented with the appropriate use of locks: Table 4 shows the locks that are held by the *DBMS* for each level. Normally, commercial systems use locks and timestamp-based concurrency control mechanisms.

isolation level	dirty read	non repeatable read	phantoms
read uncommitted	✔	✔	✔
read committed	✘	✔	✔
repeatable read	✘	✘	✔
snapshot	✘	✘	✘
serializable	✘	✘	✘

Table 3: `SQL` isolation levels

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

Isolation level / Operation	no isolation	read uncommitted	read committed	repeatable read	serializable
READ	none	none	shared record lock (RL)	shared record lock (RL)	shared file lock (FS)
WRITE or update	none or update record lock (RU)	update record lock (RU)	update record lock (RU)	update record lock (RU)	update file lock (FU) and intent file lock (IX)
SELECT	-	-	-	-	intent file lock (IX)

Table 4: SQL isolation levels and locks

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
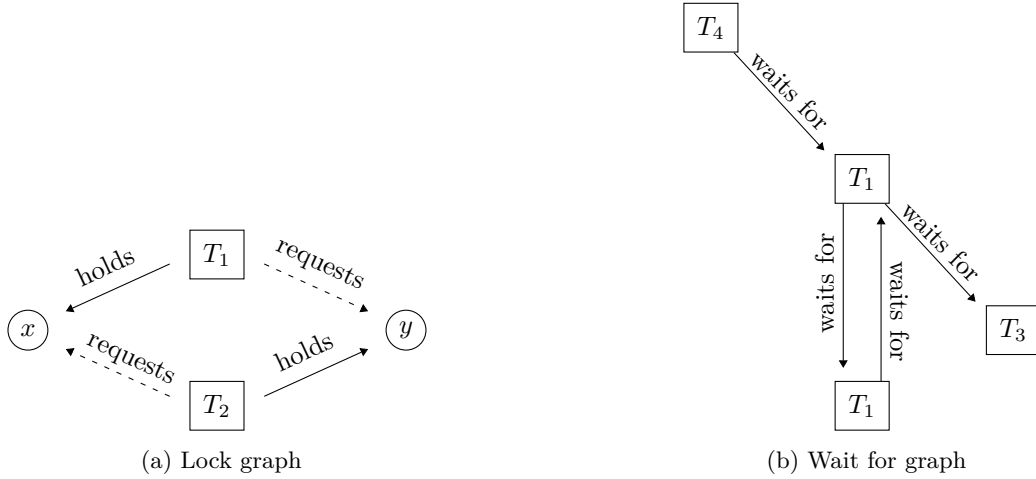
Figure 7: Representations of a deadlock

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
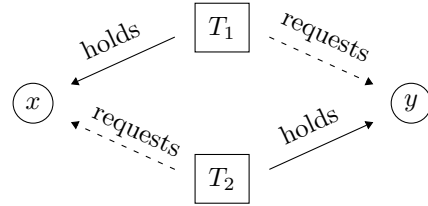
3.6 Deadlock Resolution

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

   $\rightarrow$ 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.
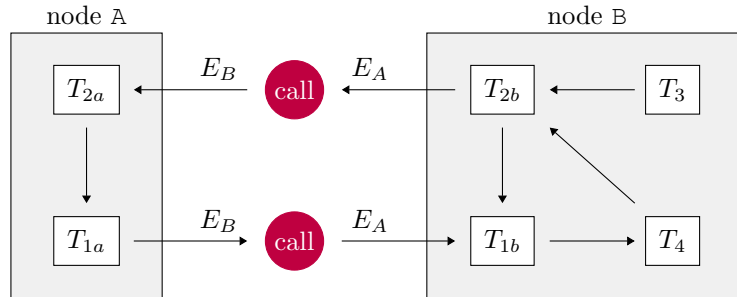
Figure 8: Distributed dependency graph

Description of the graph:

- Representation of the status:
  - at node `A`: $E_b \rightarrow T_{2a} \rightarrow T_{1a} \rightarrow E_b$
  - at node `B`: $E_a \rightarrow T_{1b} \rightarrow T_{2b} \rightarrow E_a$
- E_a and E_b are external call nodes
- The symbol $\rightarrow$ 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 regard to 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 5 shows the escalation of the update lock.

status *request*	*free*	SL	UL	XL
SL	✔	✔	✔	✘
UL	✔	✔	✘	✘
XL	✔	✘	✘	✘

Table 5: 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

```
  field
    ↓
 database
    ↓
  table
    ↓
 fragment
    ↓
  tuple
```
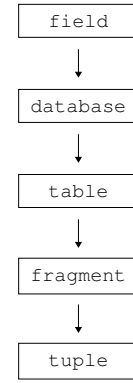
The technique works by:

- 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 6

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.

request \ resource state	free	ISL	IXL	SL	SIXL	XL
ISL	✔	✔	✔	✔	✔	✖
IXL	✔	✔	✔	✖	✖	✖
SL	✔	✔	✖	✔	✖	✖
SIXL	✔	✔	✖	✖	✖	✖
XL	✔	✖	✖	✖	✖	✖

Table 6: Hierarchical lock compatibility

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; as such, the timestamp is not an indicator of global time anymore. 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 $r_{ts}(x)$ or a $w_{ts}(x)$ 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
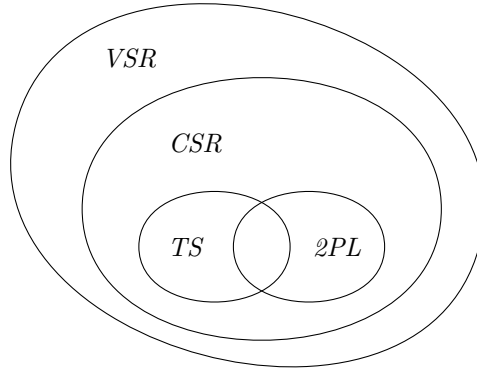
Figure 9: Taxonomy of concurrency control methods

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*

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

	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 7: Differences between *2PL* and *TS*

- - S contains $OP_1(x)$ and $OP_2(x)$ where at least one of the two is a WRITE
  - S also contains $OP_2(y)$ and $OP_1(y)$ where at least one of the two is a WRITE
- When $OP_1(y)$ arrives, the following statements are true:
  - if $OP_1(y)$ 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 $OP_1(y)$ 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 7.
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 WTM$_i$ 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:

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

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 additional, 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 do this is by analysing the triggering graph:

- A node i is associated with each trigger T_i
- An arc from a node i to a node j exists if the execution of trigger T_i causes the execution of trigger 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   **SQL** syntax

```
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 condition holds); 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 changes (if the condition holds); 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 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`, the new view state is `V' := Q(D')`: 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*

    → do 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

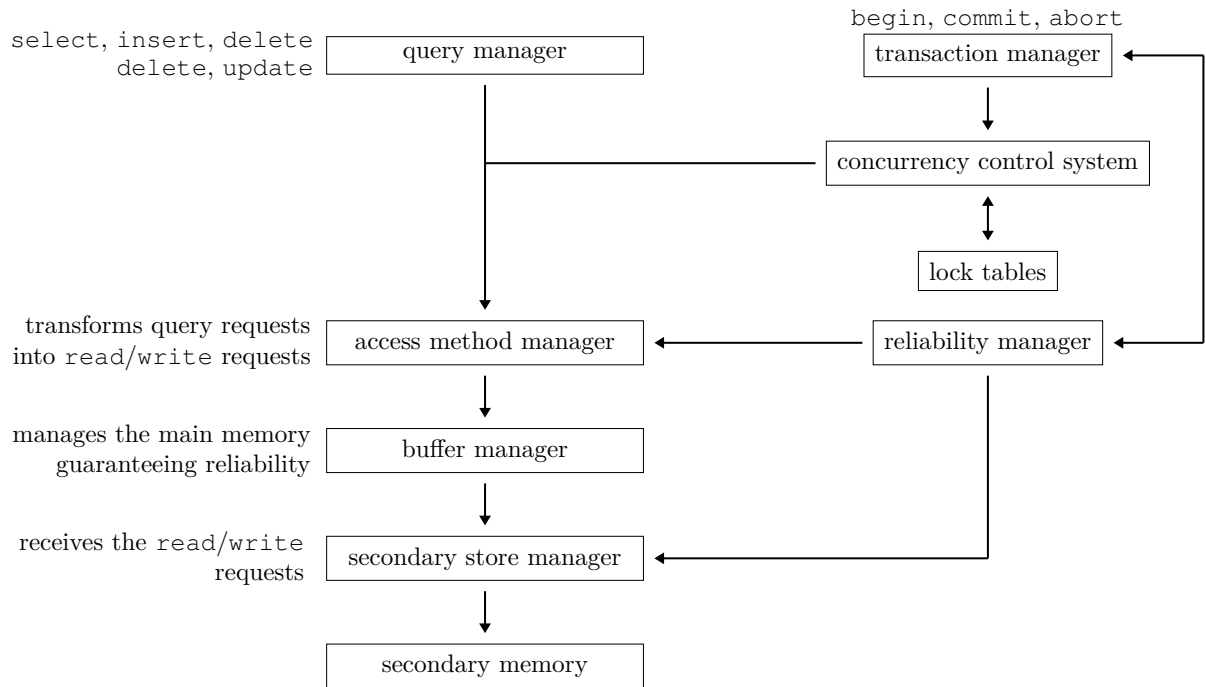A simple illustration of the physical database is shown in Figure 10.

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 *(4KB)*
- **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

- 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
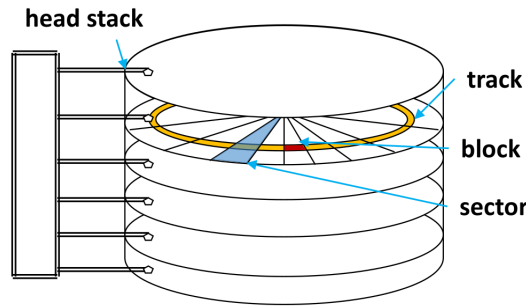- transfer time $0.1 - 1\,ms$, data transfer from the disk to the buffer

Figure 11: Structure of a mechanical hard drive

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.

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.

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**

Then the block factor is:

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

The rest of the space can be:

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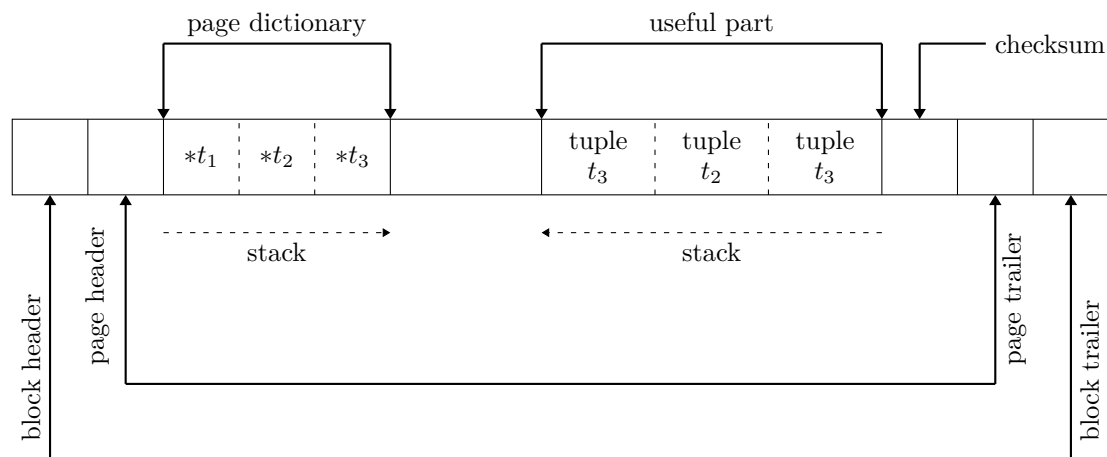- not used, if the records are unspanned

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*). They contain records in the form ⟨search key, pointer to block⟩; index entries are sorted with regard to the search key.

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 8.

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 8: 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 down the *DBMS*.

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: in order to access a tuple, the index is accessed first, 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: in order 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:

$$\text{search key } = \text{ ordering key}$$
$$\text{SK } = \text{ OK}$$

- 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:

$$\text{search key } \neq \text{ ordering key}$$
$$\text{SK } \neq \text{ OK}$$

- Multiple secondary indexes 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**

	primary	*secondary*
sequential	✔ (typical)	✖
hash-based	✔ (in some *DBMS*)	✔
tree-based	✖ (obsolete)	✔ (typical)

Table 9: Access structures

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 one or more optional 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 9.

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

In order 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 10 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*
`tuple size`	fixed or variable	fixed or variable
`INSERT`	✔	✖
`UPDATE`	✔	✖
`DELETE`	✖	✖
`SELECT * FROM T` `WHERE ...`	✖	✔

Table 10: 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 parts.

1. **folding**: transforms the key values to positive integers, uniformly distributed in a big range
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;
Techinques 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 $\beta = \frac{B}{\# \text{ of items}} = \dfrac{1}{\alpha}$, representing the average number of items per block

Remember that:

- 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 indexes: 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 then inefficient for access based on interval predicates or on 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 of binary search trees; the length of all the possible paths from the root node to the leaves is the same. 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 on 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

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, $0 \leq j \leq F$

5.4   Structures and Indexes

A comparison of indexing methods related to the data structures is shown in Table 11.

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 11: Comparison of the three indexing techniques

sub-tree that contain keys
$K_j \leq K \leq K_{j+1}$

sub-tree that contain keys
$K < K_1$

sub-tree that contain keys
$K \geq K_F$

(a) Internal node

pointer to block of primary storage
containing tuples with search-key K_i

pointer to block of primary storage
containing tuples with search-key K_1

pointer to the next leaf node

(b) Leaf node
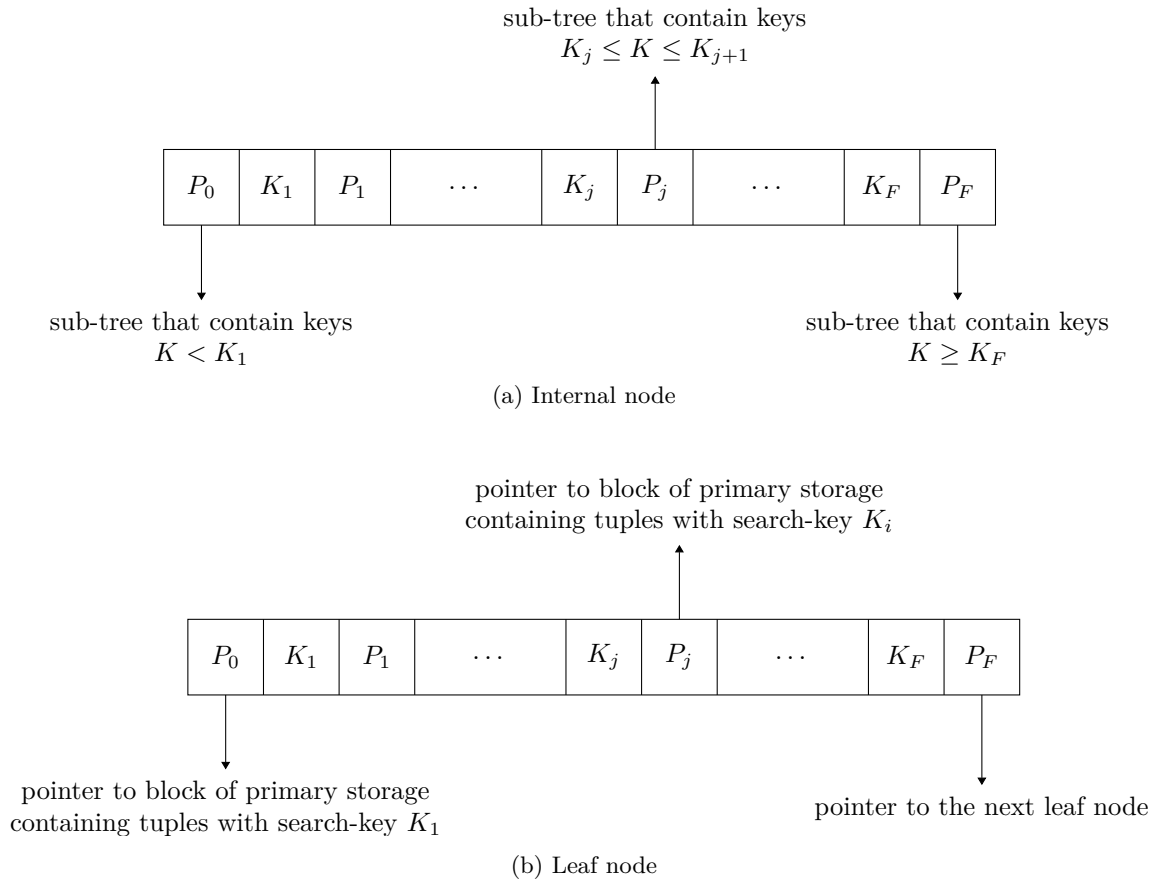
Figure 13: Structure of the nodes of a B+ tree

6 Tricky Exercises and how to slay them

6.1 Concurrency Control

6.1.1 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)*.

6.1.1.1 *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.

6.1.2 Update lock

A common notation to avoid ambiguity between Unlock and Update is to use the notation $\texttt{SL}_i(x)$ for a $\texttt{SL}$ 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 $\texttt{SL} \rightarrow \texttt{XL}$ is not allowed; the transaction needs to acquire a $\texttt{UL}$ lock first and then upgrade it to a $\texttt{XL}$ lock.