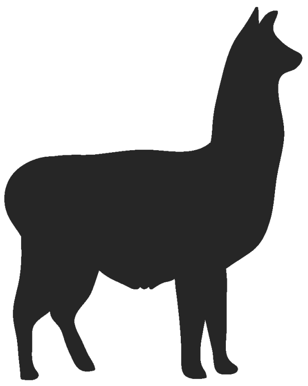
Data bases 2

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2022-09-20



no alpaca has been harmed while writing these notes

Contents

1 Introduction

1.1 Architecture of the DBMS

A **DBMS** 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

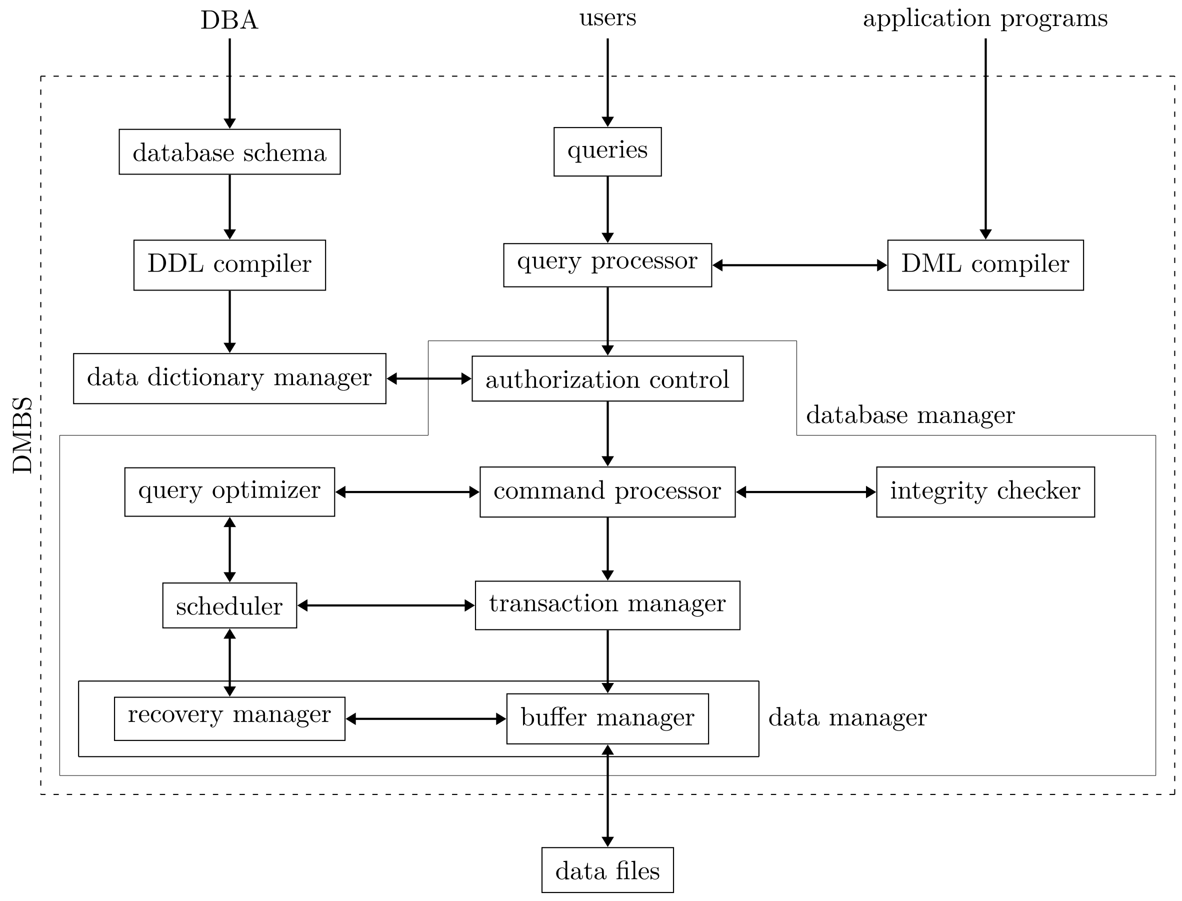


Figure 1: Architecture of the DBMS

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 kind 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 of 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 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 by 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 completely 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; intermediate state S_i may not be consistent
- **Isolation**
  - the execution of a transaction is independent of simultaneous execution of other transactions
  - the parallel execution of a set of transaction 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 Concurreny 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 transaction per second *(TPS)* managed by the DBMS. By exploiting the parallelism, the *TPS* can be increased.
The **Concurrency Control 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
  - a transaction writes data that has been read by another transaction that commits

3.1 Concurrency control theory

Model: an abstraction of a system, object of process, which purposely ignores some details in order to focus on the relevant aspects.
The concurrency theory builds upon a model of transaction and concurrency control principles that helps understanding real systems. Real systems 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 final outcome (either `commit` or `abort`).
- 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 respects 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 is denoted by $r_i(x)$
- A write operation 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 $\langle T_1, \dots, T_n \rangle$ each with k_i operations.
Then:

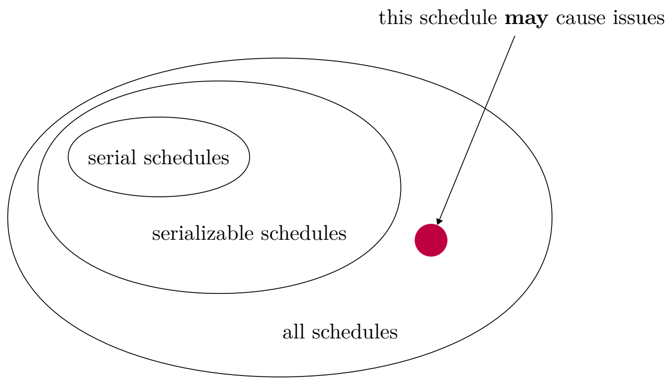


Figure 2: Schedules equivalence

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

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

3.1.3 Principles of Concurrency Control

The goal of the Concurrency Control is to reject schedules that cause anomalies. In order to enable the 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.
In order to identify classes of schedules that ensure the serializability, it's required to establish a notion of schedule equivalence. Let's make a two assumptions first:

1. 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
2. In practice, all schedulers must make decisions while the transactions are still running

Finally, it's possible to define:

- 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
- 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 reads-from relations
- 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 NP-complete problem, as it requires considering the reads-from and final writes of all possible serial schedule 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 actually 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 NP-complete problem. In order to test the conflict-serializability of a schedule, it's necessary to build a conflict graph (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.

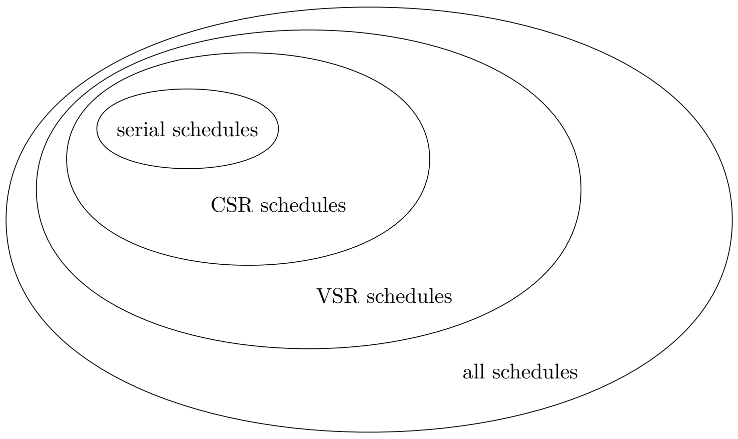


Figure 3: Relation between VSR and CSR

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 word, the concurrency control (CC) methods explained so fare 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 NP-complete, so it's not possible to solve it in 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 ads transactions can be aborted at any time. In order 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 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 an 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.

In order 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
- Optimistic
  - based on timestamping and versioning
  - serve as many requests as possible, possibly using out of date data

Both families of approaches will be compared later.

3.2.1 Locking

The concurrency control mechanisms implemented by most DBMSs is called **locking**. It works on a simple principle: a transaction can access a data item (either to write or read it) only if it has acquired a lock on it.
Three primitive operations are defined:

- `r_lock(`x`)`: acquire a read lock on x
- `w_lock(`x`)`: acquire a write lock on x
- `unlock(`x`)`: release the lock on x

The scheduler (or lock manager) 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).

3.2.1.1 Locks implementation

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

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`.

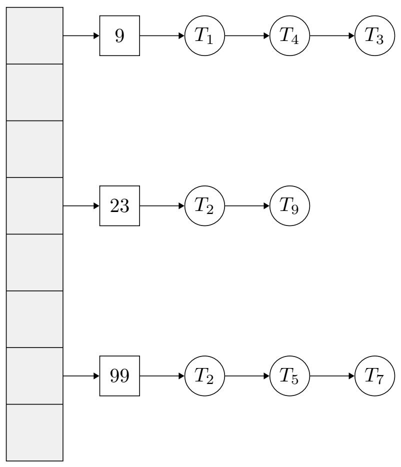


Figure 4: Illustration of a lock table

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.
In fact, 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

In order to avoid this situation, the locking method can be extended with a **two-phase locking** mechanism (2PL). 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 use in the two phases is shown in Figure 5.

Figure 5: Illustration of the resources use 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 the 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:

$$\mathit{VSR} \subset \mathit{CSR} \subset \mathit{2PL}$$

3.3.1 *2PL* implies *CSR*