POLITECNICO
DI TORINO

Recap Database Management System (01NVVOV)

Jacopo Nasi
Computer Engineer
Politecnico di Torino

I Period - 2017/2018

20 novembre 2017

Indice

License

Acknowledgments

Questo breve riepilogo non ha alcuno scopo se non quello di agevolare lo studio di me stesso, se vi fosse di aiuto siete liberi di usarlo.
Le fonti su cui mi sono basato sono quelle relative al corso offerto (**Database Management System (01NVVOV)**) dal Politecnico di Torino durante l'anno accademico 2017/2018.
Non mi assumo nessuna responsabilità in merito ad errori o qualsiasi altra cosa. Fatene buon uso!

1 Database Management System

1.1 Introduction

The DataBase Management System **DBMS** is a software package designed to store and manage databases. The architecture of the system is similar to the one in the figure 1. Since the DB data part can be really big it can't fit

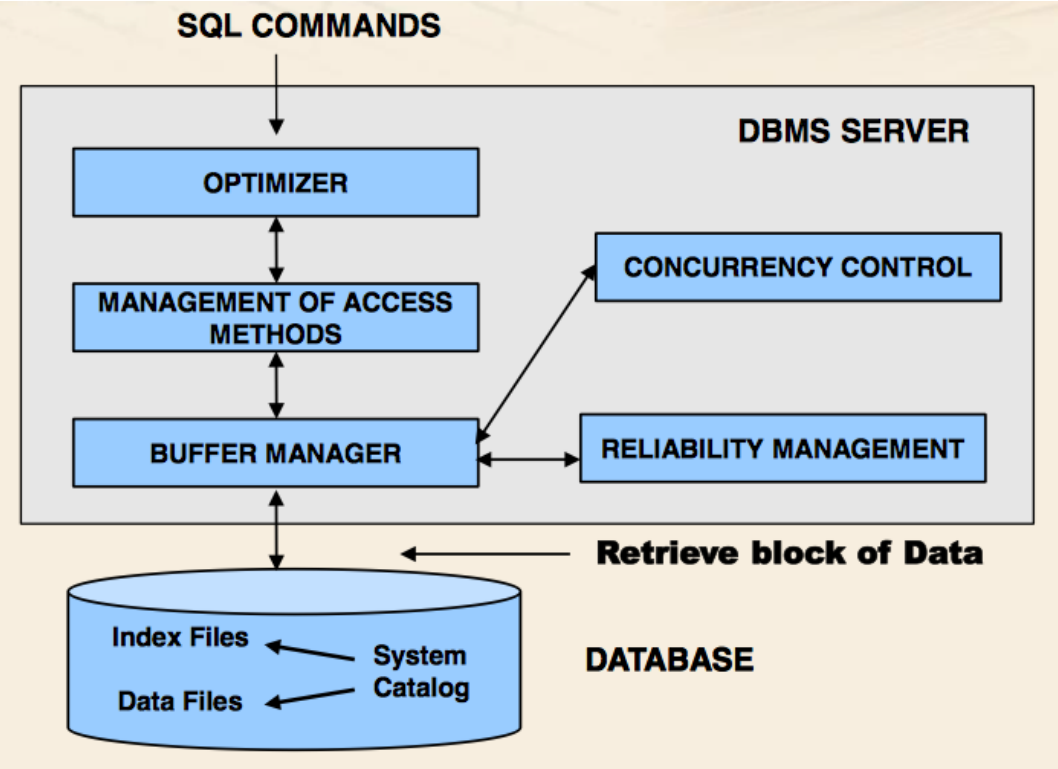

Figura 1: DBMS Architecture

always in the main memory (RAM) and, for this fact, is often stored in the secondary memory, like HDD. For this reason is necessary a system that define the operations to grab and manage the data from the secondary memory.
All the blocks has different behaviours. The **Optimizer** have multiple roles:

- Define an appropriate execution strategy for accessing data to answer queries.
- Receives in input the SQL instructions (DML).
- Check the lexical, syntactical and sematical correctness (not all the errors).
- Translate the query in an internal algebra rappresentation.
- Select the "right" strategy for accesing data.

- Guarantees the **data independence** property in the relation model.

The **Access Method Manager** is used for physical access to data and it implements the strategy selected by the optimizer. The **Buffer Manager** instead manage the page transfert from disk to main memory and vice versa and the main memory portion that is pre-allocated to the DBMS that is shared among many applications. The **Concurrency Control** coordinate the concurrent access to data (important for write operations) to guarantess the consistency of it. The **Realiability Manager** guarantees correctness of the database content duing the system crashes, the atomic execution of a transaction and it exploits auxiliary structures (log files) the correct the database in case of failure.
The **transaction** is an unit of work performed by an application, it's a sequence of one or more SQL RW operation charaterized by *correctness, reliability* and *isolation*. The START of a transaction is typically implicit and coincides with the first SQL instruction. The END instead can be of two differents types, it can be a COMMIT that it means the correct end of a transaction, or with ROLLBACK that it means error during the execution. In this second case the DBMS needs to go back to the state at the beginning of the transaction. The rollback can be of two type suicide, when is required by the transaction, and murder when is required by the system. The transaction have four important properties:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Atomicity means that they cannot be divided in smaller units, is not possibile to leave the system in a intermediate state of exec, guarantee by UNDO (undoes all the work perfomed, used for rollback) and REDO (redoes all work performed, used for commit the result in presence of failure). The consistency means that the transaction execution should not violate integrity constraints on a database, in case of it the system will perform solution to correct the violation. The system can be considered Isolated when the execution of a transaction is indipendent of the concurrent execution of other transaction, everything is enforced by the Concurrency Control block. The last properties means that, in presence of failures, the effect of a commited transaction IS NOT LOST, it guarantees the reliability of the DBMS and is enforced by the Reliability Manager block.

1.2 Buffer Manager

This block have a real important behaviour, it manages page transfer from disk to main memory and it's in charge of managing the DBMS buffer. The

operation of the pages trasnfert is the bottleneck of every system and this is why this block is really important. increasing the performance of this operation could really improve the speed of the entire system.
The buffer is:

- A large main memory block.
- Pre-allocated to the DBMS.
- Shared among executing transactions.

this part is organized in pages where the size depends on the size of the OS I/O block. There are two empirical law often used for hte management strategies:

1. Data Locality: Data referenced recntly is likely to be referenced again.
2. 20-80: The 20% of data is RW by 80% of transaction.

The buffer manager keeps additional snapshot information on the current content of the buffer, it shot, for every page, the physical location of the page on the secondary memory (file identifier and block number) and two state variables, one that count of the number of trasn using the page in that time (count), and the dirty bit that is set if the page has been modified.
It provides different access methods to load pages from disk and vice versa:

Fix Primitive used by transactions to require access to a disk page, after the page is loaded into the buffer a pointer is returned to the requesting transaction and the Count is incremented by 1. This procedure requires an I/O operation only id the page is not already in the buffer. There are two behaviour:

- Page already in buffer: Return the pointer to the data.
- Page not in buffer: It search a place for the page.
    1. Free pages
    2. Not free pages, Count=0; if the data is dirty it performs a synchronous write on the disk.

Unfix Primitive it tells the buffer manager that the transaction is no longer using the page and it decrease the Count.

Set Dirty Primitive is tells the buffer manager that the page has been modified by the running transaction and it sets the dirty bit to 1.

Force Primitive it requires a synchronous transfert of the page to the disk, when this operation is performed the transaction is suspended.

Flush Primitive is an autonomous transfert of the pages on the disks, is internal to the buffer manager and is runned when the CPU is not too much loaded. It transfer the page that are not valid (count=0) or not accessed since long time.
The are four writing strategies:

- **Steal**: The BM is allowed to select a locked page with Count=0 as victim. It writes on disk the dirty pages belonging to uncommitted trans. It can be undone.
- **No Steal**: The BM is not allow to steal.
- **Force**: All the pages are synchronous written on the disk during the commit operation.
- **No Force**: The pages are written asynchronously with the Flush Primitive.

The mostly used solution is **steal/no force** because of its efficency. The no force provides better I/O performance, steal may be mandatory for queries accessing a very large number of pages.

File System the BM is using services provided by the file system:

- Create/Delete of a file.
- Open/Close file.
- Read: It provides a direct access to a block in a file and it requires File Identifier, Block number and buffer page where to save data.
- Sequential Read: It provides seq. access to a fixed number of blocks in a file, it requires file identifier, strating block, number of blocks to be readed and the starting page for saving.
- Write and Sequential Write.
- Directory management.

1.3 Physical Access

Data may be stored in different format to provide efficient query execution. The **Access Method Manager** transform the decision taken by the optimizer into sequence of physical access to data. An access method is a software module specialized for single data structure that provide primitives for read and write. The AM can select the appropriate blocks of a file to be loaded in memory and it knows the organization of data into a page.
There are several solution for manage the data in relational system:

- Physical data storage
  - Sequential Structure
  - Hash Structure
- Indexing
  - Tree Structure
  - Unclustered Hash Index
  - Bitmap Index

In the sequential solution the tuples are stored in a given sequential order, in the case of the heap file are sorted in the insertion order, typically append at the end of the file.

- **PRO**: No wasted space, sequential read/write fast.
- **CONS**: Delete may cause wasted space.

this structure are frequently used jointly with unclustered indices to support search and sort operations.
In the ordered structures everything is sorted by the value of a given key, called sort key, it can contain one or more attributes.

- **PRO**: Sort, group by, search or join operations on the sort key really fast.
- **CONS**: Inserting new value preserving order.

the main problem of this solution is to keep the order of the data during new data insertion. There are two main solution, the first il leaving a percentage of free space in each block during the table creation; the second one create an overflow file containing tuples which do not fit into the correct block.
The ordered structure are typically used with B^+-Tree clustered (primary) indices where the index key is the sort key. Are used bt the DBMS too to storing intermediate operation results. This structure provide "direct" access to data based on a key (one or more attributes). This Tree have one root node with many intermediate nodes and each node has many children. The leaf nodes provide access to data in 2 different ways:

- **Clustered**: It store the data in the main memory. Used for primary key indexing. [figure 2]
- **Unclustered**: It store a pointer to the secondary memory of the data. Used for secondary indices. [figure 3]

There are two kind of B-Tree:

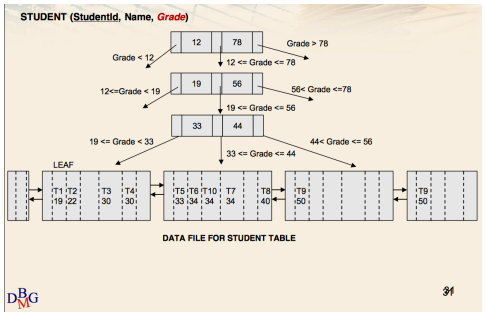

Figura 2: Clustered

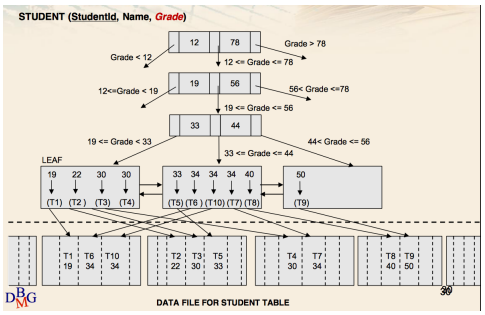

Figura 3: Unclustered

- **B-Tree**: Data pages are reached only through key values by visiting the tree. [figure 4]
- B^+**-Tree**: Provides link leaf allowing sequential access in the sort order. [figure 5]

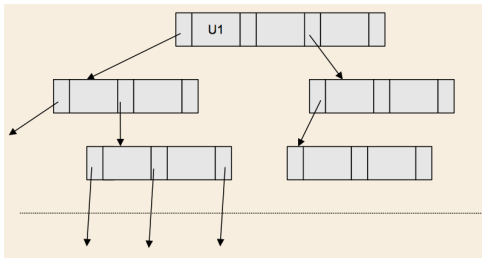

Figura 4: B-Tree

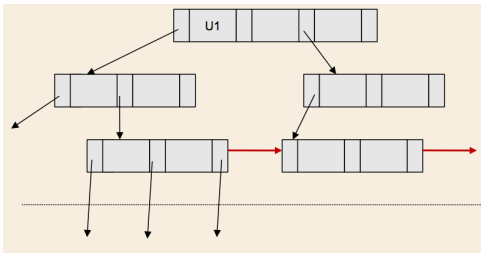

Figura 5: B^+-Tree

the B stands for **Balanced** where leaves are all at the same distance from the root and the search time is the same indipendently by the value.
This structure have some:

- **Advantages**:
  - Very efficent for range queries.
  - Appropriate for sequential scan in the order of the key field (always for clustered, not guarantee otherwise).
- **Disadvantage**:
  - Insertion may require a leaf or nodes split.
  - Deletions may require merging uncrowded nodes and re-balancing.

The **Hash** structure is another kind of weel-know structure is guarantees direct and efficent access to data based on the value of a key field (one or more attributes). Supposing to have B blocks in the hash structure the hash function is applied to the key value of a record and in return a values between

0 and b-1 which defines the position of the record, the idea is to not completly fill the blocks to allows new data insertion.

- **Advantages**:
  - Very efficent for queries with equality predicate on the key.
  - No sorting of disk blocks is required.
- **Disadvantage**:
  - Inefficent for range queries.
  - Collision may occur.

The unclustered versione is similar to the hash index, the main difference is that the actual data is stored in a separate structure and the position of tuples is not constrained to a block.

The **bitmap index** is another structure that provides direct and efficient access to data based on the value of a key field, it's based on a bit matrix. The bit matrix references data rows by means of RIDs (Rows IDentifiers), the actual data is stored in a separate structure and the tuples position is not constrained.
The bit matrix has:

- One column for each different value of the indexed attribute
- One row for each tuple.

the *(i, j)* position has a 1 if the tuple i as j like attributes for the key field, 0 otherwise. the main characteristics are:

- **Advantages**:
  - Very efficent for boolean expressions of predicates.
  - Appropriate for attributes with limited domain cardinality.
- **Disadvantage**:
  - Not used for continuous attributes.
  - Required space grows significantly with domain cardinality.

1.4 Query Optimization

The query optimizer is part of the Optimizer and its job is selecting an efficient strategy for query execution, this block is really important. Another important task is to guarantees the data independence property, in fact, the form in which the SQL query is written does not affect the way in which it is

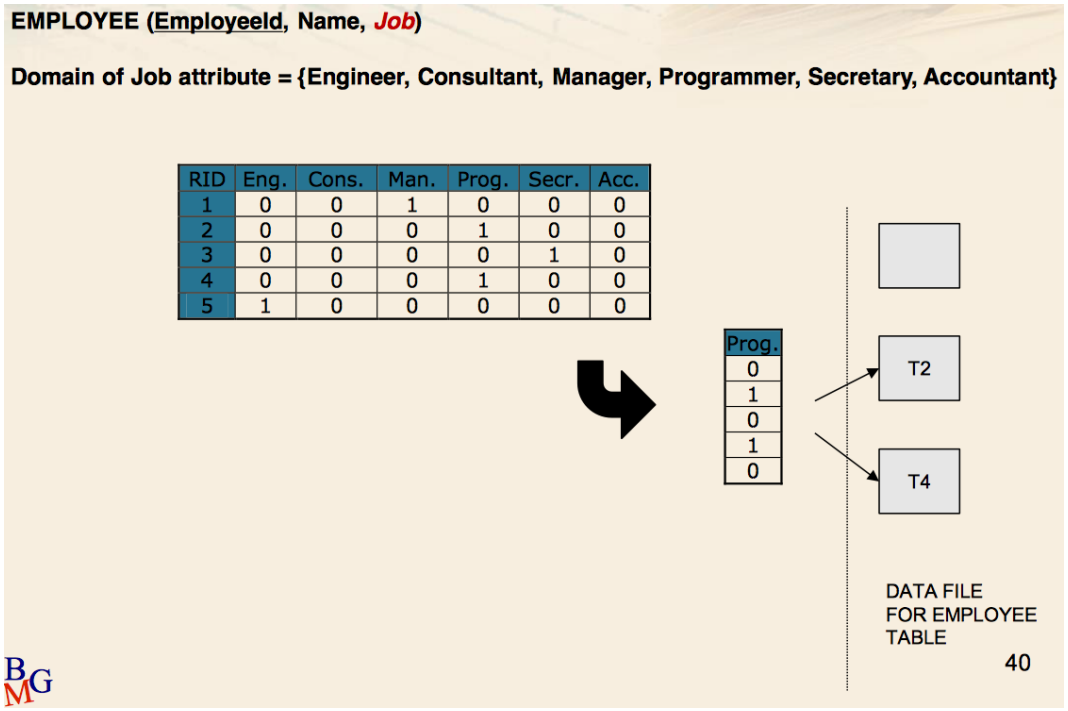

Figura 6: Bitmap Index

implemented and a physical reorganization of data does not require rewriting SQL queries.
The query optimizer generates a **Query Execution Plan** to use the best strategies to run the query, it evaluates many different alternative, it use data statistics, use best-know strategies and it adapts automatically on data changes. The plan has more phases as you can see in figure 7. The behaviour of each phase is:

Lexical, Synctactic and Semantic analysis : check the SQL for Lexical errors (e.g., misspelled keywords), Syntactical errors in the SQL grammar and for Semantic erros not existing object called in the query (require data dictionary). The output of this block is an internal rappresentation of extended relational algebra because it can represent the order in which operators are applied (procedural) and there are a lot of theorems and properties.

Algebraic Optimization : executing algebraic transformations is considered to be always beneficial, it should eliminate the difference among different formulations of the same query and is usually independet of the data distribution. The output of this phase is a "canonical" tree.

Cost Based Optimization : This phase select the best execution plan evaluating the execution cost, it use a selection of:

- Best access method for each table.
- Best algorithm for each relational operator among available alternatives.

the last step of this phase is the generation of the code implementing the best strategies, the output is the executable and all the dependencies used.
There are two types of execution modes:

- **Compile and Go**: Compilation and immediate execution, no storage of query plan and no need of dependencies.
- **Compile and Store**: The plan is stored in the DB together with its dependencies, it's executed on demand and it need to be recompiled in data structure changes.

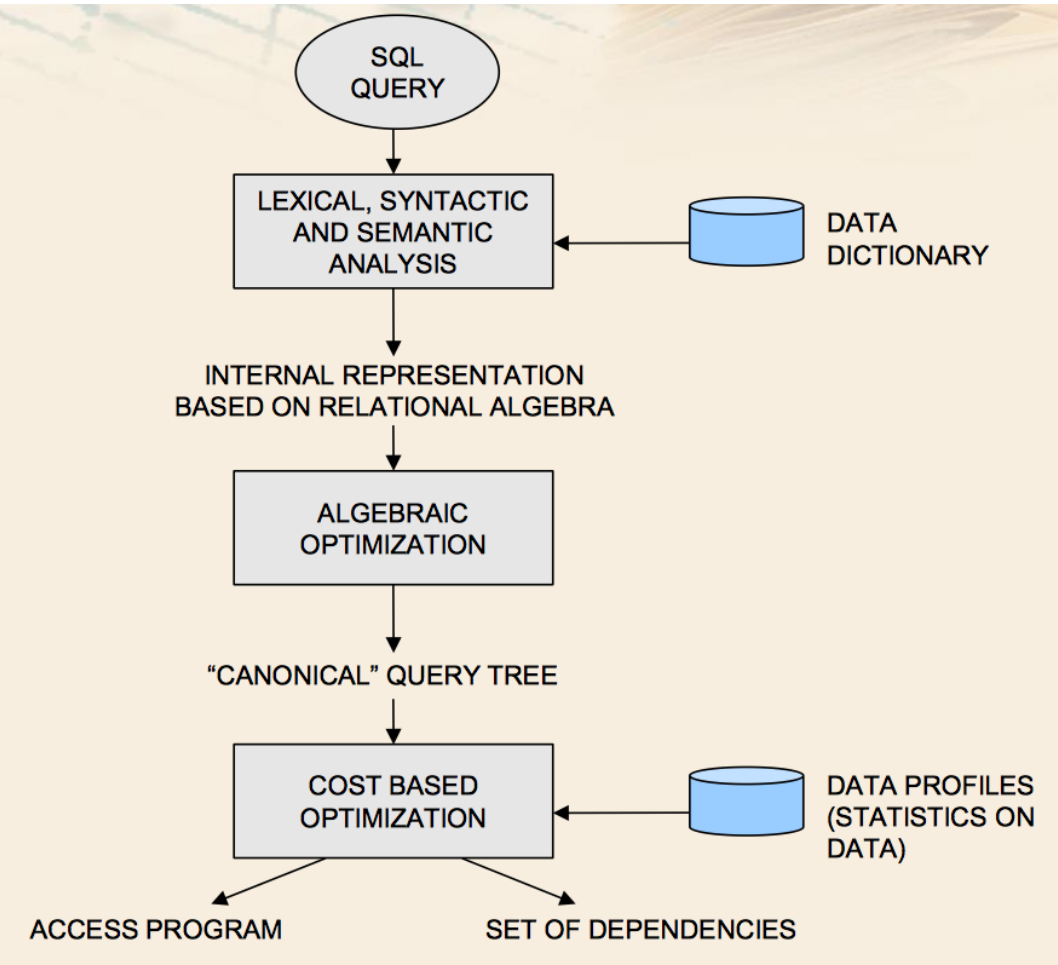

Figura 7: Optimization Execution Plan

The phase of **Algebraic Optimization** require a little more of analysis. The part is bbased on equivanlence trasformations, *two relational expressions are equivalent if they both produce the same query result for any arbitrary database instance.* The main objective of this part is to **reduce the size of the intermediate result**.
There are some well-know transformation:

1. **Atomization of selection**: Applying all attributes of selection one at the time or all together can provide different performance in case of indicies.

2. **Cascading Projections**: It possibile to perform directly the final projection o doing more projections with different sub-set and you can obtain the same result.

3. **Selection before join**: Anticipating the selection respect a join-operation can reduce the cardinality of the system reducing the number of operations for the join (is always used by the DBMS).

4. **Join derivation from Cartesian Product**: Perform a cartesian product and then a selection over data get the same result as the join operation but more slowly.

5. **Distributing selection respect union**: Is equivalent to select a subset and then merge it with another subset or merge the two set and then selecting it.

6. **Distributing selection respect difference**: The formulas explain better: $\sigma_F(E_1 - E_2) = \sigma_F(E_1) - \sigma_F(E_2) = \sigma_F(E_1) - E_2$.

7. **Distributing projection respect union**: Projection of union of 2 table is equivalent to the union of 2 already projected table.

8. **Other**: $\sigma_{F1 \vee F2}(E) = (\sigma_{F1}(E)) \cup (\sigma_{F2}(E))$

9. **Other**: $\sigma_{F1 \wedge F2}(E) = (\sigma_{F1}(E)) \cap (\sigma_{F2}(E))$

10. **Distributing join respect union**: Perform join of a table E with two merged table (E1 $\cup$ E2), is equivalent to join E with E1 and E2 separate and the merge it.

The phase of **cost based optimization** is a little bit more complicated, it is based on:

- **Data Profiles**: Statistical information describing data distribution for tables and intermediate relational expression.

- **Approximate cost**: Evaluating cost by looking at CPU, HDD and main memory usage and time.

The profilings of table save quantitavive information on the characteristics of tables and columns:

- Cardinality of tuples.
- Size in bytes of tuples.
- Size in bytes of each attributes.
- Number of dinstinct values of each attribute.
- Min and max value of each attribute.

all this information are stored in the data dictionary that is periodically refreshed.

The access operators can perform different types of scans. The **sequential scan** execute sequential access to all tuples in a table (a.k.a Full Table Scan). The operation performed during a sequential scan:

- Projection.
- Selection (Simple predicate).
- Sorting based on attribute list (memory sort or sort on disk).
- Insert, Update or Delete.

the predicate evaluation is fundamental to provide an efficent access to the data. The index access it may be exploit with all kind of structures, in case of simple equality predicate all structure are appropriate. Instead, for range predicate, the only appropriate one is the B^+-Tree. For predicates with limited selectivity full scan is better (if available bitmap could be used). In case of conjunction fo predicates the most selective one is evaluated first through the index, then the other. A possible optimization could be computing the intersection of bitmaps coming from available indices and then a table read for remaing predicates. In the disjunction the index access can be used only if all predicates have and usable index, otherwise FTS.

The **join** operation can be a critical operation for a relational DBMS, the connection between tables is based on values instead of pointer. There are several algorithms that can be used for the join:

- **Nested Loop**: For each tuples of the outer table, the inner one is readed once. (BRUTE FROCE)
  - Efficent when the inner tavle is small and fits in memory or when the join attribute in the inner table is indexed.

  - Not cost symmertric. It depends on which table takes the role of inner.

- **Merge Scan**: It sort the two tables on the join attribute and it start a parallel scan.
  - Symmetric in terms of cost. Efficent for large and already-sorted tables.
  - Requires sorting both table (already sorted or trough clustered index).

- **Hash Join**: It applicate the same hash function to the join attrbute of both table. Tuples to be joined will fill the same bucket.
  - Very fast join.
  - Local sort and join is performed into each bucket.

- **Bitmapped Join Index**: It precompute the join. The position (i,j) of the matrix is 1 if the tuple with RID j of A joins with tuple with RID i in table B and 0 otherwise.
  - A data change need a recompute of table.
  - Used in OLAP queries.
  - It can exploit one or more bitmapped join indices (one for each pair of joined tables) and accessing the large central table is the last step.

The **Group By** is one of the most important function of SQL and is performed in 2 different ways: The first one is the sort based, it sort the data on the group by attributes and the compute aggregate functions on groups; the hash based one instead it perform and hash function over data, sort the bucket just created and than compute the aggregate function.

Execution plan selection is based on some data input, the data profiles (statistics over data) and the internal representation of the query tree, the output of this part is the "optimal" (it can't assure that it will be the best one) execution plan. This phase evaluate the cost of different alternatives for read tables and executing each relational operator exploting an approximate cost of execution.
The search is based on the following parameters:

- Scan type of data (full scan, index).
- Execution order among operators.
- Type of operators implementation (different join methods).

- Sorting time (when).

The approach work on a tree of alternatives where each nodes represent a decision on a variable and the leaf one complete query execution plan. Of course the system select the cheapest one. The general formula is $C_{Total} = C_{I/O} * n_{I/O} + C_{CPU} * n_{CPU}$ where $n_{I/O}$ is the number of I/O operations and n_{CPU} is the number of CPU operations.
The final plan is an approximation of the best solution. Th optimizer looks for a solution which is of the same order of magnitude of the "best" solution. In the **Compile and Go** execution mode the search is stopped when the time spend for the search is comparable to the time required to execute the current best plan.

1.5 Physical Design

The physical distribution of the data in the system is fundamental for providing good performance. Taking in account the logical schema of the DB, the features of the selected DBMS and thw workload this block provides a physical schema of the databse (table organization, indices) and all necessary set up parameters for storage and configurations.
The possible physical file organization are:

- Unordered (heap).
- Ordered (clustered).
- Hashing on hash-key.
- Clustering several relations.

The number of indicies is related to the structure type of the system. In case of clustered is possible to define only one index, instead, unclustered structures allow to define multiple different indices.
The workload distribution is different in case of a normal query or for an update. The first case involve:

- Accessed tables.
- Visualized Attributes.
- Attributes involve in selections and joins.
- Selectivity of selections.

the update case instead:

- Attributes and tables involved in selections.
- Selectivity of selections.

- Update type (Ins/Del/Up) and updated attributes.

The selection of the structure is important and it could be changed during the usage of the system for improvement (database tuning). Changes in the logical schema are allowed and they can or cannot preserve the BCNF (Boyce Code Normal Form). There isn't a general methodology the best solutions are trial and error, some general criteria and "common sense" heuristic.
Some general criteria could be:

- Primary Key is usually exploited for selection and joins, indexing it could be useful.
- Adding new indices for most common query predicates. Evaluate the actual plan and verify the improvement if available.
- Never index small table, the entire table requires few disk reads.
- Never index attributes with low cardinality (e.g. gender). This is not true for data warehouses.

from the heuristics point of view there are severals "common sense" ideas:

- For attributes involved in simple predicates of where clause equality:hash and range:B^+-Tree.
- Evaluated clustered improvement for slow queries.
- For where clauses involving many simple predicates use multi attributes index or appropriate key order.
- Maintenance cost.
- To improve joins use index on inner table in case of nested loop or B^+-Tree, for merge scan, on the join attribute.
- For group by hash index or B^+-Tree.
- Consider group by push down that anticipate the group respect to joins.

of course after all the changes a good choice could be update database statistics, for future improvements the database tuning could help. The last chance can be affecting optimizer decision, the main problem is the lost of data independence.

1.6 Concurrency Control

The workload of operational DBMS is measured in *transaction per second* (banking and flight reservation are on 10-1000 tps). This block provide concurrent access to data maximizing the throughput and minimizing response time. The elementary operations are of course **READ r(x)** and **WRITE w(x)**. The block that manage the concurrency is called scheduler is in charge of deciding if and when read/write request can be satisfied.
The most common anomalies are:

- **Lost Update**: It occur when a tr2 read a value that is already under operations by another tr. (figure 8)
- **Dirty Read**: When a tr2 read the value of x in an intermediate state which never become permanent. (figure 9)
- **Inconsistent Read**: When a tr1 read multiples times x with different value each time. (figure 10)
- **Ghost Update**: It occur when two transaction are working over multiple data at the same time performing read and write. (figure)

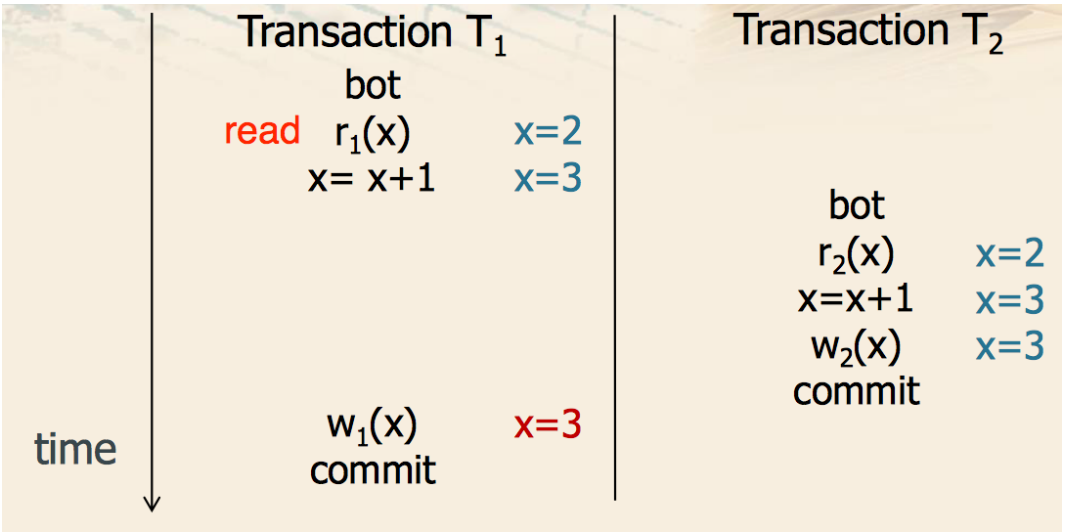

Figura 8: Lost Update Behaviour

Theory of Control a *transaction* is a sequence of R/W operations with the same TID (*Transaction Identifier*); the *schedule* is a sequence of read/write operations presented by concurrent transaction. The scheduler is in charge of accepts or reject the requests to avoid anomalies without knowing the outcome (commit/abort) of it.

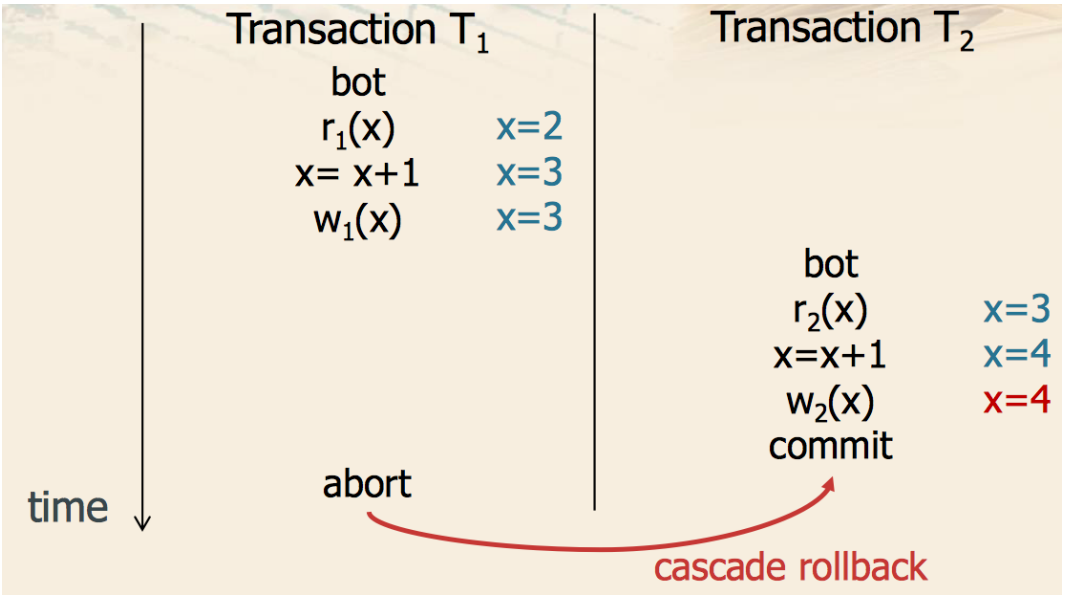

Figura 9: Dirty Read Behaviour

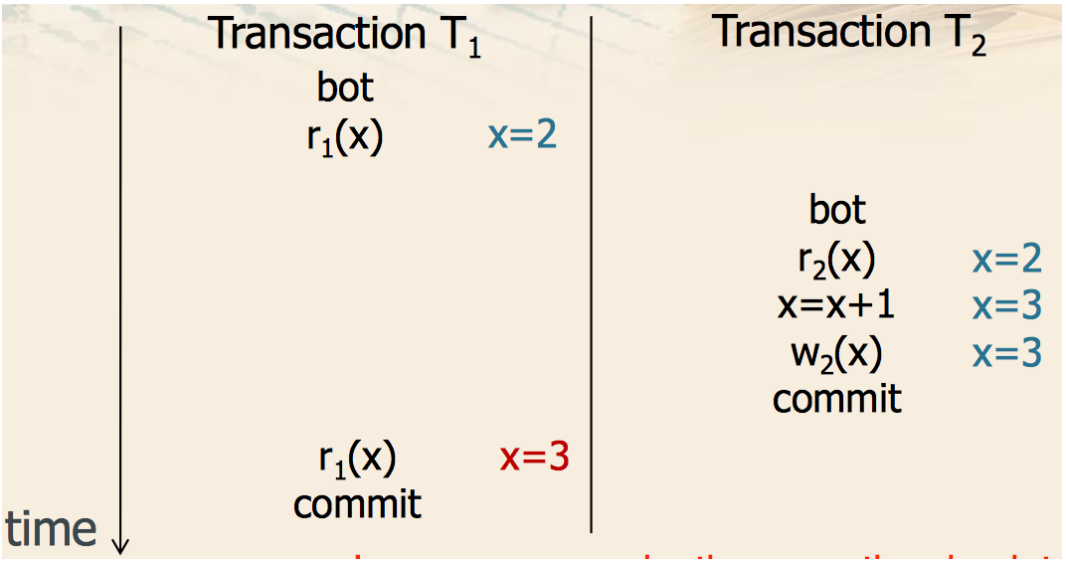

Figura 10: Inconsistent Read Behaviour

Commit projection is a simplifying hypothesis (the schedule only contains transaction performing commit), it avoid dirty read anomaly, it will be removed later.
In a **serial schedule**, the actions of each transaction appear in sequence, without interleaved actions. An arbitrary schedule S_i is correct when it yields the same results as an arbitrary serial schedule S_j of the same transactions. S_i is serializable, is equivalent to an arbitrary serial schedule of the same transaction. There are different equivalence classes between two schedules:

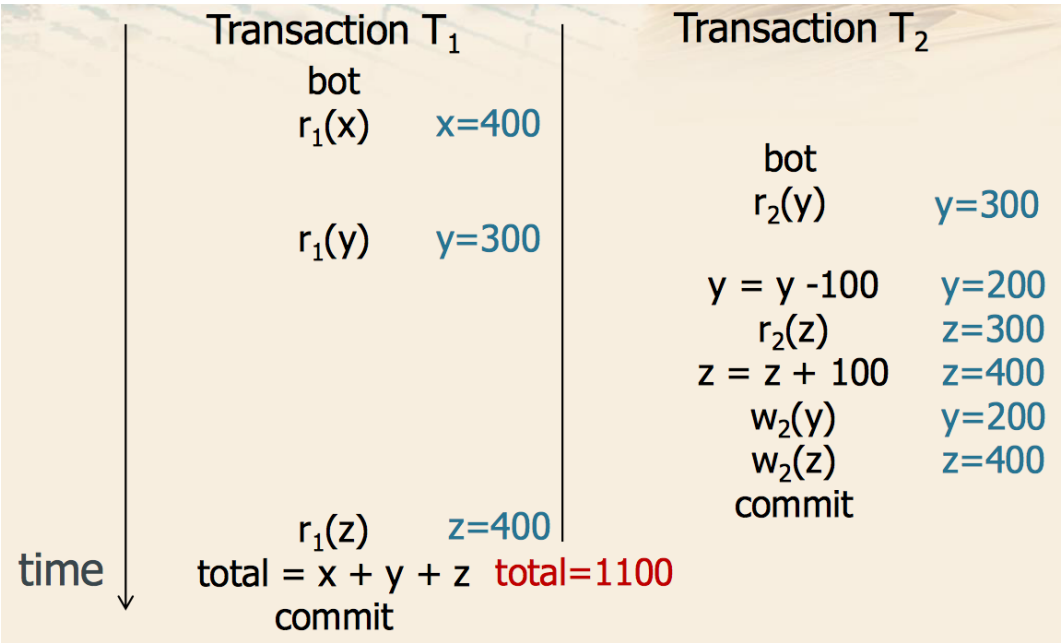

Figura 11: Ghost Update Behaviour

- View equivalence
- Conflict equivalence
- 2 phase locking
- Timestamp equivalence

each equivalence class find a set of acceptable schedules characterized by a different complexity.

View equivalence there some definitions to be introduced:

- **reads-from**: $r_i(x)$ reads-from $w_j(x)$ when:
  - $w_j(x)$ precedes $r_i(x)$ and $i \neq j$
  - There is no other $w_k(x)$ between them.
- **final write**: is a final write if it is the last write of x appearing in the schedule.

with this solution two schedules are view equivalent if they have the same reads-from set or the same final write set.
This techniques is easy to be understand using an example. Using the flow in figure 12. The corrisponding schedule is: $S = r_1(x)r_2(x)w_2(x)w_1(x)$ is this

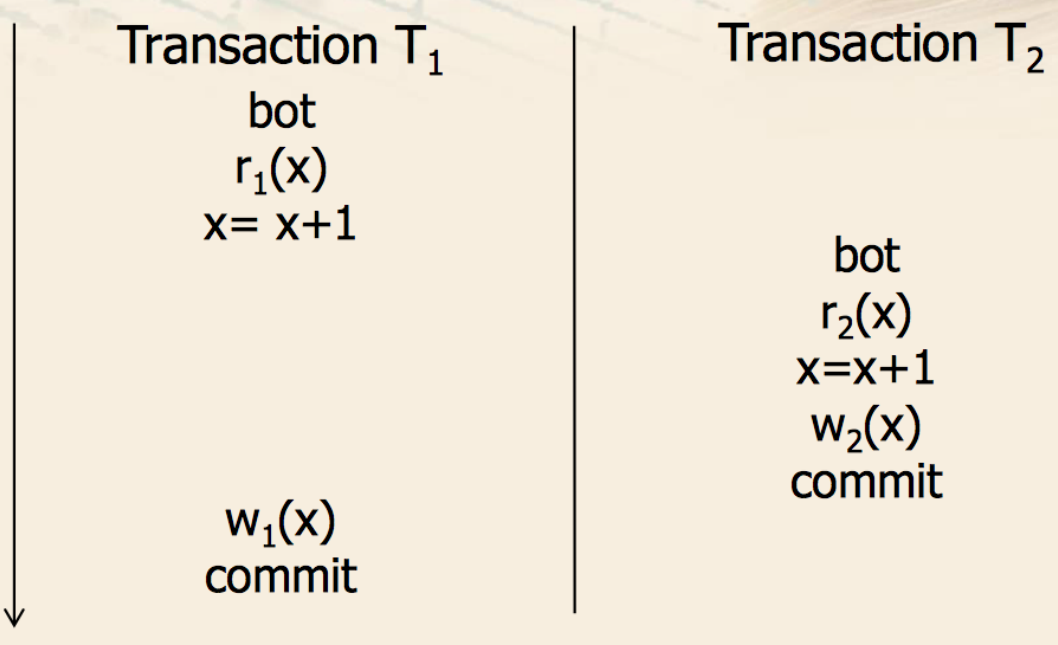

Figura 12: View Equivalence Example

schedule serializable?
There are only 2 possible serial schedules:

$$\begin{aligned} S_1 &= r_1(x)w_1(x)r_2(x)w_2(x) \\ S_2 &= r_2(x)w_2(x)r_1(x)w_1(x) \end{aligned} \tag{1}$$

In both cases S is not view equivalent to any serial schedule, not serializable, should be rejected.
The analization of this problem is linear if the schedule is given, in case of an arbitrary schedule it become NP-complete. For this problem is better to use a less accurate, but faster techniques.

Conflict equivalence Two actions are in conflict when both operate on the same object and at least one of them is a write. The conflict can be, RW, WR or WW. The conflict are equivalent if they have:

- Same conflict set.
- Each conflict pair is in the same order in both schedules.

A schedule is conflict serializable (**CSR**) if it is equivalent to an arbitrary serial schedule of the same transaction.
Dectecting the conflict serializability it is possible to exploit the conflict graph:

- Node: Each transaction.
- Edge i→j: if is there at least a conflict between T_i and T_j.

checking the cyclicity of a graph is linear in the size of the graph.

2 Phase Locking a lock is block on a resource which may prevent access to others. The operations are:

- **Lock**
  - Read Lock (R-Lock)
  - Write Lock (W-Lock)
- **Unlock**

Each operation is preceded by a request of R/W-Lock and is followed by a request of unlock. Of course the R-Lock is shared among different transaction, the write lock instead is exclusive, not compatible with any other lock.
The scheduler becomes a lock manager it receives transaction requests and grants locks based on locks already granted and so on...
When the lock request id **granted**:

- The resource is acquired by the requesting transaction.
- After the unlock, the resource, becomes again available.

When the lock is **not granted**:

- The requesting transaction is put in a waiting state.
- The wait is terminated when the resources is unlocked and becomes available again.

All the information for grant or not the lock are stored in the **lock table** with the **conflict table** (figure 13) used to manage lock conflicts.

The read locks are shared this is because the read not change the state of a data and multiple access can be exploited at the same time. A counter is used to count the number of R-Lock granted on a resources.
The lock manager, that coordinates the grant, exploit the lock table stored in main memory and, for each data object, use 2 bits to represent the 3 possible states (free, r-locked, w-locked) and a counter to count the number of waiting transaction. An example in figure 14.

There is also another version of the 2 phase locking the STRICT version. In this solution the unlock will be performed only at the end o the transaction and not when the resources is released; this difference guarantee to avoid the dirty read anomaly.
The procedure is really fast. If a transaction keep in wait over timeout the lock manager resume it and returns a NOT OK ERROR, the requesting transaction may perform a rollback o request again the same resources.

Request	Resource State		
	Free	R-Locked	W-Locked
R-Lock	Ok/R-Locked	Ok/R-Locked	No/W-Locked
W-Lock	Ok/W-Locked	No/R-Locked	No/W-Locked
Unlock	Error	Ok/It depends (free if no other R-Locked)	Ok/Free

44

Figura 13: Conflict Table

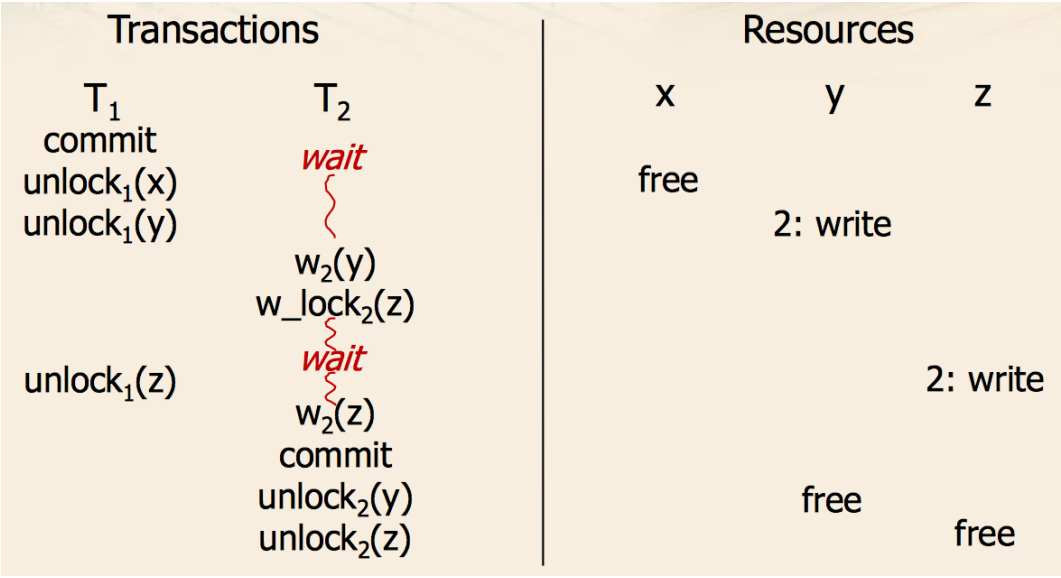

Figura 14: Conflict Table

Hierarchical Locking locks tables at different granularity levels:

- Table
- Group of Tuples (fragment)
- Single Tuple

- Single Field in a Tuple

this is an extension of the traditional locking. It allows a transaction to request a lock at the appropriate level of the hierarchy and it is characterized by a large set of locking primitives.
The **Locking Primitives** are:

- **Shared Lock (SL)**
- **eXclusive Lock (XL)**
- **Intention of Shared Lock (ISL)**: It shows the intention of shared locking on an object which is in a lower node in the hierarchy.
- **Intention of eXclusive Lock (IXL)**: Similar to ISL, but for exclusive lock.
- **Shared lock and Intetion of eXclusive Lock (SIXL)**: Shared lock of the current object and intention of exclusive lock for one or more objects in a descandant node.

The behavior is reported in figure 15.

	Resource State				
Request	**ISL**	**IXL**	**SL**	**SIXL**	**XL**
ISL	Ok	Ok	Ok	Ok	No
IXL	Ok	Ok	No	No	No
SL	Ok	No	Ok	No	No
SIXL	Ok	No	No	No	No
XL	No	No	No	No	No

66

Figura 15: Hierarchical Behavior

The selection of lock granularity depends on the application type:

- If it performs localized reads or updates of few objects (low lv - detailed ganularity).

- If it performs massive reads or updates (high lv - rough granularity).

the effect of lock granularity:

- If it is too coarse, reduces concurrency.
- If it is too fine, it forces significant overhead on the lock manager.

The predicate locking addresses the ghost update of type b (insert) anomaly, for the 2PL a read operation is not in conflict with the insert of new tuple, they can't be locked in advance. The PredLock allows locking all data satisfying a given predicate.

There are several isolation level:

- **SERIALIZABLE**:
  - Highest
  - PredLocking
- **REPEATABLE READ**:
  - Strict 2PL without PredLock
  - Read existing obj can be correctly repeated
  - No protection ghost update
- **READ COMMITTED**:
  - Not 2PL
  - The read lock is released as soon as the object is read
  - Reading intermediate states of a transaction is avoided (dirty reads)
- **READ UNCOMMITTED**:
  - Not 2PL
  - Data reads without acquiring the lock
  - Only allowed for read only transaction

Deadlocks can be frequent, the solution for solving it are implementing **timeout** the transaction waits for a given time, after the expiration of TO it receives a negative answer and it performs rollback. The time length could be LONG and SHORT (can be overloads the system). Other solution can be performed Pessimistic 2PL that acquire all locks before the transactions start (not always feasible) or the timestamp that put in wait mode only younger transaction. The deadlocks detection is performed using the wait graph, it could be expensive to build and mantain.

1.7 Reliability Management

It is responsible of the atomicity and durability ACID properties, it implements the following transactional commands:

- BEGIN transaction (B)
- COMMIT (C)
- ROLLBACK (A, for Abort)

it also provides recovery primitives WARM and COLD RESTART.
It manages the reliability of R/W requests by interacting with the buffer manager, it may generate new R/W requests for reliability purposes. It exploits the **log file** a persistent archive recording DBMS activity and is stored in a "stable" memory (not affected by failure, abstraction). It prepares data for performing recovery by means of the operations *checkpoint* and *dump*.

Log file is a sequential file written in stable memory that records transaction activities in chronological order. The record can be related to a transaction or system, they are saved interleaved between different transaction.
The transaction log are written in this way:

- BEGIN B(T)
- COMMIT C(T)
- ABORT A(T)
- INSERT I(T, O, AS)
- DELETE D(T, O, BS)
- UPDATE U(T, O, BS, AS)

where O represent the written object, AS is the After State (state of object O after modification) and BS is the Before State.

Checkpoint are periodically operation requested by the RM to the BM, it allows a faster recovery process. During the checkpoint, the DBMS writes data on disk for all completed transactions.
The flow of the checkpoint is:

1. All the TIDs of all active transaction are recorded
    - After the checkpoint start, no transaction can commit until the checkpoint ends.

2. The pages of concluded (C or A) transaction are synchronously written on disk.
    - By means of the froce primitive.
3. A the end of step 2, a chekcpoint record is synchronously written on the log.
    - Contains the set of active transactions.
    - It is written by means of the focre primitive.

Dump is a complete copy of the database, typically performed when the system is offline, stored in stabel memory, may be incremental. At the end a dump record is written in the log.

The log is designed to allow recovery in presence of failure, WAL or Commit precedence. The WAL (Write Ahead Log, SYNC) the BS of data in a log record is written in stable memory vefore database data is written on disk, during recovery it will allows UNDO operation. The COMMIT PRECEDENCE (ASYNC) solution write the AS in a stable memory before commit, this will allows the execution of redo operations.

Recovery Management there are two types of failures, the SYSTEM caused by software problem or power supply interruption that are causing losing in the main memoery content (buffer) but not on the disk. Or the MEDIA failure, caused by failure of devices managing secondary memory, this will lose the DB content on disk but not the log.
When a failure occurs the system is stopped, the type of recovery to be started depends on the failure type: SYS=WARM and MEDIA=COLD. When the recovery ends the system becomes again available to transactions.
The **WARM RESTART** is one of the solution for the recovery procedure:

- All the transaction completed before the checkpoint not need a recovery action.
- The transaction which committed, but for which some write on disk are not already performed REDO is needed.
- Active transaction at the time of failure (not committed) UNDO is needed.

the checkpoint is not needed to enable recovery, it only provides faster restart, because, without it, the entire log until the last dump needs to be read. This solution reads backwards the log to detecs action which should be undo or redo, then star to read forward the log and perform all the actions.
The **COLD RESTART** is the second solution for recovery, it is performed when a portion of the database on disk get a failure. The main steps are:

1. Access the last dump to restore the damaged portion of the DB on disk.
2. Starting from the last dump record, read the log forward and redo all actions on the database and transaction commit/rollback.
3. Perform a warm restart.