



**POLITECNICO  
DI TORINO**

# Recap Database Management System (01NVVOV)

Jacopo Nasi  
Computer Engineer  
Politecnico di Torino

I Period - 2017/2018

1 dicembre 2017

# Indice

<b>1</b>	<b>Database Management System</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	Buffer Manager . . . . .	5
1.3	Physical Access . . . . .	7
1.4	Query Optimization . . . . .	10
1.5	Physical Design . . . . .	16
1.6	Concurrency Control . . . . .	18
1.7	Reliability Management . . . . .	26
1.8	Triggers . . . . .	28
1.9	Distributed Architectures . . . . .	30
<b>2</b>	<b>Data Warehouses</b>	<b>33</b>
2.1	Introduction . . . . .	33
2.2	Design . . . . .	37
2.3	Data Analysis . . . . .	47

## License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

You are free:

- **to Share:** to copy, distribute and transmit the work
- **to Remix:** to adapt the work

Under the following conditions:

- **Attribution:** you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work)
- **Noncommercial:** you may not use this work for commercial purposes.
- **Share Alike:** if you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

More information on the Creative Commons website (<http://creativecommons.org>).



## 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

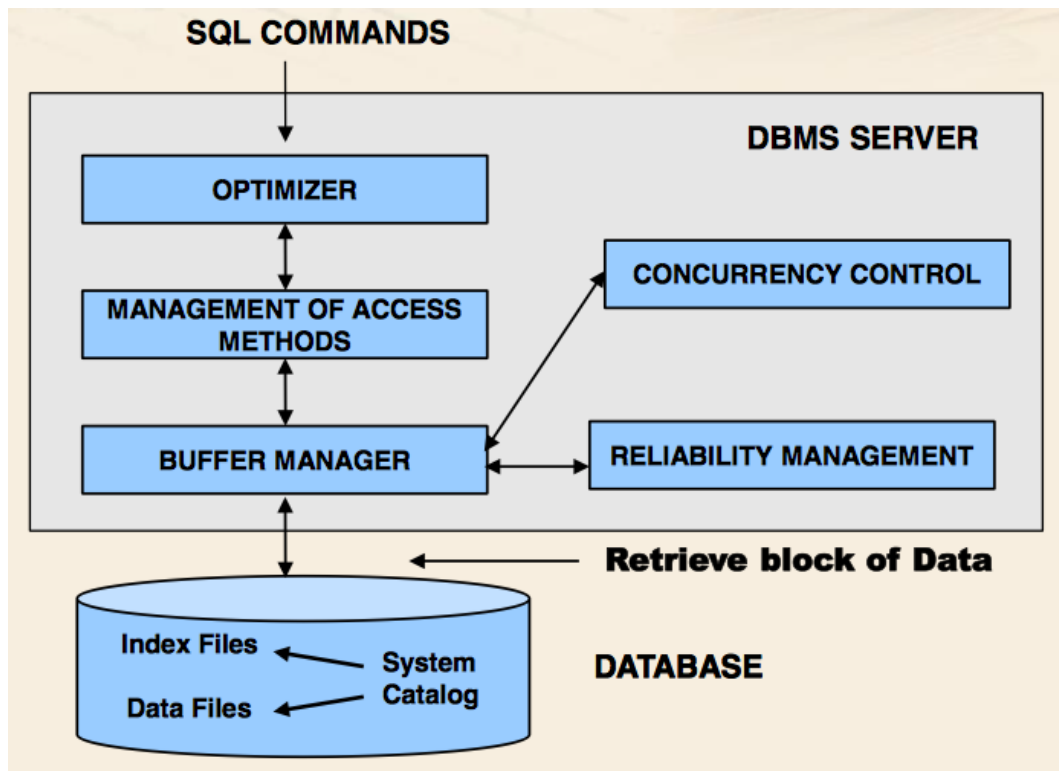


Figura 1: DBMS Architecture

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 representation.
- 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:

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity means that they cannot be divided in smaller units, is not possible 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 committed 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 transfer 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 the management strategies:

1. Data Locality: Data referenced recently 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 stores, for every page, the physical location of the page on the secondary memory (file identifier and block number) and two state variables, one that counts the number of transactions 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 if the page is not already in the buffer. There are two behaviours:

- Page already in buffer: Return the pointer to the data.
- Page not in buffer: It searches 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 decreases the Count.

**Set Dirty Primitive** it 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 transfer 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.

There 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 efficiency. 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, starting 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 is 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 by 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:



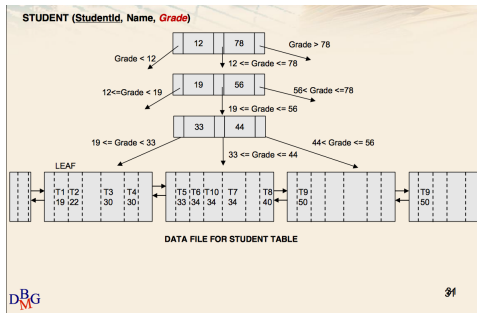


Figura 2: Clustered

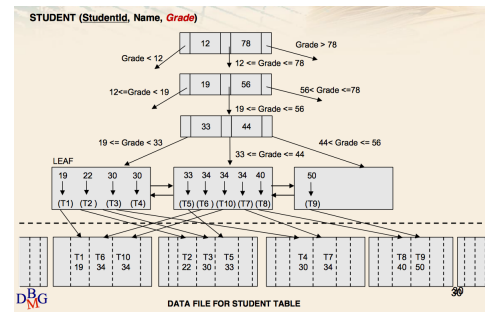


Figura 3: Unclustered

- **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]

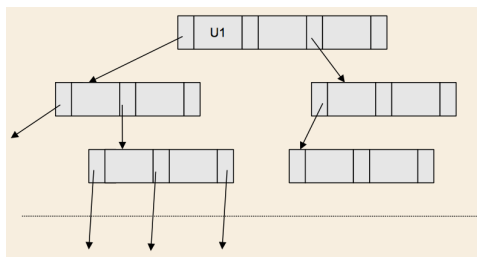


Figura 4: B-Tree

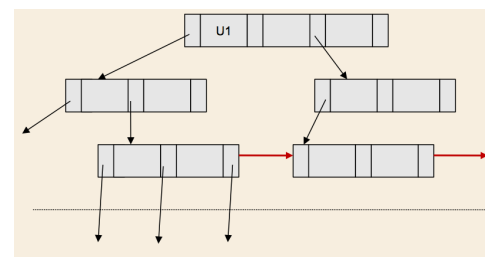


Figura 5:  $B^+$ -Tree

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

- **Advantages:**
  - Very efficient 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 well-know structure is guarantees direct and efficient 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 completely fill the blocks to allow new data insertion.

- **Advantages:**

- Very efficient for queries with equality predicate on the key.
- No sorting of disk blocks is required.

- **Disadvantage:**

- Inefficient for range queries.
- Collision may occur.

The unclustered version 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  $i$  has  $j$  like attributes for the key field, 0 otherwise. the main characteristics are:

- **Advantages:**

- Very efficient 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 guarantee the data independence property, in fact, the form in which the SQL query is written does not affect the way in which it is

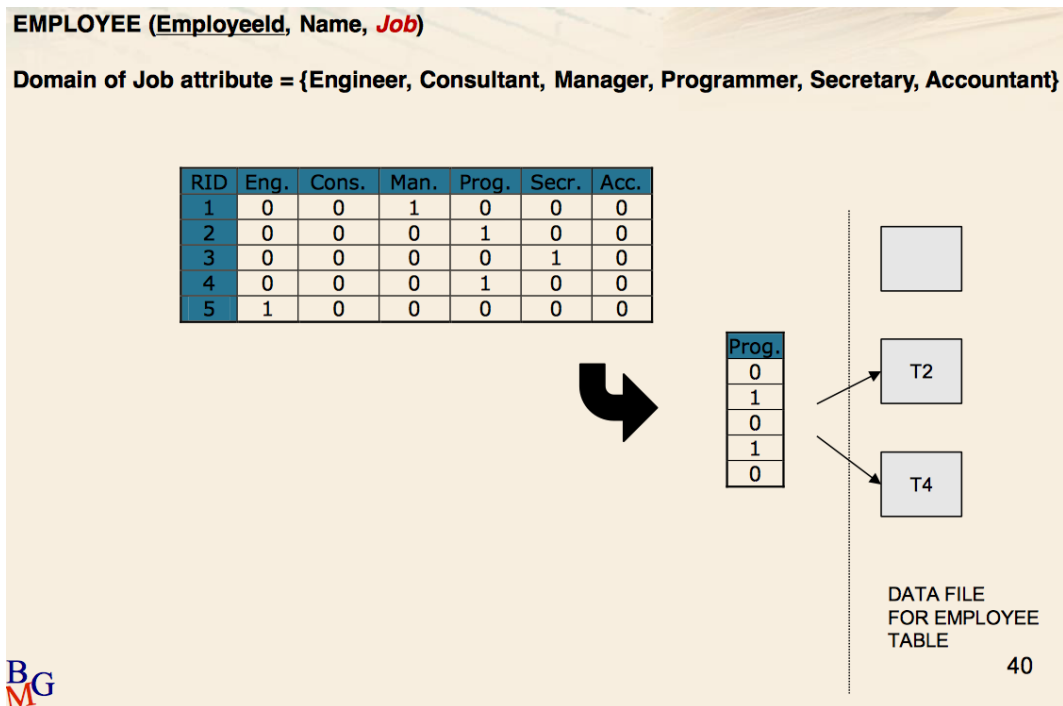


Figura 6: Bitmap Index

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, Syntactic and Semantic analysis** : check the SQL for Lexical errors (e.g., misspelled keywords), Syntactical errors in the SQL grammar and for Semantic errors not existing object called in the query (require data dictionary). The output of this block is an internal representation 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 independent 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.

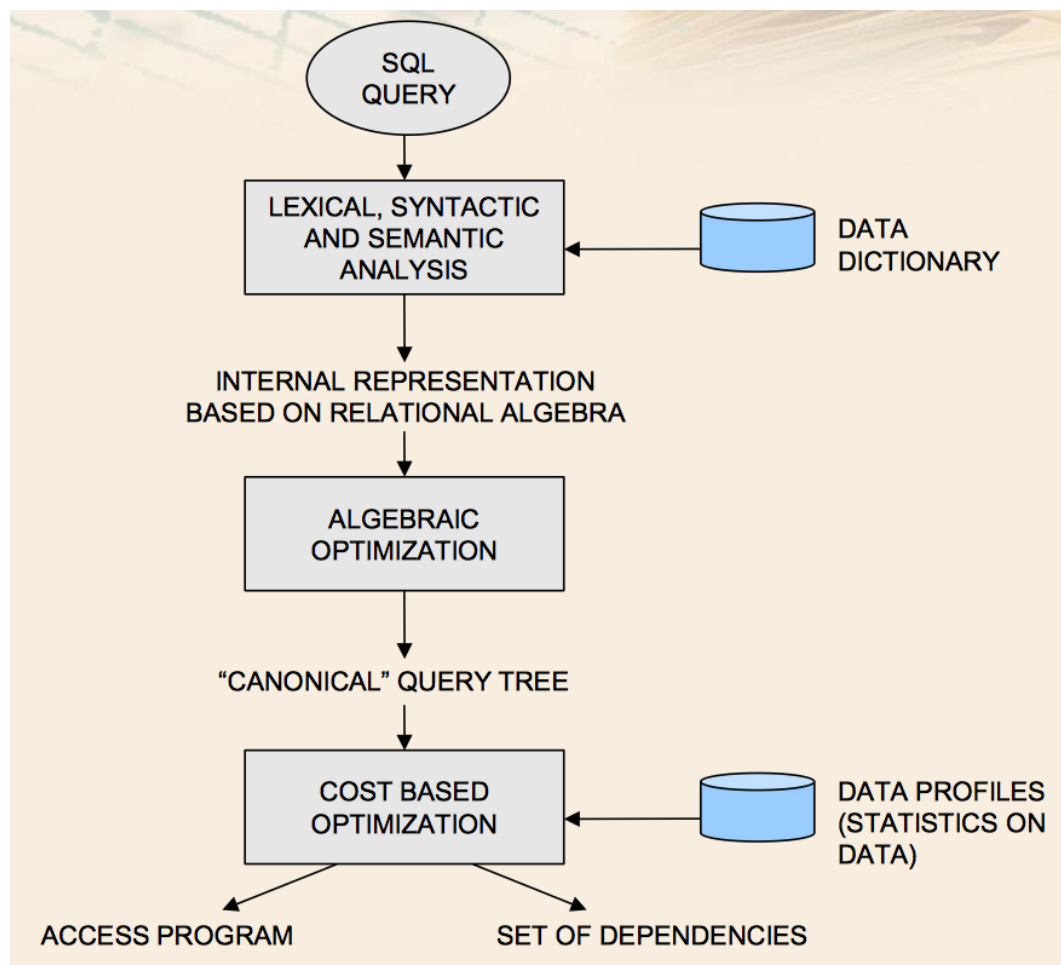


Figura 7: Optimization Execution Plan

The phase of **Algebraic Optimization** require a little more of analysis. The part is based on equivalence transformations, *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-known transformations:

1. **Atomization of selection:** Applying all attributes of selection one at the time or all together can provide different performance in case of indices.
2. **Cascading Projections:** It is possible to perform directly the final projection or doing more projections with different sub-sets 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 sub-set and then merge it with another subset or merge the two sets 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 tables is equivalent to the union of 2 already projected tables.
8. **Other:**  $\sigma_{F_1 \vee F_2}(E) = (\sigma_{F_1}(E)) \cup (\sigma_{F_2}(E))$
9. **Other:**  $\sigma_{F_1 \wedge F_2}(E) = (\sigma_{F_1}(E)) \cap (\sigma_{F_2}(E))$
10. **Distributing join respect union:** Perform join of a table E with two merged tables ( $E_1 \cup E_2$ ), is equivalent to join E with E1 and E2 separately and then 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 expressions.
- **Approximate cost:** Evaluating cost by looking at CPU, HDD and main memory usage and time.

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

- Cardinality of tuples.
- Size in bytes of tuples.
- Size in bytes of each attributes.
- Number of distinct 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 efficient 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 of 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 remaining 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 FORCE)
  - Efficient when the inner table is small and fits in memory or when the join attribute in the inner table is indexed.

- Not cost symmetric. 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. Efficient for large and already-sorted tables.
  - Requires sorting both table (already sorted or through clustered index).
- **Hash Join:** It apply the same hash function to the join attribute 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 then 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 exploiting 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 )

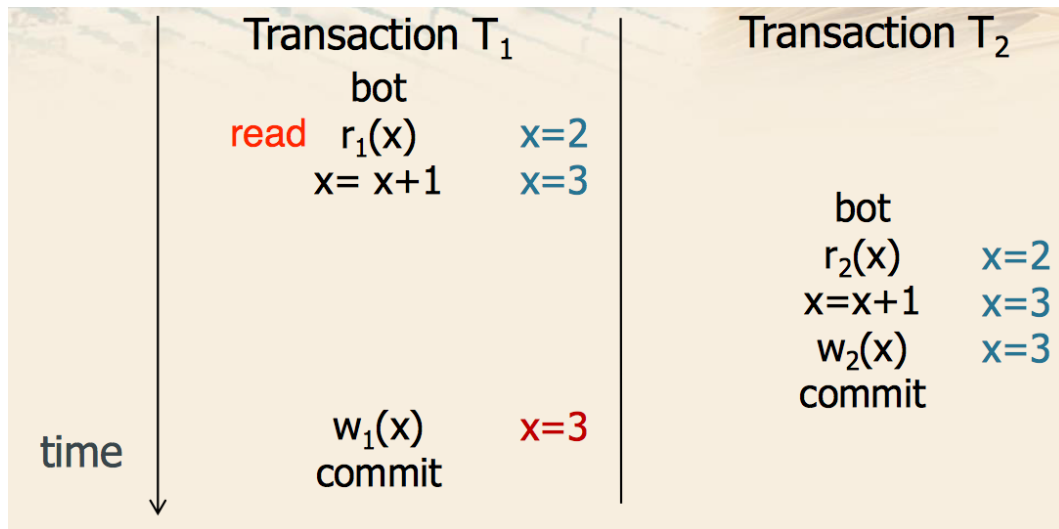


Figura 8: Lost Update Behaviour

**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.

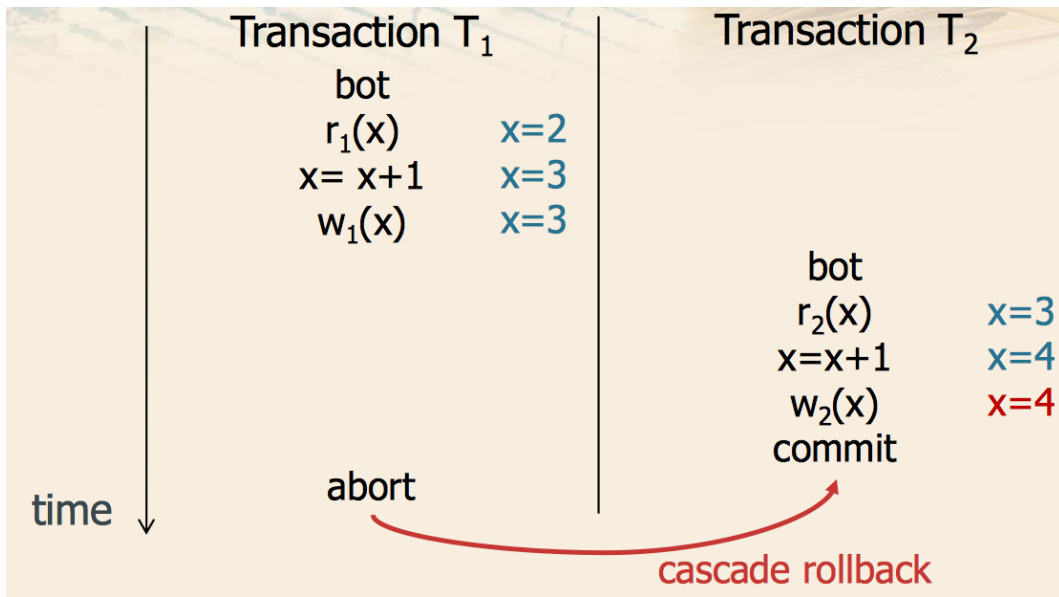


Figura 9: Dirty Read Behaviour

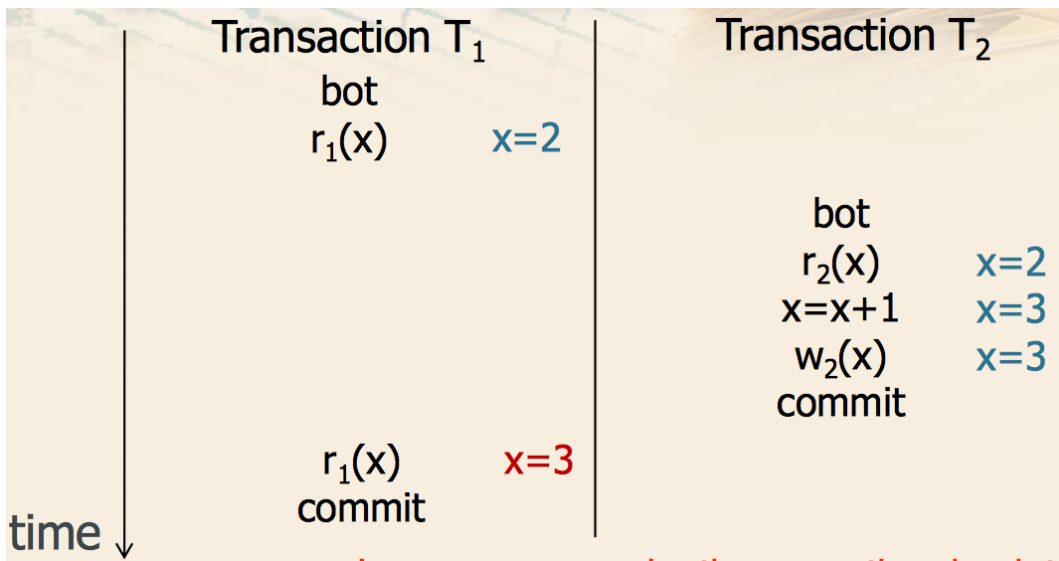


Figura 10: Inconsistent Read Behaviour

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:

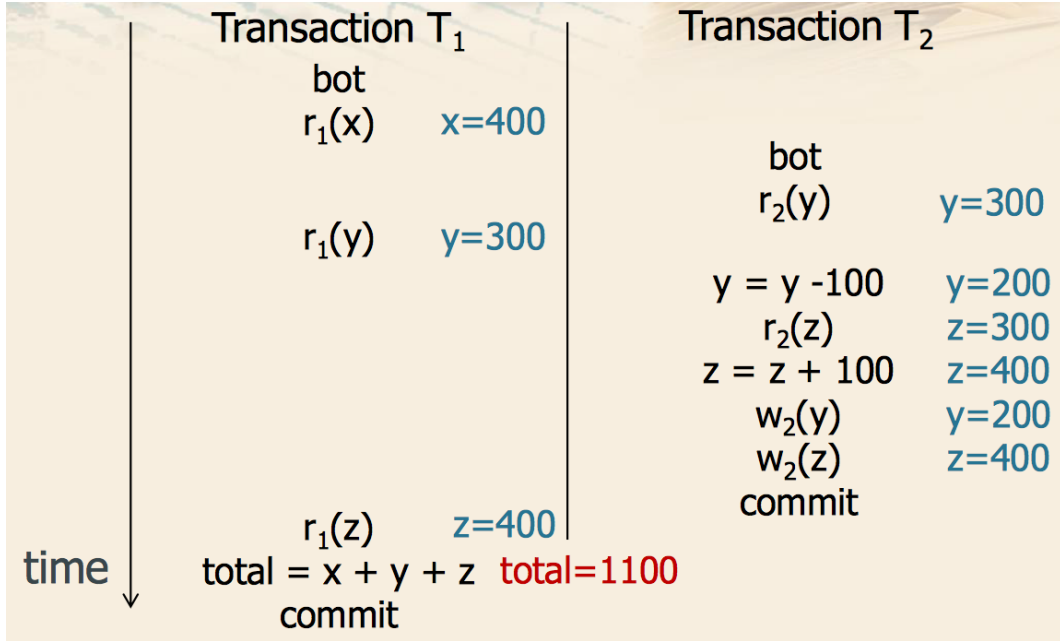


Figura 11: Ghost Update Behaviour

- 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 corresponding schedule is:  $S = r_1(x)r_2(x)w_2(x)w_1(x)$  is this

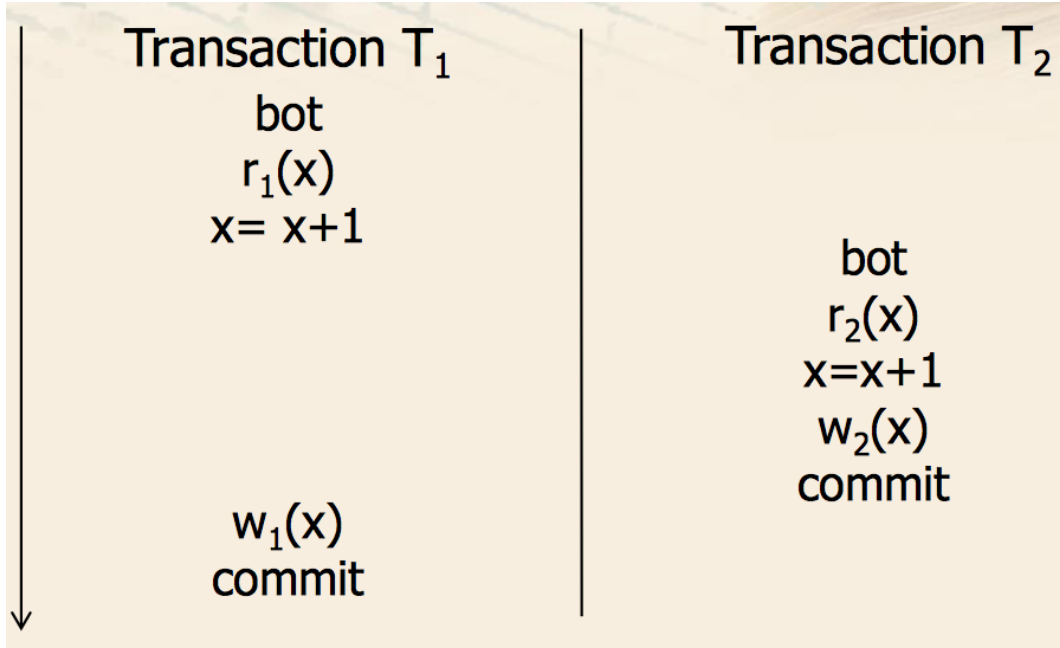


Figura 12: View Equivalence Example

schedule serializable?

There are only 2 possible serial schedules:

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

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.

Detecting the conflict serializability it is possible to exploit the conflict graph:

- Node: Each transaction.
- Edge  $i \rightarrow j$ : 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 is **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 resource 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 resource.

The lock manager, that coordinates the grant, exploits 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 of the transaction and not when the resource is released; this difference guarantees to avoid the dirty read anomaly.

The procedure is really fast. If a transaction keeps in wait over timeout the lock manager resumes it and returns a NOT OK ERROR, the requesting transaction may perform a rollback of 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

Transactions		Resources		
T <sub>1</sub>	T <sub>2</sub>	x	y	z
commit	<i>wait</i>	free	2: write	
unlock <sub>1</sub> (x)	w <sub>2</sub> (y)			
unlock <sub>1</sub> (y)	w_lock <sub>2</sub> (z)			
	<i>wait</i>			2: write
unlock <sub>1</sub> (z)	w <sub>2</sub> (z)			
	commit			
	unlock <sub>2</sub> (y)		free	
	unlock <sub>2</sub> (z)			free

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 Intention of eXclusive Lock (SIXL)**: Shared lock of the current object and intention of exclusive lock for one or more objects in a descendant 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

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 maintain.

## 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 force primitive.
3. At the end of step 2, a checkpoint record is synchronously written on the log.
  - Contains the set of active transactions.
  - It is written by means of the force primitive.

**Dump** is a complete copy of the database, typically performed when the system is offline, stored in stable 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 before database data is written on disk, during recovery it will allow UNDO operation. The COMMIT PRECEDENCE (ASYNC) solution write the AS in a stable memory before commit, this will allow 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 memory 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 solutions for the recovery procedure:

- All the transactions completed before the checkpoint do not need a recovery action.
- The transactions which committed, but for which some writes on disk are not already performed REDO is needed.
- Active transactions 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 detect actions which should be undo or redo, then starts 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 gets 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.

## 1.8 Triggers

The traditional DBMS is passive, query and updates are explicitly requested by users, the knowledge of processes operating on data is typically embedded into applications. The active DBMS instead have a Reactivity service provided that monitors specific database vents and triggers actions in response. Reactivity is provided by automatically executing rules, they can be:

- Event: Modification operation.
- Condition: Predicate on the DB state, `cond==true: action==execute`.
- Action: Sequence of SQL instructions or application procedure.

The rule engine is the component in charge of tracking events and executing rules when appropriate. The execution of the rules is interleaved with traditional transactions.

SQL provides instructions for defining triggers (CREATE TRIGGER) the syntax and semantics are covered in the SQL3 standard. The structure is divided in 3 main part:

- WHEN: the events takes place.
- IF: the condition is true.
- THEN: the action is executed.

there are also some execution modes:

- Immediate: before or after the triggering statement.
- Deferred: executed immediately before commit. (Not commercial)

and the granularity:

- Tuple (or row level): One separate exec of the trigger *for each tuple* affected by the triggering statement.
- Statement: One single trigger execution *for all tuples* affected by the triggering statement.

**Oracle Triggers** The base structure of a trigger is:

```
CREATE TRIGGER TriggerName
Mode Event {OR Event }
ON TargetTable
[[ REFERENCING ReferenceName ]
FOR EACH ROW
[WHEN Predicate ]
PL/SQL Block
```

It can be divided in:

- Mode is BEFORE or AFTER
- Event ON TargetTable is:
  - INSERT
  - DELETE
  - UPDATE
- FOR EACH ROW specifies row level execution semantics.
  - OLD.ColName and NEW.ColName are used for accessing to the two types of data.
- WHEN is used only for row level execution.

The execution algorithm is:

1. Before statement triggers are executed.
2. For each tuple in *TargetTable* affected by the triggering statement.
  - (a) Before row triggers are executed.
  - (b) The triggering statement is executed + Integrity constraints are checked on Tuples.
  - (c) After row triggers are executed.
3. Integrity constraints on tables are checked.
4. After statement triggers are executed.

the execution order for triggers with the same event, mode and granularity is not specified and it could be source of non determinism. If an error occurs the roll back of the triggers operation is performed. The triggers could also called in cascade, the maximum is defined by the user. The *mutating table* is the table modified by the statement triggering the trigger. The MT cannot be accessed in row level triggers, may only be accessed in statement triggers (limited access only on Oracle application).

## 1.9 Distributed Architectures

A possible architectural implementations is using a distributed system, the main advantages are related to performance improvement, increased availability and stronger reliability. Of course the classic client/server mechanism is much easier to be implemented and maintained. The distributed one are able to collaborate and are autonomous, the only problem is to guaranteeing the ACID property that requires more complex techniques.

**Client/Server** there are two main types of structure, the first is the **2-TIER**:

- $n$  clients: With some application logic.
- DBMS Server: Provides access to data.

the **3-TIER** solution instead:

- $n$  clients: Browser.
- Application server: Business logic and also web server.
- DBMS Server: Provides access to data.

**Distributed** system are accessed by many user at the time. Each user can be also access more than one DBMS server. Each server need to have a local autonomy, each manages its local data, concurrency control, recovery, ecc... The localization instead could be the most important difference respect c/s system, this can perform a geographical distribution. Also the data availability, less probability off total block, but more in terms of local block. Least but not last, the scalability.

**Design** given a relation  $R$ , a data fragment is a subset of  $R$  in terms of

- Tuples:Horizontal = Not overlapped, union of table possible.
- Schema:Vertical = Overlapped on PK, join of table.
- Both:Mixed

The distributed system are based on data fragmentation over different servers. The allocation schema describes how fragments are stored on different server nodes, it could be redundant or not redundant if some fragments are replicated or not on different servers. When there are replication the data availability increase, but also the complexity, synchronization is needed.

The trasparency levels explains how data distribution are visible by the query programmer, the could be invisible, in this case programmer will call only one

table without knowing the fragment division. Another option is knowing the existence of fragments but not their allocation, in this case each fragment not to be used like a different table.

**Classification** the client is only responsible to request the execution of the query, the task of redistributing the computation is demanded to the DBMS server. The transaction could be classified:

- **Remote Request:**
  - Read only request
  - Single remote server
- **Remote Transaction:**
  - Any SQL command
  - Single remote server
- **Distributed Transaction:**
  - Any SQL command
  - Each SQL statement is addressed to one single server
  - Global atomicity is needed
- **Distributed Request:**
  - Each SQL command may refer to data on different servers
  - Distributed optimization is needed
  - Fragmentation transparency is in this class only

**Technology** using more system requires synchronization. To guarantee the ACID property some techniques need to be implemented:

- Atomicity: 2 phase commit.
- Consistency: Enforced only locally.
- Isolation: Strict 2PL and 2 Phase Commit.
- Durability: Extension of local procedures to manage atomicity in case of failure.

The Distributed Query Optimization have the tasks to split in different sub-queries a single query execution request. After fast execution plan definition, the DBMS, start the different operations and coordinates everything for a correct information exchange.

The **2-Phase Commit** protocol has the objective to coordinate the conclusion of a distributed transaction. The behaviour is similar to a wedding. There is one coordinator, the Transaction Manager (like priest) and several DBMS servers which take part to the transaction, Resource Manager (like the couple). There also 2 new logs, the TM add some information related to the protocol start/end, the RM add the ready log to synchronize the commit with the other system. The procedure is similar to the window-networks protocol, with packets and ack. Figure 16 schematize the flow.

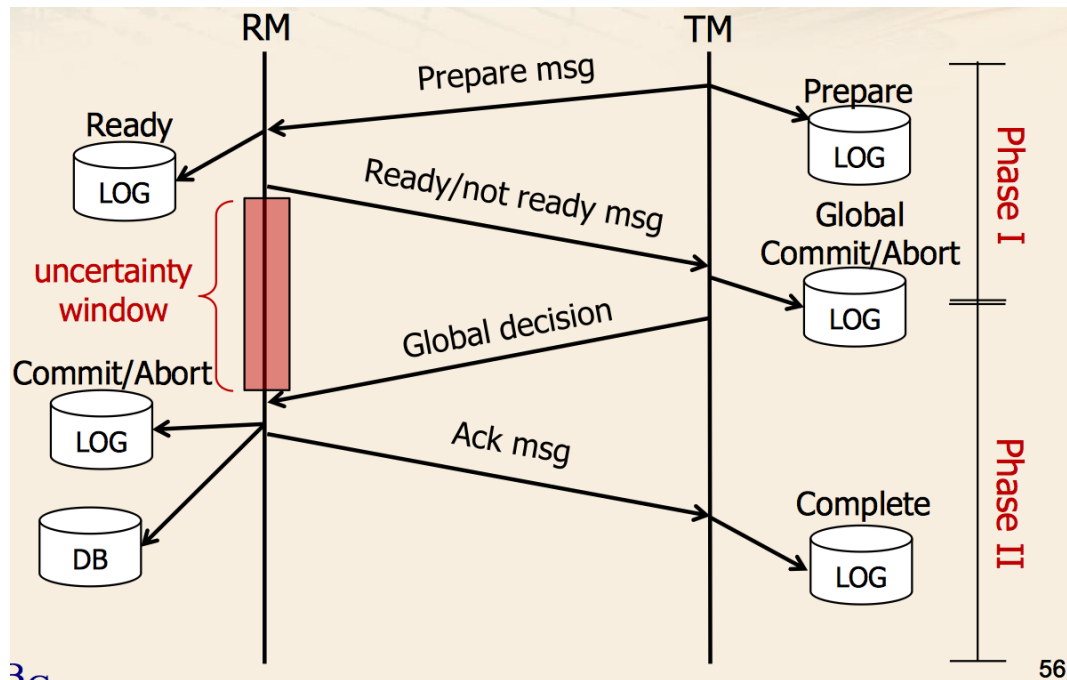


Figura 16: 2-Phase Commit

The 2 phases of the coordinator are:

- **I Phase:**
  - Prepare (Outgoing)
  - Ready (Incoming)
- **II Phase:**
  - Global Decision (Outgoing)

Also the recovery phase is a little bit modified, the warm restart will read the READY log and asked to the TM how to proceed to the restore (Remote Recovery Request). In case of the failure of the coordinator:

- Last Record=PREPARE: A global abort is written in log and send to all participants.
- Last Record=GLOBAL DECISION: Repeat the phase II.



## 2 Data Warehouses

### 2.1 Introduction

The database was developed for giving a service to the final users, like university, flight companies and other stuff. During the 80's they have understood that this system could be also used for made analysis over data. This analysis could be useful for improving decision, forecast, cost reduction and other stuff. The goal of Business Intelligence is to provide support to strategic decision, transforming company data into actionable information. This request requires of course an appropriate hardware and software infrastructure. The data warehouse is a database devoted to decision support, which is kept separated from company operational databases. The data which is:

- Devoted to specific subject
- Integrated and Inconsistent
- **Time Dependent**, non volatile

all data related to the timestamp are bigger than the other.  
The data are kept separated for multiple questions:

- **Performance:** Complex queries reduce performance. Data may vary during operation, etc... The system could be developed for doing some operations but the warehousing use the system in a different way.
- **Data Management:** The data for the service could be different from the data needed for the analytics. (ex. Data addresses changing). There could be also inconsistency problem.

One of the representation solution proposed is the (hyper)cube with three or more dimensions (figure 17). In this solution the cross of the three axes could be our important data, 3 like number of Milk sales in 2-3-2000 by the SupShop or the total amount in \$ etc... The empty cells represent an inconsistency data, product not sales in that day.

The hypercube is a representation based on the relational representation, the STAR MODEL. This model have:

- Numerical measures: Value stored in the fact table. (ex. #sales)
- Dimensions: Describe the context of each measure in the fact table.

in figure 18 the Dimensions are *Shop*, *Date* and *Product* and the fact is the *Sale*.

An important analysis is related to the dimensions of the data warehouse, supposing: to store 2 years of data, for 300 shops for 3000 product sold every day in every shop, the fact table reach the dimensions of 660 millions of row

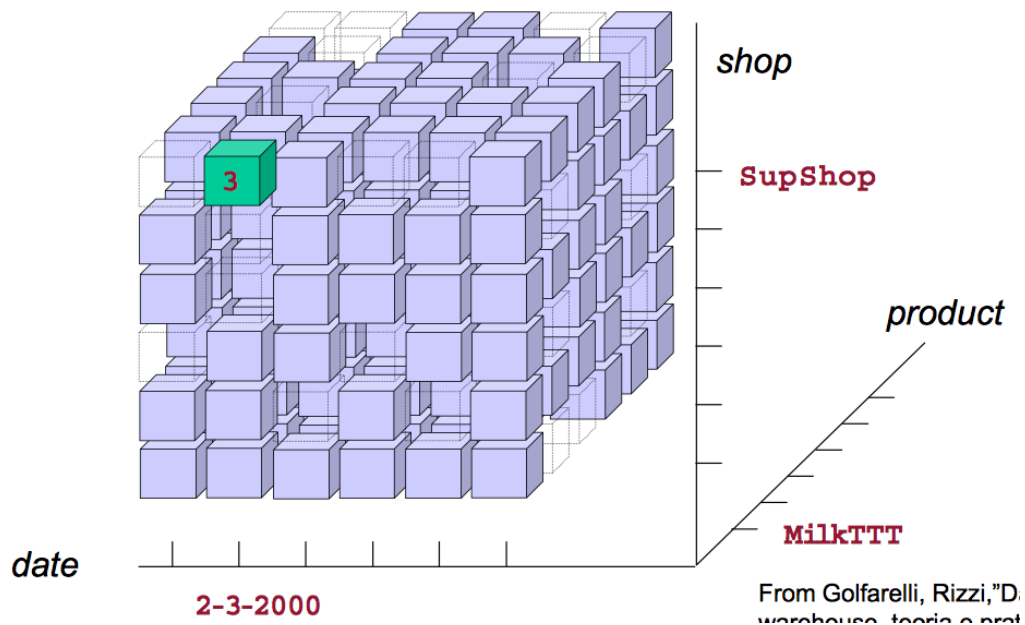


Figura 17: Hypercube representation

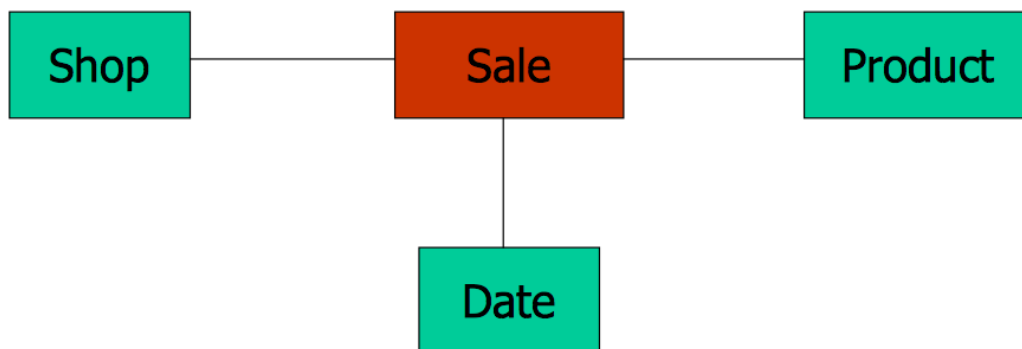


Figura 18: Star Model

(around 20GB of data).

The main operations over this kind of data is the computation with aggregated functions, with more complex operand (moving average, top ten, ecc...) or applications of data mining techniques. This important evaluation are unusefull if the data is presentate bad.

**Architecture** the system is build on several block:

- Data Sources: External source, the DB.
- Data Warehouse: The big system.

- **Data Marts:** Smaller data warehouse related only to a single department.
- **OLAP servers:** Multi dimensional data representation.
- **Metadata:** Schemas information.

The data warehouse contains all the information on the company business and it requires a long time to be developed, the data mart are departmental information subset focused on a given subject, the implementation is faster than the warehouses, they requires a careful design. The data mart could be dependent (fed by the company warehouse) or independent (fed directly by the sources). The most know solutions are:

- **ROLAP:** (Relational OLAP)
  - Extended relational DBMS (not sparse).
  - SQL extension for aggregate computation.
  - Specialized access methods which implement efficient OLAP data access.
- **MOLAP:** (Multidimensional OLAP)
  - Data represented in multidimensional matrix (sparse data required compression).
- **HOLAP:** (Hybrid OLAP) use MOLAP for fast and user stats and ROLAP for high detailed data.

There are 3 different solution for the system architecture, 1,2 and 3 level. The first it better to be avoided, is not a good choice to have DBMS and DW on the same side. The other two are, the 2 level represented in figure 19. These solution split the data and the DW, the only problem is data as soon as the data is added an "On the fly" data transformation is requires. Instead, in the three level solution (figure 20), the problem is avoided introducing a staging area used for managing and cleaning operation.

**ETL** Extraction, Transformation, Loading this process prepares data to be loaded into the data warehouse, is usually performed during the first load of the DW or during periodical DW refresh. The part are:

- **Data Extraction:** Data acquisition from sources.
- **Data Cleaning:** Techniques for improving data quality (correctness and consistency).
- **Data Transformation:** Data conversion from operational format to DW format.
- **Data Loading:** Update propagation to the data warehouse.

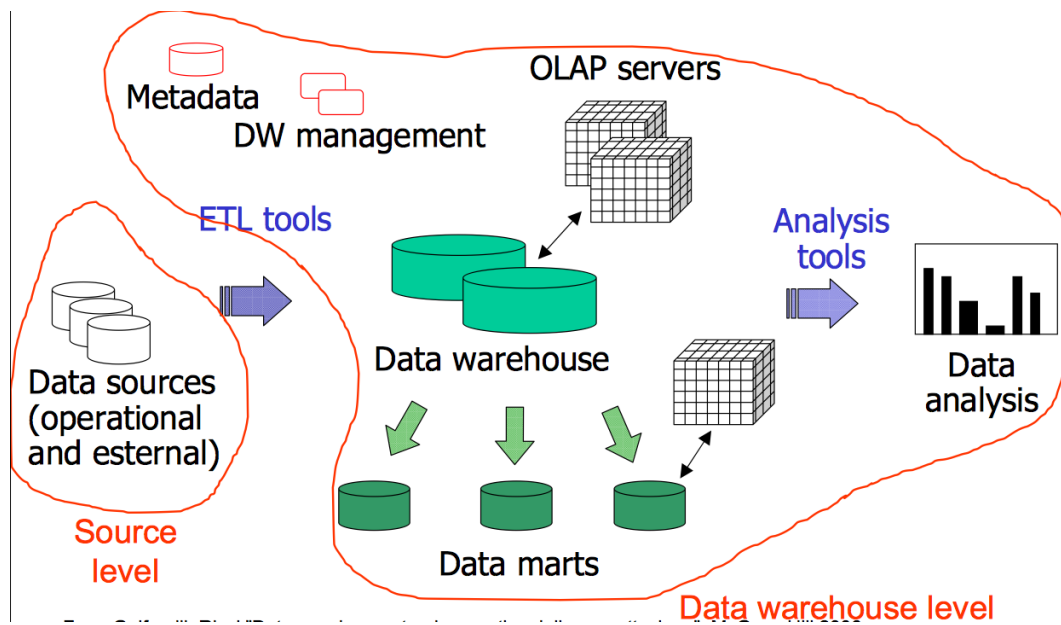


Figura 19: Two Level Architecture

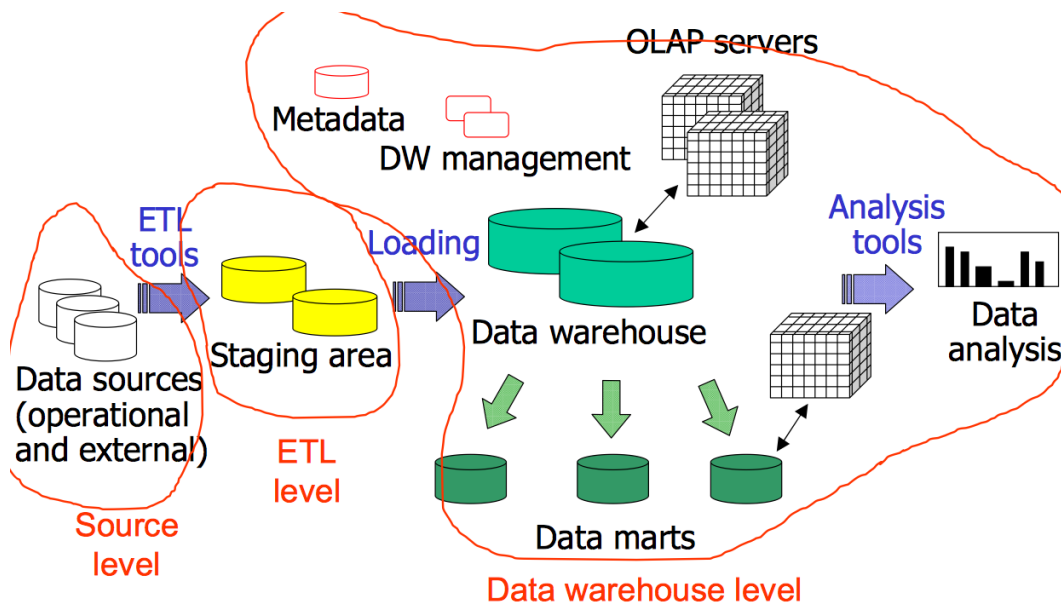


Figura 20: Three Level Architecture

**Metadata** these are data about data. There are different types:

- Data transformation and loading: Describe data sources and needed transformation operations.
- Data management: Describe the structure of the data in the DW, also for materialized view.

- Query management: Data on query structure and to monitor query execution (execution plan, memory and CPU usage).

## 2.2 Design

Using this type of structures involve some risks. Often the user think that data warehousing can solve the company's problems but is not like this, the behavior of this system is to provide that to solve in a better way the problem, not to solve problem. Also the source data could generate problems, incomplete or unreliable, also non integrated or non optimized business processes. Good idea could be "politically correct" and developing a easy-to-use system.

**Approach** there are two types of development:

- **Top-Down:** Global and complete representation, complex and expensive.
- **Bottom-Up** incremental growth, separately focused on specific business areas.

in general the system follow a common flow, the Kimball lifecycle is a good representation.

The data mart design:

- Operational source schemas
- Reconciled schema: Not easy do design.
- User Requirements
- Fact schema: non standard representation, usefull for developing.
- Feeding: Define script for saving data.
- Physical Design: Good system for improving performance.

**Requirement Analysis** the first phase of designing is the analysis of the whole problem. We need to know:

- Collects: The data needed in the data mart and the constraints related to previous information system.
- Sources: Business user and operational system administrator.
- Select: Is good to strat from the most important sector of the company and feeded by few reliable sources.

The application requirements, what we need to keep in the data mart, are different:

- Description of relevant facts: Could be sales, phone call, investments, ecc... With its usefull information (dimension). Granularity and time span.
- Workload: From already existent report generate new report. Other stuff produced in natural language.

There are also some structural requirements:

- Feeding periodicity: Different from realtime or weekly reports.
- Available space
- System architecture: 1, 2 or 3 level.
- Deployment: Start up and training.

**Conceptual Design** there isn't a standard model for this of kind of design, the dimensional fact model is proposed from Golfarelli e Rizzi in their book. This model define, for a given fact, dimensions, hierarchies and measures. The part are, figure 21:

- **Fact:** It evolvs in time modelling a relevant events (ex. sales).
- **Dimension:** Describes the coordinate of the fact (ex. sales date, shop, ecc...).
- **Measure:** Describes a numerical property of the fact (ex. number of sold unit).

The hierarchy is the collection of all associated attributes of each dimensions. The attributes describe the dimension at different abstraction levels, this hierarchy represent a generalization relationship among subset of attributes in a dimension. Each edge represent a functional dependency 1:n. An example of hierarchy is:  $Shop \rightarrow ShopCity \rightarrow Region \rightarrow Country$ .

There some advanced features in these model:

- Non-additivity: Not aggregatable over sum.
- Optional edge
- Convergence: Two different hierarchy converge at the sime point at the end of its.
- Optional dimension: Not always existent dimension.

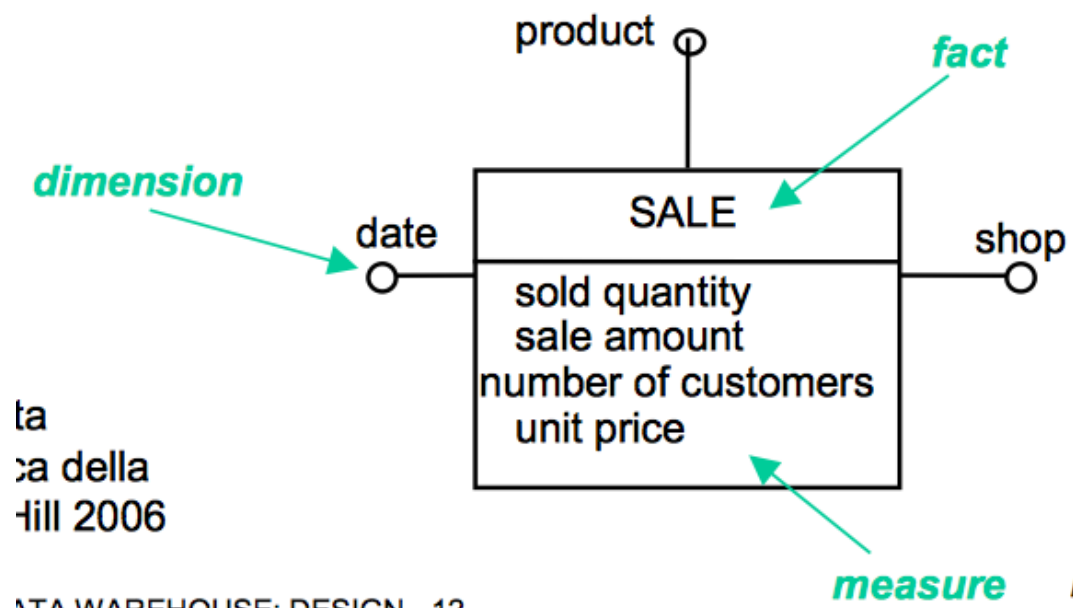


Figura 21: Conceptual Design

- Descriptive attribute: Not aggregatable for computation, only descriptive.

there other stuff likes:

- Multiple Edge: Relation n:n instead of 1:n.
- Shared Hierarchy: Different role from the same subset of data.

An interesting situation could be when the representation of a fact without measure, is called factless fact schema. It can occur when you need to record the occurrence of an event, count the number of occurrence or representing events not occurred.

**Aggregation** computes measures with a coarser (less fine) granularity than those in the original fact schema. The reduction is obtained climbing the hierarchy. The standard operator are SUM, MIN, MAX, AVG, COUNT.

The measure can be additive, not additive (by SUM), not aggregatable. Measure can be of 3 type:

- **Stream:** Evaluable at the end of a time period, aggregatable by all operators (ex. sold quantity).
- **Level:** Evaluated at given time (snapshot), not additive along the time dimension (ex. Balance).
- **Unit:** Evaluated at a given time and expressed in relative terms, not additive (ex. Unit price product).

When is possible to compute aggregate from view of already aggregated value these operators are called **distributive**. Not all operators are distributive, AVG is **algebraic**, these types can compute higher level aggregations from more detailed data only when supplementary support measures are available. There is also another type called **Olistic** that can't compute aggregate from more detailed data.

**Time Representation** the data modification over time is explicitly represented by the event (ex. time of buying, timestamp measurement). Also dimension could be change, the number of professors, the number of stores, ecc... The only difference is related to the speed of change, this is named *slowly changing dimension*.

The first (**TYPE I**) solution of this problem is to overwrite the data with the current value, this is used when the change is not important for the system like solving an error in a surname. This project the new situation over the past events. The second (**TYPE II**) solution is to directly correlate the events with the corresponding dimension value. This could be achieved partitioning the data and creating an instance of the dimension. For example, *Purchases performed by married Mario Rossi* and *Purchases performed by unmarried Mario Rossi*. The last solution (**TYPE III**) is similar to the type II because it creates a new instance of the dimension, but it introduces a validity time stamp (start/end) for the dimension and a new attribute which allows to identify the root of the all dimension. The main utility of the last type is the projection to the future or to the past of an identity.

For example:

*If we have a geo sub-division like nord-ovest and we compute the sels 2016 of this area, if we move one store to the area nord-center the computation of the sels of 2016 get a different results. Supposing to calculate the sels of the 2017, if we want to look at the results of the old nord-ovest configuration, but with the new data, the only solution is to use the TYPE III representation.*

The third solution must be used only if the system really needs this because it could be really complicated to be maintained.

**Other** the workload must be defined during the design phase, this can depend on user number, complexity and dimension. Probably a phase of tuning will be necessary for improving the system work.

The estimation of data volume is necessary due to a correct development of the system. Everything must be considered, from indices, to materialized view, to time span, attribute length, ecc.. This is because this type of structure everything could become really big. Difficulties on volume estimation come from sparsity, because when we reduce granularity, with a high sparse cube, the reduction factor could be greater than expected.



**Logical Design** this phase start from the relational model (ROLAP):

- Conceptual fact schema
- Workload
- Data Volume
- System Constraints

Is a little bit different from the traditional logical design because the data redundancy (lose of normalization) is acceptable due to a performance improve. Is a good idea to add redundancy only in the dimension table and not in the fact table, because the number of records is really really bigger in the second one. The star schema (figure 22) is composed by:

- Dimensions:
  - One table for each dimension
  - Surrogate primary key (counter)
  - It contains all dimension attributes
  - Not explicit hierarchies respect ROLAP schema
  - Totally denormalized representation
- Facts
  - One fact table for each fact schema
  - Primary key composed by foreign keys of all dimensions
  - Measures are attributes of the fact table

Directly from the star schema we can derive the **Snowflake** schema that introduce a bit of normalization to reduce the size of the dimension table. This representation is created splitting in two or more table one dimension table, an example in figure derived directly from the previous star schema.

The main advantage are related to some space optimization, the disadvantage is that this solution need one or more further join. Normally the star schema is preferred. The only cases could be when one part of one dimension is shared above more dimension.

**Multiple edges** needs an appropriate implementation, there are two solution:

- **Bridge Table:** similar to classic relational schema add a table between the two tables linked by the edge to "merge" the cases. This solution allow to add usefull attributes, like weight for computing specific calculations (ex. author income in a not equally division book).

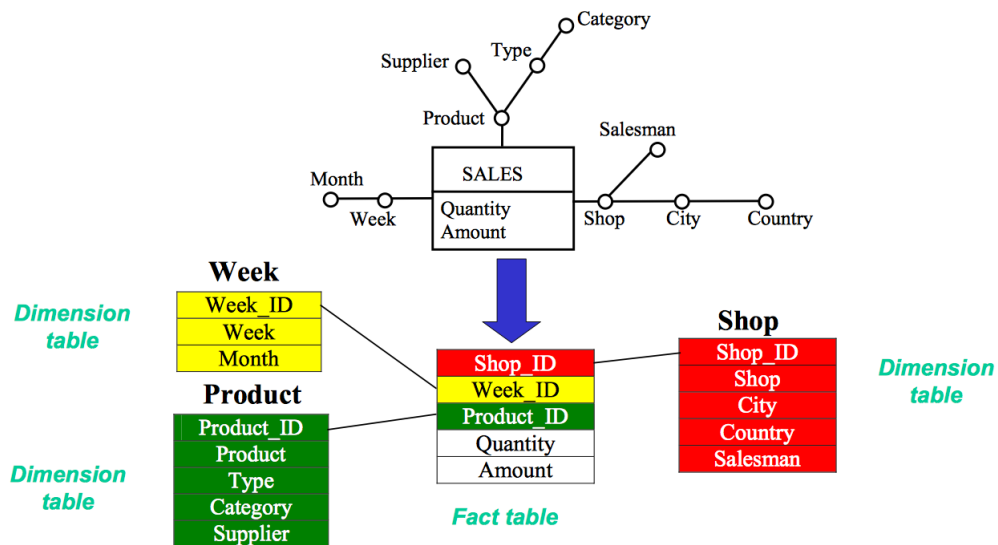


Figura 22: Star Schema

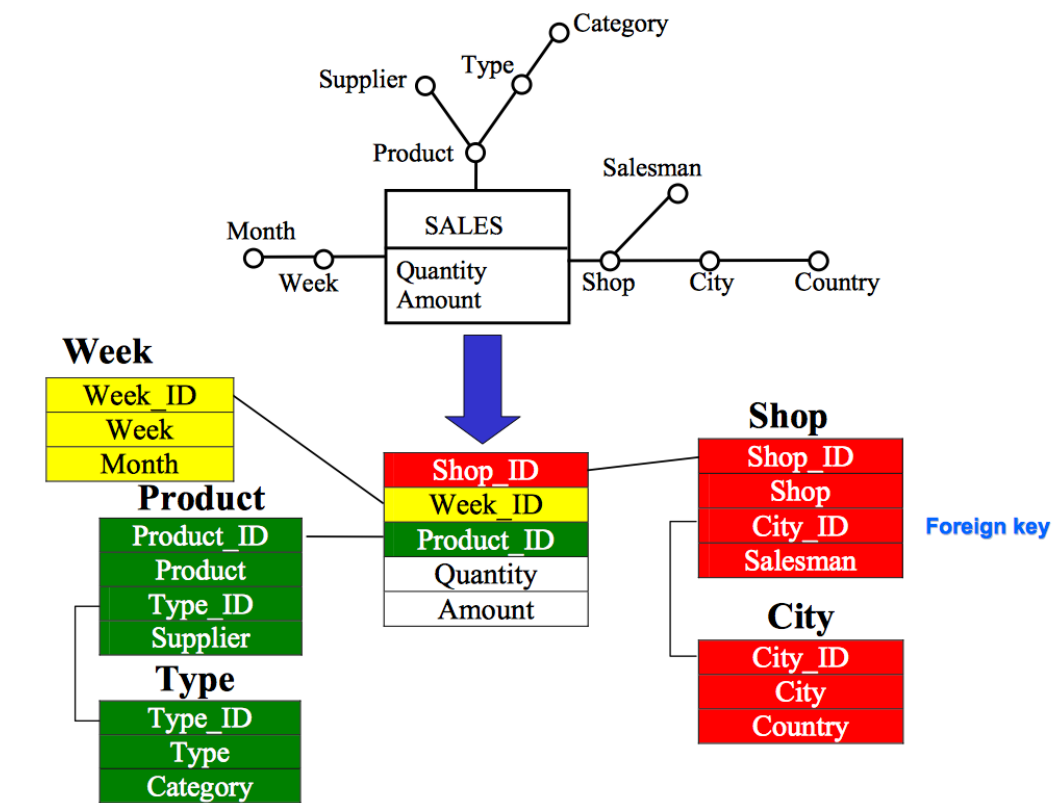


Figura 23: Snowflake Schema

- **Push Down:** Add 2 keys to the fact table. The weight in this case is wired in the fact table. The PD add problem related to increase of

redundancy, the only advantage is a minor number of joins required.  
NOT GOOD SOLUTION.

In some cases it could be necessary to create a dimension with only one attribute for a dimension, this is named **degenerate dimension**. There are two options to overcome the problem:

- Attribute directly integrate into the fact table (only attribute with very small size).
- Junk dimension: A single dimension containing several degenerate dimensions with no functional dependencies among them.

**Materialized Views** respect the view, the materialized ones, are tables from all aspects as they aren't computed at the time the view is requested. The materialized view implements an already aggregated result of a previous interrogation of the fact table. They are used to improve the performance during the computation of queries less detailed. An example in figure 24. The views

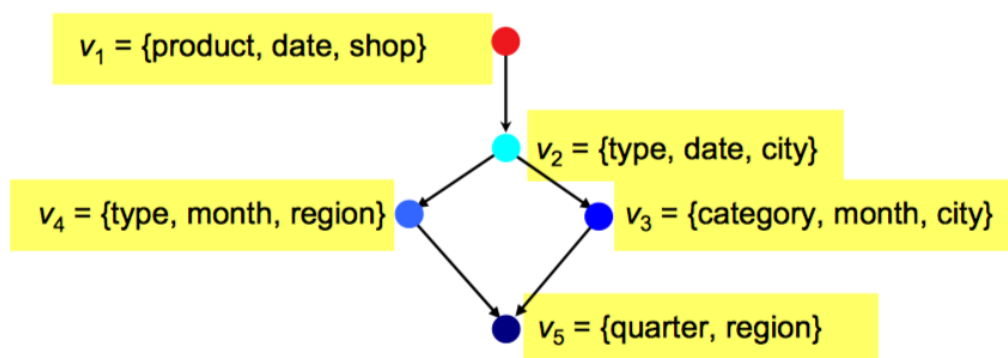


Figura 24: Example of materialized view

are SQL statements, generated starting from base tables or views with higher granularity. Considering that these views can take up a lot of space, it is better to use them only when they are really useful. A good advice is to develop views that are used from a lot of interrogation. Of course, generating a view that includes a lot of aggregation attributes can be too big to be maintained, also the space occupied could be too much. In general, the idea is to minimize these values:

- Disk space
- Update window
- Query cost

During the physical development of a data warehouse is important to manage the workload. The queries are computed with a lot of aggregation level and they will probably access to an huge part of each table involved. One of the main advantage of this system is the they are read-only and they don't require to be modified. A periodical scheduled refresh is needed to keep the data valid, also indices and other stuff.

Respect the OLTP this type of system use different index types like bitmap, join, bitmapped join index, ecc...

An important phase of the development is the tuning where, looking at the real utilization behaviour the system is fitted to improve performance during real-life application. This phase is also used in case of structural changes. In more complex environment the parallelism become important, it can really increase the performance.

The index selection is performed on:

- Attributes frequently involved in selection predicates.
- High cardinality domain, with B-tree index.
- Low cardinality domain, with bitmap index.

Indexing only foreign keys in the fact table is rarely appropriate because the index could become big like the table reducing the real advantages of this stuff.

**ETL** the Extraction, Transformation and Loading phases are fundamental to prepare data to be loaded into the warehouse. Is performed at the first DW load and during periodical refresh, it's involve:

- Data extraction from OLTP or external sources.
- Data Cleaning.
- Data transformation.
- Data loading.

The process is eased by using the staging area. In general this phase is different if the system is during the first load or if is a periodical update/

The **extraction** acquire the data from the sources. If is the first time perform a *static* extraction grabbing snapshot of operational data. If is an update, it will use an *incremental* solution to extracting data since the last update only. In case of multiple sources is important to choose the best one. Of course is important to analyze how the data is collected:

- **Historical:** All modification, with a datastamp, are stored for a given time in the OLTP system (bank transaction, exam, ecc...).
- **Partly Historical:** Not all the modification are stored, only a limited number of state.

- **Trasient:** Only the current data state is keeps on the OLTP.

Of course the real challenge is over trasient data.

The incremental extraction comparing 2 snapshot of the system it's a mess. One solution to this problem is to be **assisted by the application**, this means force the OLTP system to keep track of modification, on the selected table, in dedicated table. This means doubling the application load because every insert or update must be performed 2 times. This is the last solution, just used only on legacy system, because require a modification directly in the applications that perform the modification.

The best solution is using the **log**, this is written in any case for recovery pour-pose, this means that using the log not require unnecessary data operations. Both operations are performed at the same time that the main transaction is done, are immediate.

Another possible solution is using the **triggers**, there is the duplication like the application support, but this solution not require editing the single application, but editing only the DBMS.

The last solution is using the **timestamp**. The system need a new attribute for the *last modification time*. During the data extraction, first the system will find the first "not yet added" record, and then will start to scan the table from this point. This is the only deferred extraction and it means possible lose of intermediate state if the data is transient. A fast comparison over the various techniques in figure 25.

	<i>Static</i>	<i>Timestamps</i>	<i>Appilcation assisted</i>	<i>Trigger</i>	<i>Log</i>
<i>Management of transient or semi-periodic data</i>	No	Incomplete	Complete	Complete	Complete
<i>Support to file-based systems</i>	Yes	Yes	Yes	No	Rare
<i>Implementation technique</i>	Tools	Tools or internal developments	Internal developments	Tools	Tools
<i>Costs of enterprise specific development</i>	None	Medium	High	None	None
<i>Use with legacy systems</i>	Yes	Difficult	Difficult	Difficult	Yes
<i>Changes to applications</i>	None	Likely	Likely	None	None
<i>DBMS-dependent procedures</i>	Limited	Limited	Variable	High	Limited
<i>Impact on operational system performance</i>	None	None	Medium	Medium	None
<i>Complexity of extraction procedures</i>	Low	Low	High	Medium	Low

Figura 25: Comparison of extraction techniques

Another important phase of the ETL is the **cleaning**. That phase have the task to clean all the data problem related to entry errors, different field formats and for evolving business pratices. This means remove duplicate, fill missing data, correct wrong use of field, remove impossibile or wrong data and

all the other inconsistency correlated. That problem can be solved using **data dictionary** used for entry or format errors. This of course can be performed only over data domains with limited cardinality. Another solution could be the **approximate fusion**, these techniques tries to find duplicates or similar, with a defined criterion, and perform operation to fix it when is possibile, otherwise the data must be purge again manually.

The **trasformation** required data conversion from the operational to the warehouse format. The phase is divide in two part:

- From operational to the reconciled data in the staging area.
  - Conversion
  - Matching
  - Data selection
- From reconciled to the data warehouse.
  - Surrogate keys generation
  - Aggregation computation (view, fact table, indices, ecc...)

the dimesion table loading is represented in figure 26 and the fact table loading in figure 27.

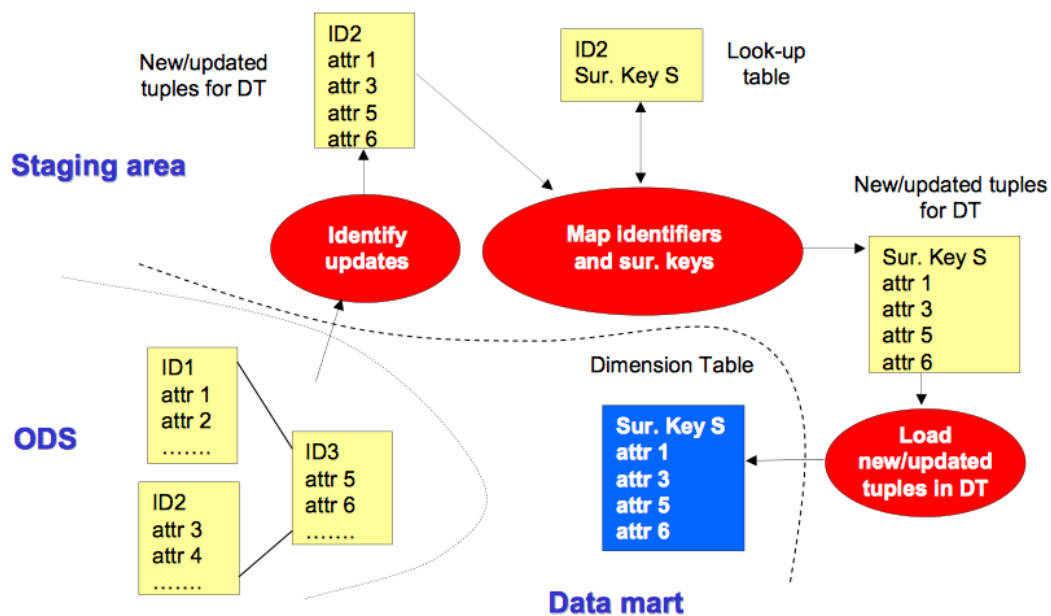


Figura 26: Dimension table loading

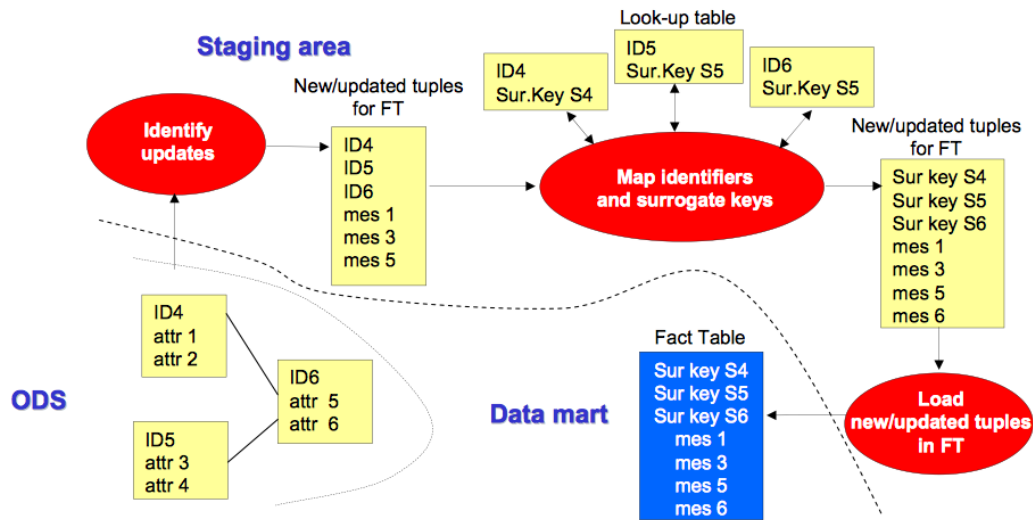


Figura 27: Fact table loading

## 2.3 Data Analysis

The analysis over data warehouse is slightly different from the normal query over relational DB. The OLAP analysis requires to compute complex aggregate function for aggregation, for comparisons or data mining. The normal SQL isn't enough comfortable to be used for this kind of computation. The datawarehouse could be query by various tools:

- Controlled Query Environments
- Query and Report generation tools
- Data mining

In the **Controlled query environment** have always a predefined structure with complex query, ad hoc analysis procedures and predefined reports. This environment requires ad hoc code development, stored procedures, predefined joins, aggregation and so on. Also some useful and flexible package for a better and easy-to-use report management are available. The system could be also fitted to use KPI of a specific economic area, like economical or financial indicators.


When the CQE is not enough for the user a solution can be to develop an **Ad hoc query environment**. With this solution the user can define OLAP queries with a point and click techniques that generates SQL code, this means that the user must know a little bit the data structure of the warehouse. The advantage is that, like the CQE, the user can exploits complex query with spreadsheet reports techniques.

**OLAP Analysis** this adds some useful operations to perform more complex query.

The rollup is a technique used for decreasing the data detail, this could be obtained by climbing up the hierarchy, from *group by store, month* to *group by city, month*, or dropping a whole dimension, from *group by product, city* to *group by product*. The drill down is the opposite of the rollup, it increase the data detail add a whole dimension or walking down the hierarchy. This operation could generate some problem related to sparse matrix and data explosion, is problem can be solved using the slice and dice technique. This last operations selects a data subset by means of selection. The *slice* means selecting a slice not changing the granularity but adding only a equality predicate, the *dice* try to reduce the information of the slice over some reduced set. The result of this operation in figure 29, over the data in figure 28. The last

Category	Year	Metrics Customer Region	Dollar Sales									
			North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germa
Electronics	1997		\$ 138	\$ 1.774	\$ 384	\$ 138	\$ 2.346	\$ 2.554	\$ 2.184	\$ 566	\$ 199	\$
	1998		\$ 1.184	\$ 4.529	\$ 1.892	\$ 7.232	\$ 651	\$ 9.488	\$ 476	\$ 2.683	\$ 462	\$ 7
Food	1997		\$ 759	\$ 682	\$ 729	\$ 262	\$ 588	\$ 469	\$ 807	\$ 156	\$ 615	\$ 1
	1998		\$ 538	\$ 925	\$ 959	\$ 677	\$ 213	\$ 1.503	\$ 261	\$ 165	\$ 175	\$ 1
Gifts	1997		\$ 2.532	\$ 1.355	\$ 1.854	\$ 1.413	\$ 2.535	\$ 2.132	\$ 1.904	\$ 908	\$ 375	\$ 1.0
	1998		\$ 1.955	\$ 2.785	\$ 2.800	\$ 2.695	\$ 1.813	\$ 2.844	\$ 1.778	\$ 1.158	\$ 717	\$ 6
Health & Beauty	1997		\$ 624	\$ 640	\$ 1.317	\$ 647	\$ 588	\$ 754	\$ 654	\$ 143	\$ 292	\$ 3
	1998		\$ 611	\$ 887	\$ 566	\$ 382	\$ 499	\$ 1.162	\$ 1.044	\$ 273	\$ 72	
Household	1997		\$ 5.354	\$ 4.112	\$ 5.410	\$ 4.446	\$ 3.058	\$ 3.974	\$ 2.654	\$ 3.545	\$ 2.875	\$ 1.9
	1998		\$ 5.787	\$ 5.320	\$ 5.416	\$ 6.812	\$ 4.334	\$ 5.008	\$ 7.588	\$ 2.139	\$ 3.649	\$ 2.7
Kid's Korner	1997		\$ 201	\$ 398	\$ 485	\$ 186	\$ 409	\$ 323	\$ 396	\$ 105	\$ 34	\$
	1998		\$ 247	\$ 422	\$ 441	\$ 380	\$ 221	\$ 592	\$ 290	\$ 198	\$ 19	\$
Travel	1997		\$ 624	\$ 505	\$ 564	\$ 386	\$ 300	\$ 978	\$ 416	\$ 48	\$ 38	
	1998		\$ 608	\$ 559	\$ 1.096	\$ 611	\$ 464	\$ 316	\$ 573	\$ 257	\$ 198	\$

Figura 28: Starting table



Filter Details: Category = Electronics AND Dollar Sales > 80 AND Customer Region = North-West AND Year = 1997							
Subcategory	Metrics Customer City	Dollar Sales					
		Alta	Armstrong	Avery Heights	Lane	Mt. Everest	San Fransisco
Audio			\$ 98		\$ 123	\$ 85	
Comfort				\$ 118		\$ 1.495	
Gadgets		\$ 199					\$ 199

Figura 29: Table sliced and diced

operation is the pivot, this solution reorganize the multidimensional structure,



swapping the axes, without varying the detail level, this could be comfortable to increase the readability of the same information.

**Extension of SQL language** this extension was introduced to support, in a better way, the new computation request of the data warehouse world. The were also standardized from the ANSI.

The computation window is a new clause introduced by the extended SQL. Is characterized by:

- **Partitioning:** Rows are grouped, like *group by*, but without collapsing them.
- **Reordering:** Rows are ordered inside its group created by the partitioning.
- **Aggregation Windows:** It defines where perform the aggregation.

Example:

*Show, for each city and month: (1) Sales amount and (2) Average on the current month and the two previous months, separately for each city.* The query will be:

```
SELECT City, Month, Amount,  
        AVG(Amount) OVER (PARTITION BY City  
                        ORDER BY Month  
                        ROWS 2 PRECEDING)  
  
        AS MovingAvg  
FROM Sales
```

Some consideration over this function: The sort order is required although the computation of the window become inconsistent. When the window is not complete, like end of the month, the computation takes place on the available rows. Is possible to use several window at the same time. The aggregation window may be defined in two ways:

- **Physical Level:** It builds the group by counting rows (*current and 2 preceding rows*).
- **Logical Level:** It builds the group by defining an interval on the sort key (*current and 2 preceding months*).

both of this solution can implement a variable window using PRECEDING, BETWEEN and FOLLOWING. Is also possible to define an UNBOUNDED window that change its size during that computation using only one fixed bound, useful for cumulative total. The difference is that the physical one not

look at missing rows and is possible to order by more than one keys. Although the logical can perform ordering only over numerical or data type where a distance can be defined and only by one key at the time. Another important characteristics is the possibility to compare detailed and total data without problem.

There is also a Ranking function that computes the rank of a value inside a partition, it could be of 2 types, **rank()** that computes the rank by leaving an empty slot after a tie, and **denserank()** that rank by leaving an empty slot after a tie. Of course this function requires an ordering although would be meaningless, the partition is not necessary because it can compute also over the whole table. Is important to notice that the order by of the over is not the visualizing order, but is only the order for the ranking, if a ordered visualization is needed an external group by is compulsory.

The main important extension is related to the *group by* that allows to compute multiple aggregation without performing separate query, this increase the efficiency of the whole system.

Using the rollup function over that computes multiple aggregates one at the time. The order in the expression is fundamental, it will compute the aggregate starting with all attribute and removing it one by one from right to left. The visualization of this computation have NULL value to represents the superaggregates, an example in figure 30. In this figure the first row with NULL

City	Month	Pkey	TotSales
Milano	7	145	110
Milano	7	150	10
Milano	...	...	...
Milano	7	NULL	8500
Milano	8	...	...
Milano	NULL	NULL	150000
Torino	...	...	150
Torino	...	NULL	2500
Torino	NULL	NULL	135000
...	...	...	...
NULL	NULL	NULL	25005000

Figura 30: Rollup with 3 values

present the total for the 7th month for Milano, the second one, with 2 NULL, is the total of all months for Milano, the last one, with 3 NULL, is the total

for all months, for all cities.

The **cube** function is used for computing all aggregates of all possible combination, in this case the order of the parameters is not important. This function is like performing more rollup at the time, is also well implemented respect software computation.