Data Management Notes

Giuliano Abruzzo

June 10, 2019

Contents

1 Cap 1: Introduction

The topics of the course are:

- **The structure of a Data Base Management System (DBMS)**: relational data and queries;
- **Transaction management**: concept of transaction, concurrency management;
- **Crash Management**: classification of failures, recovery;
- **Physical structures for data bases**: file organization for data base management, principles of physical database design;
- **Query processing**: evaluation of relational algebra operators, fundamentals of query optimization;
- **Advanced topics in data management (no SQL systems)**;

1.1 Relation Data Model

The **relational data model** uses the mathematical concept of a **relation** as the formalism for describing and representing **data**. A relation is a subset of a *Cartesian product* of *sets*, so it can be considered as a **table** with *rows* and *columns*. Mr. Codd introduced two different *query languages* for the relational data model: **Relational Algebra** and **Relational Calculus**.

- **Relational Algebra** is a **procedural language**: *queries* are expressed by applying a *sequence of operations* to relations.
- **Relation Calculus** is a **declarative language**: *queries* are expressed as *formulas of first-order logic.*

The **Codd's Theorem** says that **relational algebra** and **relational calculus** are *essentially equivalent* in terms of expressive power. **SQL** is an *hybrid* language that combines features from both *relational algebra* and *relational calculus.*

1.2 Relational Algebra

The operations of **relational algebra** can be divided in two groups:

- Three standard set-theoretic binary operations: **Union, Difference, Cartesian product**;
- Two special unary operations on relations: **Projection, Selection**;

The *relational algebra* consists of all expressions obtained by combining these five operations. A **k-ary relation** is a relation with k-places (finite number of places).

1.2.1 Union

- Takes as input **two k-ary relations** R and S, for some k;
- Gives as output the **k-ary relation** $R \cup S$ where:

  $R \cup S = \{(a_1, ..., a_k) : (a_1, ..., a_k)\ is\ in\ R\ or\ (a_1, ..., a_k)\ is\ in\ S\}$

  $R \cup R = R;\ R \cup S = S \cup R;\ R \cup (S \cup T) = (R \cup S) \cup T$
- So, all the elements of the **union** are contained in the two initial relations (R and S);

1.2.2 Difference

- Takes as input **two k-ary relations** R and S, for some k;
- Gives as output the **k-ary relation** $R - S$ where:

  $R - S = \{(a_1, ..., a_k) : (a_1, ..., a_k)\ is\ in\ R\ or\ (a_1, ..., a_k)\ is\ not\ in\ S\}$

  $R - R = 0;\ R - S \neq S - R;$
- So, all the elements of the **difference** are not contained in S, there are only elements of R;

1.2.3 Cartesian Product

- Takes as input an **m-ary relation** R, and an **n-ary relation** S;
- Gives as output **the (m + n)-ary relation** $R \times S$ where:

  $R \times S = \{(a_1, ..., a_m, b_1, ..., b_n) : (a_1, ..., a_m)\ is\ in\ R\ and\ (b_1, ..., b_n)\ is\ in\ S\}$

  $|R \times S| = |R| \times |S|;\ R \times S \neq S \times R;$

  $R \times (S \times T) = (R \times S) \times T;\ R \times (S \cup T) = (R \cup S) \times (R \cup T);$
- So, all the elements of the **Cartesian Product** are contained in the two initial relations (R and S);

EMPLOYE

EMP	DEPT
Rossi	A
Neri	B
Bianchi	B

DEPT

DEPT	CHAIR
A	MORI
B	BRUNI

EMPLOYE X DEPT

EMP	DEPT	CODE	CHAIR
ROSSI	A	A	MORI
ROSSI	A	B	BRUNI
NERI	B	A	MORI
NERI	B	B	BRUNI
BIANCHI	B	A	MORI
BIANCHI	B	B	BRUNI

Figure 1: Example of a Cartesian Product

1.2.4 Projection

- Is used when we want to **rearrange** the order of the columns and/or **suppress/rename** some columns;
- **Projection** has the form: $\pi_{<attribute\ list>}(<relation\ name>)$; or: $PROJ_{<attribute\ list>}(<relation\ name>)$;
- When we apply a **projection** operation on a relation R, it removes all *columns* whose attributes do not appear in the $<attribute\ list>$, as well as any duplicate *rows*. The remaining *columns* may be re-arranged according to the order in the $<attribute\ list>$;

1.2.5 Selection

- Is used when we want to extract some kind of information from the table;
- **Selection** has the form: $\sigma_{\Theta}(R)$ or $SEL_{\Theta}(R)$;
- R is a relation and Θ is a condition that can be applied to each row of R;
- When we apply a **selection** operation on a relation R, it returns the subset of R that contains all the rows that satisfy the condition Θ;
- A **condition** is an expression built from the comparison operators $(=, <, >, \neq, \leq, \geq)$ applied to operands that are constants or attribute names or components numbers and from boolean logic operators $(\wedge, \vee)$, **applied to basic clauses**, therefore you can apply the boolean conditions only to comparison operations (cioè fai *and* e *or* solo su operazioni di $=, <, >$, ..);

1.2.6 Relational Algebra Expression

The **relational algebra expression** is an expression obtained from relation schemas using *union, difference, cartesian product, projection and selection.*

1.2.7 Intersection

- Takes as input **two k-ary relations** R and S, for some k;
- Gives as output **the k-ary relation** $R \cap S$ where:

  $R \cap S = \{(a_1, ..., a_k) : (a_1, ..., a_k)\ is\ in\ R\ and\ (a_1, ..., a_k)\ is\ in\ S\}$

  $R \cap S = R - (R - S) = S - (S - R)$
- So, all the elements of the **Intersection** are contained in the two initial relations (R and S);

1.2.8 Θ-Join

- **Θ-Join** has the form: $\sigma_{\Theta}(R \times S)$ or $R\ JOIN_{\Theta}\ S$;
- **Θ-Join** selects those tuples from $R \times S$ that satisfies the condition Θ;
- If every tuple in $R \times S$ satisfy Θ then: $\sigma_{\Theta}(R \times S) = R \times S$;

1.2.9 Natural Join

- The **Natural Join** $\bowtie$ between two relations is essentially the **equi-join** on common attributes;
- Let $A_1, ..., A_k$ be the **common attributes** of two relations R and S, so we have:
- $R \bowtie S = \pi_{<list>}(\sigma_{R.A_1\ =\ S.A_1\ \wedge\ ...\ \wedge R.A_1\ =\ S.A_k}(R \times S))$;
- $< list >$ contains all *attributes* of $R \times S$ where duplicate *columns* are eliminated;

1.3 SQL

- **SQL** or *Structured Query Language* is the standard language for relational DBMSs;
- It basic construct is SELECT DISTINCT <*attribute list*> FROM <*relation list*> WHERE <*condition*>;

SQL	Relational Algebra
SELECT	Projection
FROM	Cartesian Product
WHERE	Selection

2 Cap 2: Buffer Management

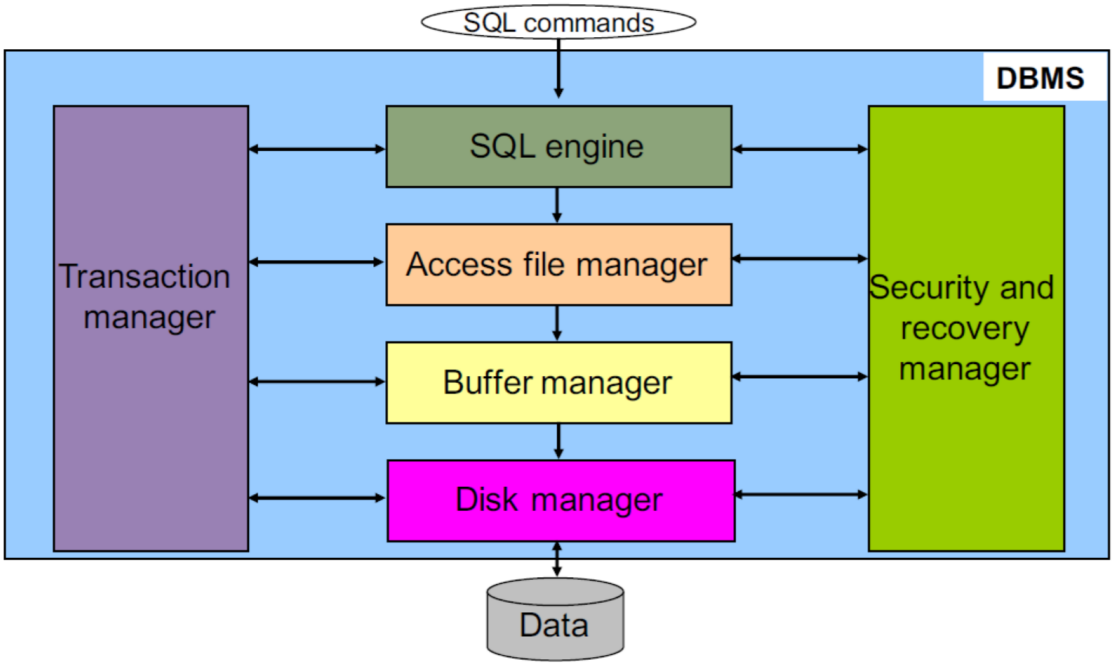


Figure 2: Architecture of a DBMS

2.1 Secondary Storage

At the physical level, a **database** is a set of *database files*, where each file is constituted by a set of **pages**, stored in physical blocks. Using a page requires to bring it in **main memory**. The size of a **block** (so of a *page*) is exactly the size of the portion of storage that can be transferred from **secondary storage** to **main memory** (and back).

2.2 The buffer

The **buffer** or **buffer pool** is a non-persistent memory space which constitutes that portion of the *main memory* used by the **DBMS** to process the pages of the *database*.

- The **buffer pool** is shared by all *transactions* and is used by the system;
- The **buffer pool** is organized in *frames*, that are *main memory pages*, whose size is that of a *block* (as said transfer unit from/to *secondary storage*), the typical size ranges from 2Kb to 64Kb;
- Since the **buffer pool** is in *main memory*, the management of their *pages* is more efficient with respect to the cost of the management of the *pages* of the *secondary storage*;

The **buffer manager** is responsible of the transfer of the pages from the *secondary storage* to the *buffer pool* and back. The **buffer manager** uses the *"Page Table"* data structure, that associates to each *page* appearing in the *buffer*, denoted by its *PID*, the *frame number* where the content of the page is stored. The **buffer manager** uses the following primitive operations:

- **Fix**: load a *page*;
- **Unfix**: releases a *page*;
- **Use**: uses a *page* in the buffer;
- **Force**: synchronous transfer to *secondary storage*;
- **Flush**: asynchronous transfer to *secondary storage*;

2.2.1 Fix operation

An *external module* issues the **Fix operation** in order to ask the *buffer manager* to load a specific *page* into a *frame* of the *buffer*, and for each *frame* the **buffer manager** maintains:

- The **information** about which page it contains (through the *Page Table*);
- The **pin-count**, so how many *transactions* use the *page* contained in the *frame*;
- The **dirty** bit, a bit whose value indicates whether the *page* contained in the *frame* has been modified (true) or not (false);

If a *module* M issues a **fix operation** that asks the *buffer manager* to load *page* P (denoted by its **PID**):

- The **buffer manager** checks if there is a *frame* F in the *buffer pool* that already contains the *page* P;
- If yes, it increments the **pin-counter** of the *frame* F;
- If not then:
  - A *frame* F' is chosen (if possible) for hosting *page* P according to a **replacement policy**;
  - If the **dirty** bit of the frame F' is true (so it has been modified, **steal strategy**), the *page* contained in F' is written back to *secondary storage*;
  - The **page** P is read from the *secondary storage* and is loaded in *frame* F'; the *pin-count* and the *dirty* bit are initialized to *0* and *false* respectively.
- The address of the **frame** that contains P is returned to module M;

2.2.2 Replacement policy

The *frame* for the **replacement** is chosen among those with **pin-count** $= \mathbf{0}$. If there are none, the request is placed in *queue* or is *aborted*. Otherwise there are several policies are possible:

- **LRU**: or *least recently used*, this is done through a queue containing the frames with **pin-count** $= \mathbf{0}$;
- **Clock Replacement**:
  - We use a variable **current** that points to *frames* in a circular fashion;
  - Each *frame* has a **referenced** bit that tells if the *page* contained in the *frame* has been used recently; this bit is set to *true* when **pin-count** $= \mathbf{0}$, this reflects the fact that the *page* is now a candidate for replacement, but it has recently used;
  - When we look for a *frame* for replacement we analyze the frame pointed by **current**;
  - If it has **pin-count** $> \mathbf{0}$ then we increment **current** and we repeat the procedure;
  - If it has **pin-count** $= \mathbf{0}$ and **referenced** $=$ **true** we set **referenced** to *false*, so this means that time has passed from the last usage of the corresponding page, and we increment **current** and we repeat the procedure;
  - If if has **pin-count** $= \mathbf{0}$ and **referenced** $=$ **false** then the *frame* is chosen for replacement;
- **Steal/No-Steal**: *steal* chooses the victim among those with **dirty** $=$ **true**, instead *no-steal* doesn't allow to choose the victim among those with **dirty** $=$ **true**;
- **Force/No-Force**: with **force** all *active pages* of a *transaction* are written in *secondary storage* when the transaction commits, instead with **no-force** the active pages of a *transaction* that has committed are written asynchronously in *secondary storage* through a *flush* operation ;

From a **recovery point of view**, *no-steal* and *force* are the most efficent strategies:

- With **no-steal**, we can avoid the undo of *transactions* that have *aborted*;
- With **force**, we can avoid the redo of *transactions* that have *committed*, but whose effects have not been registered yes because of a failure;

The problem is that with **no-steal** we are forced to keep many *active pages* in the *buffer pool*, and that with **force** the *input-output* operations tend to grow. So, the **steal, no-force strategy** is the most common cause it enables a more efficient *buffer management*;

2.2.3 Other Operations

- **Unfix**: the *transaction* certifies that it doesn't need the content of a specific *frame* anymore, and the **pin-count** of that *frame* is decremented;
- **Use**: the *transaction* modifies the content of a *frame* and the **dirty** bit is set to *true*;
- **Force**: **synchronous transfer** to *secondary storage* of the *page* contained in a *frame*, so the *transactions* waits for the completion of the operation;
- **Flush**: **asynchronous transfer** to *secondary storage* of the *pages* used by a *transaction*, the *operation* is executed when the *buffer manager* is not busy;

3 Cap 3: Concurrency

3.1 Transactions, Concurrency, Serializability

3.1.1 Transactions

A **transactions** models the execution of a software procedure constituted by a set of instructions that **may** *read from* and *write on* a database, and that form a **single logical unit**. We will assume that every *transaction* contains one **begin instruction**, one **end instruction**, one among **commit** and **rollback** (a *commit* confirms what you have done on a *database*, a *rollback* undo what you have done). We call **throughput** of a system the number of *transitions* per second accepted by the system, in a *DBMS* we want a *throughput* to be approximately 100-1000tps (*transitions* per second). In order to manage the **concurrency** the *DBMS* need to ensure the **isolation property** of the *transactions*. The desirable *properties* in *transaction management* are called **ACID** properties:

- **Atomicity**: for each *transaction* executed, either all of none of its actions have their effect;
- **Consistency**: each *transaction* execution brings the *database* to a *correct state*;
- **Isolation**: each *transaction* is independent of any other *concurrent transaction* executions;
- **Durability**: if a *transaction* execution succeeds then its effects are registered permanently in the *database*;

3.1.2 Schedule and serial schedules

Given a set of **transactions**: $T_1, T_2, ..., T_n$, a **sequence S** of executions of *actions* of such *transactions* respecting the **order** within each *transaction* (so that if action A is before action B in T_i, then A is before B also in S) is called **schedule** on $T_1, T_2, ..., T_n$.

- A *schedule* on $T_1, T_2, ..., T_n$ is called **partial** if it doesn't contain all the actions of all *transitions* $T_1, T_2, ..., T_n$;
- A *schedule* S is called **serial** if the actions of each *transaction* in S come before every action of a different *transaction* in S, so, if in S the actions of different *transactions* do not interleave;
- Two *schedule* are S_1 and S_2 are said to be **equivalent** if for each *database* state D, the execution of S_1 starting from D produces the same outcome as the execution of S_2 starting from D.;

A *schedule* S is **serializable** if for every initial state of the *database*, the outcome of its execution starting from such state is the same as the outcome of the execution of at least one *serial schedule* constituted by the the same *transactions* of S starting from the same *database* state. So a schedule S on $T_1, T_2, ..., T_n$ is serializable if there exists a serial schedule on $T_1, T_2, ..., T_n$ that is equivalent to S. A successful execution of **transaction** can be represented as a sequence of:

- Commands of type **begin/commit**;
- Actions that **read** and **write** an element in the *database*;
- Actions that **read** an **write** an element in the *local store*;

There are different types of **anomalies**:

- **Reading temporary data**: is called **WR anomaly** (write-read), it happen when a *transaction write* an element and another *transaction read* such element (the T_2 *transition* tries to *read* an element that the T_1 *transition* has not yet *written*);
- **Update Loss**: is called **RW anomaly** (read-write), it happen when a *transition reads* an element and another one *write* the same element, so we have an **update lost** on the execution (T_2 *transition* could change the value of an element that T_1 has *read* while the T_1 is still in progress, so maybe T_1 works on the element without knowing if T_2 has made some changes to the element);
- **Unrepeatable read**: another kind of **RW anomaly**, in this T_1 executes two consecutive *reads*, but due to the concurrent update of T_2, T_1 *reads* two different values;
- **Ghost update**: is called **WW anomaly** (write-write), this happened when two different *transactions* made two consecutive *write* on the same elements, at the end the two *transactions* find different values from those they had entered with their respective *write*;

3.1.3 Scheduler

The **scheduler** is part of the *transition manager* and it works as follows:

- It deals with new *transitions* entered into the *system*, assigning them an **ID**;
- It instructs the **buffer manager** so as to *read* and *write* on the **DB** according to a particular sequence;
- It is **not** concerned with specific operations on the *local store* of *transitions*, nor with constraints on the order of executions of *transactions* (so we ignore the operations on *main memory*);

We will be interested in two types of *algorithms*:

- **Algorithms for checking equivalence**: so given two *schedule*, we need to determine if they are **equivalent**;
- **Algorithms for checking serializability**: so given a *schedule*, we need to check if it is *equivalent* to any of the **serial schedule** on the same *transactions*;

We will work under two **assumptions**:

- No *transaction* **reads** or **writes** the same element **twice**, and no *transaction* **reads** an element that it has **written**;
- No *transaction* executes the **rollback** command;

3.2 View-serializability

Preliminary definitions:

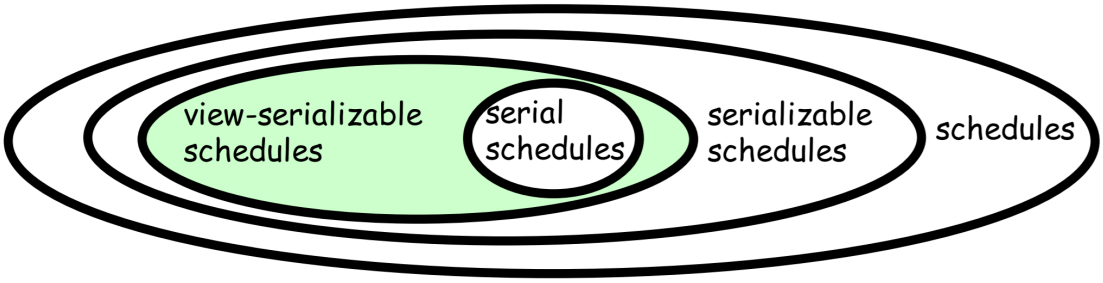
- In a *schedule* S, we say that $r_i(x)$ **reads-from** $w_j(x)$ if $w_j(x)$ precedes $r_i(x)$ in S, and there is no action of type $w_k(x)$ between $w_j(x)$ and $r_i(x)$;
- In a *schedule* S, we say that $w_i(x)$ is a **final-write** if $w_i(x)$ is the last write action on x in S;

Definition of **view-equivalence**:

- Let S_1 and S_2 be two **complete schedules** on the same *transactions*, then S_1 is **view-equivalent** to S_2 if S_1 and S_2 have the same **reads-from** and the same **final-write** set.

Definition of **view-serializability**:

- A *complete schedule* S on $T_1, T_2, ..., T_n$ is **view-serializable** if there exists a **serial schedule** S' on $T_1, T_2, ..., T_n$ that is **view-equivalent** to S.



Given two *schedules*, checking if they are **view-equivalent** can be done in *polynomial time*, while checking if a schedule S is **view-serializable** is an **NP-complete problem**, so for this *view-serializability* is not used in practice.

- For a **schedule** S, **Tran(S)** denote the set of **transactions** in S;
- For T $\subseteq Tran(S), \pi_T(S)$ denotes the **projection** of S onto T, that means the schedule S' obtained from S by deleting all operations of the transactions that are not in T.
  An example:
  $S = w1(x)\ r2(x)\ w2(y)\ r3(x)\ c1\ a2$ and $T = (t_1, t_2)$
  We obtain:
  $\pi_T(S) = w1(x)\ r2(x)\ w2(y)\ c1\ a2$

A class E of *schedules* is called **monotone** if the fact that S is in E implies that for all T $\subseteq Tran(S), \pi_T(S)$ is in E too, so E is closed under *projection*. From the definition of the **monotonicity** it follows that, if E is a class of *monotone schedules*, and a **partial schedule** constituted by a **projection** of a *schedule* S is not in E, then S is not in E too. So, a scheduler based on E can disregard a **partial schedule** s, if it's not in E, on the basis of the fact that no *extension* of s can be in E. Unfortunately, the class of *view-serializable schedule* is not **monotone**, so **Non-Monotonicity** is another reason why *view-serializability* is not used in practice.

3.3 Conflict-serializability

Two actions are **conflicting** in a *schedule* if they belong to *different transactions*, they operate on the **same element**, and at least one of them is a **write**. The **swap** of two consecutive *actions* of the same *transaction* can change the effect of the *transaction*.

- Two schedule S_1 and S_2 on the *same transactions* are **conflict-equivalent** if S_1 can be transformed in S_2 through a sequence of *swaps* of consecutive **non-conflicting actions.**
  - **Non-conflicting actions** like: *two consecutive reads* of the same element of different transactions, or *two consecutive read* of different elements by different transactions;
- A schedule S is **conflict-serializable** if there exist a *serial schedule* S' that is *conflict-equivalent* to S.

In order to check if a *schedule* is *conflict-serializable* we can analyze the **precedence graph** associated to a *schedule*. Given a schedule S on $T_1, T_2, ..., T_n$ the **precedence graph** $P(S)$ associated to S is defined as:

- The **nodes** are the *transactions* of the *schedule*;
- The **edge** (T_i, T_j) is in E if:
  - Exist two **actions** $P_i(A)$ *and* $Q_j(A)$ of different *transactions* and $P_i(A)$ appears before $Q_j(A)$ in S;
  - At least one between $P_i(A)$ *and*$Q_j(A)$ is a **write** operation;

THEOREM:

- A *schedule* S is **conflict-serializable** if and only if the **precedence graph** associated to S is **acyclic**.
- If two schedules S_1 *and* S_2 are **conflict-equivalent** then their *precedence graph* is equal: $P(S_1) = P(S_2)$; but this doesn't mean that if two *schedules* have the *same precedence graph* their are *conflict-equivalent*.

Given a *graph* G, the **topological order** of G is a **total order** S (a sequence) of the *nodes* of G, such that if the *edge* (T_i, T_j) is in the *graph* G, then T_i appears before T_j in the *sequence* S. So, if the graph G is **acyclic** then there exist at least one **topological order** of G. So the algorithm for check if a given schedule S is **conflict-serializable** is:

- Build the **precedence graph** P(S) of S;
- Check if P(S) is **acyclic** and return **true** if it is, **false** otherwise;

THEOREM:

- Let S_1 *and* S_2 be two *schedules* on the same *transactions*, if S_1 *and* S_2 are **conflict-equivalent** then they are **view-equivalent**.
- If a *schedule* S is **conflict-serializable** then is **view-serializable** too.

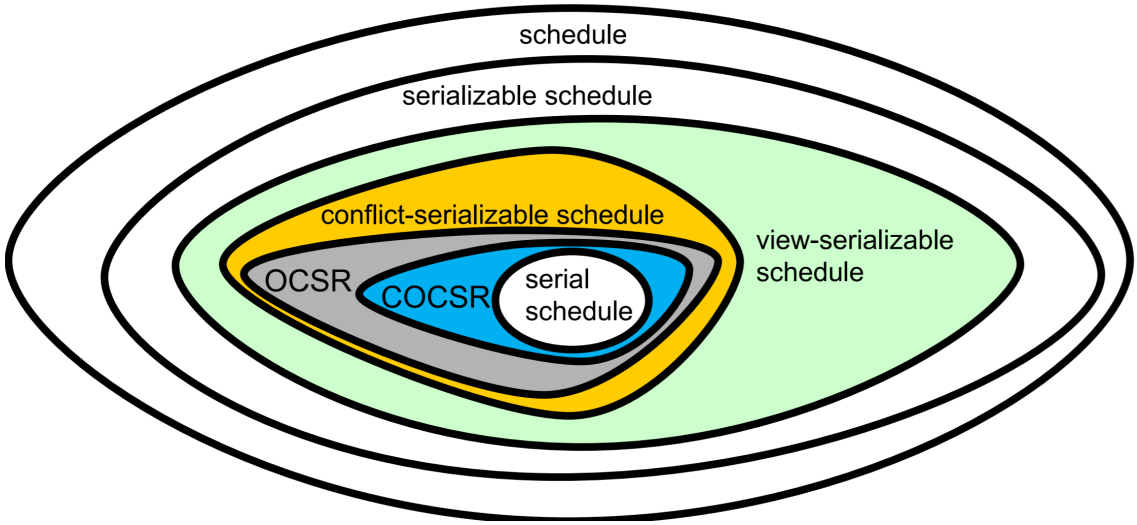
The *opposite* is **not valid**, in fact there are *schedules* that are *view-serializable* and not *conflict-serializable*. Contrary to **view-serializability** the class of **conflict-serializable** *schedules* is **monotone**. We say that a *schedule* S is in the class of **conflict-serializable** *schedules* if and only if for all T $\subseteq Trans(S), \pi_T(S)$ is in the class of **view-serializable** *schedules*. So, the class of *conflict-serializable schedules* is the **largest monotone subclass** of the class of *view-serializable schedules*. We denote as **CSR** the class of **conflict serializable schedules**.

A *schedule* S is **order preserving conflict serializable** if it is *conflict equivalent* to a *serial schedule* S', and for all $T, T' \in Trans(S)$: if T completely precedes T' in S then the same holds in S'. **OCSR** denotes the *class* of all *schedules* with this *property*, and the *theorem* says: **OCSR** $\subset$ **CSR**.

A *schedule* is **commit order preserving conflict serializable** it for all $T_i, T_j \in Trans(S)$: if there are *conflicting actions* $p \in T_i,\ q \in T_j$ *in* S such that p precedes q in S then c_i *precedes* c_j *in* S. **COCSR** denotes the *class* of *schedules* with this *property*, and the *theorem* says: **COCSR** $\subset$ **CSR**.

A *schedule* S is in **COCSR** if there is a *serial schedule* S' **conflict equivalent** to S such that for all $T_i, T_j \in Trans(S)$, T_i precedes T_j in S' if and only if c_i *precedes* c_j *in* S, and the *theorem* says: **COCSR** $\subset$ **CSR**. So in conclusion:

COCSR $\subset$ **OCSR** $\subset$ **CSR**



3.4 Concurrency through locks

In the methods based on **locks**, *a transaction* must *ask* and get a *permission* in order to operate of an *element* of the *database*, the *lock* is a mechanism for a transaction to ask and get such a **permission**. We introduce two new operations used to *request* and *release* the **exclusive use** of a *resource* (called A), where i stands for the *transaction* T_i that use the operations:

- **Lock** (*exclusive*): $l_i(A)$
- **Unlock**: $u_i(A)$

When we use the **exclusive locks**, *transactions* and *schedules* should obey **two rules**:

- Every *transaction* is **well-formed**.
  - A *transaction* T_i is **well-formed** if every *action* of T_i (*read* or *write* on A) is contained in a **critical section**, so it delimited by a pair of **lock-unlock** on A. (*Per ogni azione ce deve sta una coppia di lock-unlock*);
  - A *transaction* T_i is **ill-formed** if it is not *well-formed*;
- The *schedule* is **legal**.
  - A *schedule* S with locks is **legal** if no transaction in it locks an element A when a different transaction has granted the lock on A and has not yet unlocked A. (*Nessuna transizione deve tocca un elemento se non è ancora stato sbloccato da un altro*);

A *scheduler* based on **exclusive locks** behaves as follows:

- When an **action request** is issued by a *transaction*, the *scheduler* checks whether this **request** makes the *transaction* **ill-formed**, in which case the *transaction* is **aborted** by the scheduler;
- When a **lock request** on A is issued by a *transaction* T_i while another transaction has a lock on it, the *scheduler* doesn't grant the request and T_i is **blocked** until the other transaction *release* the lock on A;
- To trace all the *locks granted*, the *scheduler* manages a table of the locks called **lock table**;

A *schedule* S with *exclusive locks* follows the **two-phase locking protocol** (*2PL protocol*) if in each *transaction* T_i appearing in S, all *lock operations* **precede all** *unlock operations*. This *protocol* avoids the **ghost update** while accepting *concurrency*. In some cases the *scheduler* can **block** all the *transactions* and none can proceed, this is called **deadlock**. So we can add a new rule to the scheduler:

- Every *transaction* is **well-formed**.
- The *schedule* is **legal**.
- Each *transaction*, extended with the inserted lock/unlock commands, follows the **2PL protocol**.

A *scheduler* based on **exclusive locks** and **2PL** behaves as follows (*if the **lock/unlock** commands are automatically inserted the first two points do not occur*):

- If a *request* by a *transaction* T_i shows that T_i is not **well-formed**, then T_i is **aborted** by the scheduler;
- If a *lock request* by a *transition* T_i shows that T_i doesn't follow the **2PL** then T_i is **aborted** by the scheduler;
- If a **lock** is requested for A by a *transition* T_i while A is used by a different *transition* T_j, then the *scheduler* **blocks** T_i until T_j *release the lock* on A, and if the *scheduler* figures out that a **deadlock** has occurred or will occur then the *scheduler* adopt a method for **deadlock management**;
- To trace all the *locks granted*, the *scheduler* manages a table of the locks called **lock table**;

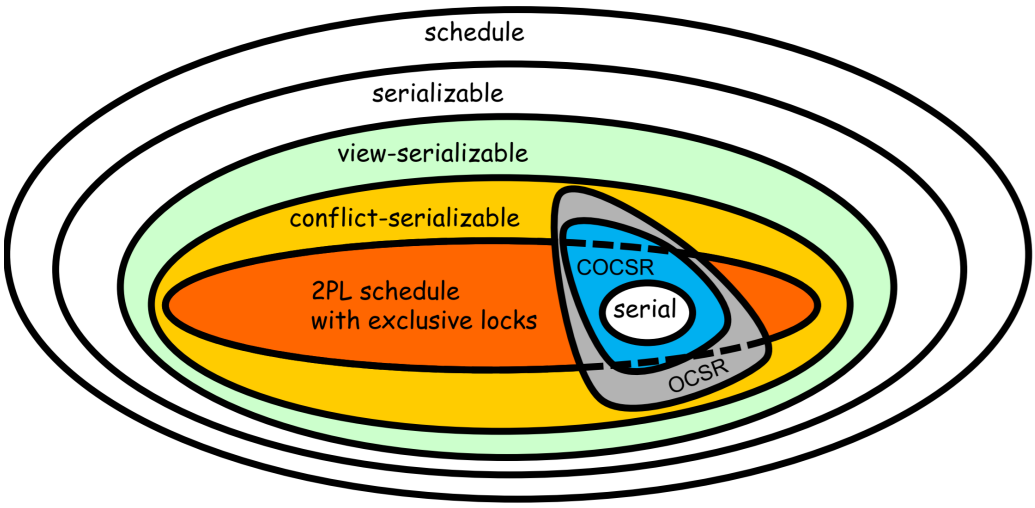
THEOREM:

- Every *legal schedule* constituted by *well-formed transactions* following the *2PL protocol* and with *exclusive locks* is **conflict-serializable**.

The theorem says that every *legal schedule* constituted by N **well-formed** *transactions* following the *2PL protocol* is **conflict-equivalent** to the *serial schedule* that orders the *transaction* in this order: from 1 to N-1 transition of the schedule takes the *transaction* that execute the first *unlock* operation in the remaining *transaction* in S (so from S, N-1, N-2, ... , to last 2) and at N takes the last *transition* left as the N^{th} *transition* of the *schedule*.

THEOREM:

- There exists a **conflict-serializable schedule** that doesn't follow the *2PL protocol* (with *exclusive locks*).



There is a different kind of lock, not restrictive as the **exclusive lock** (in fact two *read operations* do not create conflict) and it's called **shared lock**:

- $sl_i(A)$: **shared lock**;
- $xl_i(A)$: **exclusive lock**;
- $u_i(A)$: **unlock**

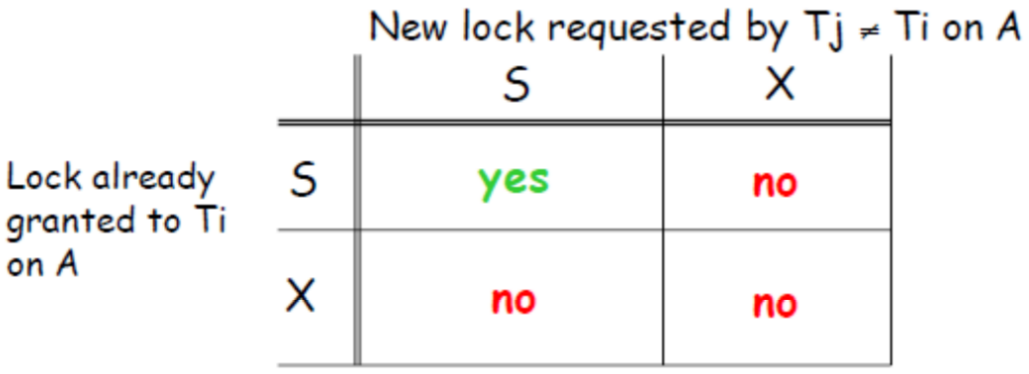
It is possible to upgrade a **shared lock** to an **exclusive lock** on the *same element* by the *same transition* without the *unlock* (of the *shared*) and this is called **lock upgrade**. With **shared** and **exclusive locks** the following rules must be respected:

- We say that a *transaction* T_i is **well-formed** if:
  - Every **read** $r_i(A)$ is preceded by a $sl_i(A)$ *or by* $xl_i(A)$, with no $u_i(A)$, so every *read operation* is preceded by a **lock**;
  - Every **write** $w_i(A)$ is preceded by a $xl_i(A)$ with no $u_i(A)$ in between, so every *write operation* is preceded by an **exclusive lock**;
  - Every **lock** $sl_i(A)$ *or by* $xl_i(A)$ of a *transaction* T_i is followed by an **unlock** on A by T_i;
- We say that a *schedule* S is **legal** if:
  - An **exclusive lock** $xl_i(A)$ is not followed by any other **lock** $xl_j(A)$ or by any $sl_j(A)$ (from a different transaction T_j) without an **unlock** $u_i(A)$ in between;
  - A **shared lock** $sl_i(A)$ is not followed by any other **exclusive lock** $xl_j(A)$ (from a different transaction T_j) without an **unlock** $u_i(A)$ in between;

With shared locks the **2PL (two-phase locking)** becomes:

- A *schedule* S follows the **2PL protocol** if in every *transaction* T_i of S, all *lock operations* (**exclusive** or **shared**) precede all **unlocking** operations of T_i;
- In other words, no *action* $xl_i(X)$ or $sl_i(X)$ can be preceded by an **unlock** $u_i(Y)$;

The scheduler uses the **compatibility matrix** for deciding whether a **lock request** should be granted or not. In this matrix $S = shared\ lock, X = exclusive\ lock, YES = request\ granted, NO = request\ not\ granted$.



The problem for the *scheduler* of automatically inserting the **lock/unlock commands** becomes more complex in the presence of *shared locks*, and also the execution of the **unlock commands** requires more work, in fact when an *unlock* is issued there may be several *transaction* waiting for a *lock* (*exclusive* or *shared*) and the *scheduler* must **decide** to which *transaction* grant the *lock* and there are several methods:

- **First-come-first-served**;
- Give priorities to the *transaction* asking for a **shared lock**;
- Give priorities to the *transaction* asking for a **exclusive lock**

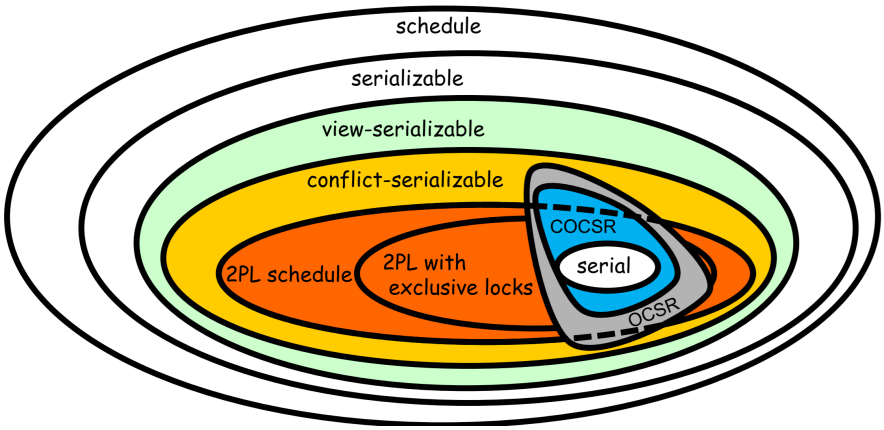
The **FCFS** is the most used one cause it avoids **starvation**, that is the situation in which a *request* of a *transaction* is never granted. The properties of two-phase locking with shared and exclusive locks are similar to the case of exclusive locks only:

THEOREM:

- Every *legal schedule* constituted by *well-formed transactions* following the *2PL protocol* and with *exclusive locks and shared locks* is **conflict-serializable**.

THEOREM:

- There exists a **conflict-serializable schedule** that doesn't follow the *2PL protocol* (with *exclusive locks and shared locks*).



Even with **shared locks** the risk of **deadlock**(two *transaction* need an *exclusive lock* on the element used by the other, so none proceed) is still present for example in: $sl_1(A), sl_2(A), xl_1(A), xl_2(A)$. The probability of a *deadlock* grows **linearly** with the *number of transactions* and **quadratically** with the *number of lock requests* in the *transactions*.

There are several techniques for **deadlock management**:

- **Timeout**;
  - The system fixes a **timeout** t after which a *transaction* waiting for a *lock* is **killed**. It is a very simple technique, but it t is *high*, the risk to be late in solving the problem, instead if t is *low*, too many *transaction* are *killed*. And there is another problem, the risk of **individual block** (the same transaction is killed several times);
- **Deadlock recognition**;
  - A **wait-for graph** is incrementally maintained: the **nodes** are the *transactions*, and the **edge** (T_i, T_j) means that T_i is waiting for T_j to release a *lock*. When a *cycle* appears in the graph, that means that there is a **deadlock**, and it's resolved by killing one of the involved *transactions*;
- **Deadlock prevention**;
  - Also called **wait-die**, in which to each *transaction* T_i a **priority** $pr(T_i)$ is assigned, in such a way that different *transactions* have different *priorities*, and the rule is: in case of conflict on a *lock*, T_i is allowed to wait T_j only if $pr(T_i) > pr(T_j)$ otherwise the *transaction* T_i is killed. In practice when a new *edge* (T_i, T_j) is added in the **wait-for** graph if $pr(T_i) \leq pr(T_j)$ then T_i is killed;

3.5 Recoverability of transactions

So far, we study under the assumption that no *transaction* are **rollbacked**. In fact, with **rollbacks** the notion of *serializability* that we have considered up to now is not sufficient for achieving the **ACID properties**. This fact is testified by an anomaly called **dirty read**, a **WR anomaly**. Let's consider two transactions T_1 and T_2 both with the same commands:

T_1	T_2
begin	begin
READ(A,x)	
x := x+1	
WRITE(A,x)	
	READ(A,x)
	x := x+1
rollback	
	WRITE(A,x)
	commit

$$READ(A, x),\ x := x + 1,\ WRITE(A, x)$$

Let's consider the following *schedule* (where T_1 executes the **rollback**), the problem is that T_2 reads a *value* written by T_1 before T_1 **commits** or **rollback**, so, T_2 reads a **dirty value**, that is shown to be *incorrect* when the *rollback* of T_1 is executed. The behavior of T_2 depends on a *incorrect input value*.

Recall that, at the end of *transaction* T_i:

- If T_i has executed the **commit operation**:
  - The *system* should ensure that the effects of the *transaction* are **recorded permanently** in the *database*;
- If T_i has executed the **rollback operation**:
  - The *system* should ensure that the effect of the *transaction* has **no effect** on the *database*;

It's important to note that the **rollback** of a *transaction* T_i can trigger the *rollback* of other *transactions*, in a **cascading mode**. In fact if a transaction T_j has read from T_i, we should kill T_j, and if another T_k has read from T_j... This is called **cascading rollback** and should be avoided cause it causes several *performance problems*.

3.5.1 Recoverable schedules

A schedule is **recoverable** if no *transaction* in S *commits* before all other *transaction* it has **read from**, *commit*. **Serializability** and **recoverabilty** are two orthogonal concepts, in fact there are *recoverable schedules* that are *non-serializable* and *serializable schedule* that are not *recoverable*. Every **serial schedule** is **recoverable**. In other words:

- S is **recoverable** if no *transaction* in S *commits* before the *commit* of all the *transactions* it has **read from**, *example*:
$$w_1(A)\ w_1(B)\ w_2(A)\ r_2(B)\ c_1\ c_2$$
- *Ogni transizione committa dopo di tutte le transizioni da cui ha letto (?).*

3.5.2 ACR schedules

Recoverable schedules can still suffer from the **cascading rollback problem**, and to avoid it, we need a stronger condition:

- A *schedule* S avoids *cascading rollback* (the schedule is **ACR**) if every *transactions* in S *reads* values that are *written* by *transactions* that have already **committed**, so an ACR schedule blocks the dirty data anomaly;
- S is **ACR** if no *transaction* **reads from** a *transaction* that has not *committed* yet, *example*:
$$w_1(A)\ w_1(B)\ w_2(A)\ c_1\ r_2(B)\ c_2$$
- Not all ***ACR*** *schedules* are **serializable**, and every **ACR** is **recoverable** and every *serial schedule* is **ACR**.
- So every *ACR schedules* blocks the **WR** *dirty read anomaly*;

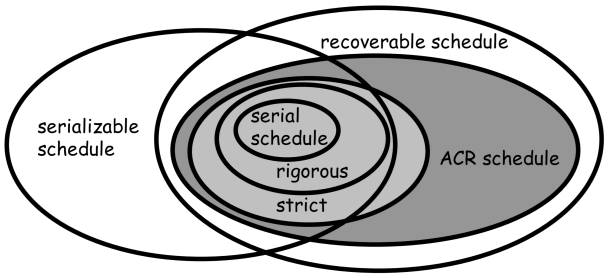
3.5.3 Strict schedules

In a *schedule* S, we say that a *transaction* T_i **writes on** T_j if there is a $w_j(A)$ in S followed by $w_i(A)$ and there is no *write* operations on A in S between these two *actions*.

- A *schedule* S is **strict** if every *transaction* **reads only** values written by *transactions* that have already *committed*, and **writes only** on *transactions* that have already *committed*;
- So, every **strict** schedule is ACR and when a transaction T_i rollbacks, it's immediate to determine which are the values that have to be stored back in the database to reflect the rollback of T_i, cause no transaction may have written on this values after T_i;
- Every *serial schedule* is **strict**, and every **strict** is *ACR* and so *recoverable*, but not all **ACR** *schedules* are *strict*;
- So every *strict schedules* blocks **WR** and **WW** operations;

3.5.4 Rigorous schedules

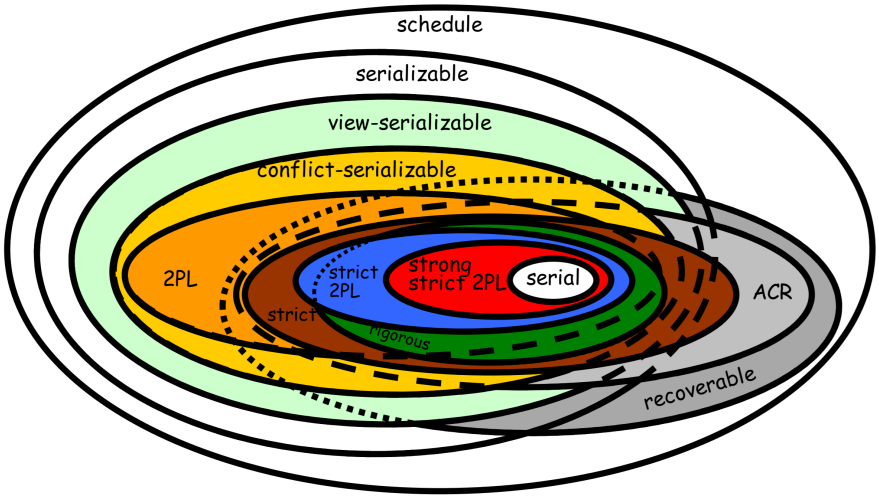
- A *schedule* is **rigorous** if for each pair of **conflicting actions** a_i (of T_i) and b_j (of T_j) appearing in S, the *commit* command c_i of T_i appears in S between a_i and b_j;
- Every *serial schedule* is **rigorous** and every **rigorous** is *strict*, and therefor *ACR* and *recoverable*, but not all *strict schedules* are *rigorous*;



3.5.5 Strict 2PL

A *schedule* S is said to be in the **strict 2PL** if S is in **2PL** and S is **strict**;

- A *schedule* S follows the **strict 2PL protocol** (S2PL) if it follows the *2PL protocol*, and all **exclusive locks** of every *transaction* T are kept by T until either T *commits* or *rollbacks*.
  - Every *schedule* following this protocol is **strict** and *serializable*.
- A *schedule* S follows the **strong strict 2PL protocol** (SS2PL) if it follows the *2PL protocol*, and all **locks** of every *transaction* T are kept by T until either T *commits* or *rollbacks*.
  - Every *schedule* S following this protocol is **rigorous**, *serializable* and the commit order of S is also a *conflict-serializablity order;*



3.6 Concurrency through timestamps

Each *transition* T has an associated **timestamp** $ts(T)$ that is unique among the *active transactions* and is such that $ts(T_i) < ts(T_j)$ whenever a *transaction* T_i arrives at *scheduler* before T_j. So, the timestamps actually define a total order on transactions. Every *schedule* respecting the **timestamp order** is **conflict-serializable** cause it's *conflict-equivalent* to the *serial schedule* corresponding to the *timestamp order*. The use of *timestamps* avoids the use of *locks*, but the **deadlock** may still occur. The basic idea is that at each action execution, the *schedules* checks whether the involved *timestamps* violates the **serializablity** condition according to the *order* induced by the *timestamps*. In particular for each element X we maintain the following data:

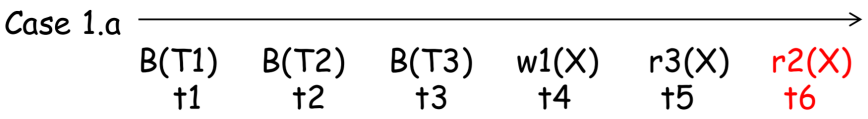
- *rts(X)*: the *highest timestamp* among the *active transactions* that have **read** X;
- *wts(X)*: the *highest timestamp* among the *active transactions* that have **written** X
- *wts-c(X)*: the *timestamp* of the last *committed transaction* that has **written** X;
- *cb(X)*: a bit called **commit-bit** that is *false* if the last transaction that **wrote** X has not *committed* yet, or *true* otherwise;

The rules are:

- The *actions* of *transaction* T is a *schedule* S must be considered as being **logically executed** in *one spot*;
- The **logical time** of an *action* of T, is the **timestamp** of T, $ts(T)$;
- The *commit-bit* is used in order to avoid the **dirty-read anomaly**;
- We assume that the **timestamp** so the **logical time** of each *transaction* coincide with the subscript: $ts(T_i) = i$, instead the **physical time** will be denoted as $t_1, ..., t_n$;

The system manages two *temporal axes*, the **physical time** and the **logical time**. The values of *rts(X)* and *wts(X)* indicate the *timestamp* of the *transaction* that was the last to *read* and *write* on X according to the *logical time*. An action of a *transaction* T executed at the **physical time** t is *accepted* if its *ordering* according to the *physical temporal* order is compatible with respect to the **logical time**: $ts(T)$. This principle of compatibility is checked by the *scheduler*.

3.6.1 Case 1a - Read ok



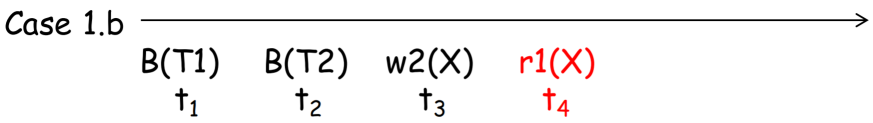
Consider $r_2(X)$ with respect to the **last write** on X, $w_1(X)$:

- The **physical time** of $r_2(X)$ is t_6 that is *greater* than the *physical time* of $w_1(X)$ that is t_4;
- The **logical time** of $r_2(X)$ is $ts(T_2)$ that is *greater* than the *logical time* of $w_1(X)$ that is $wts(X) = ts(T_1)$;

So there is **no incompatibility** between *logical* and *physical time*, so we proceed as follows:

- If $cb(X)$ is *true*, so if T_1 **committed**, the $rts(X)$ should be to the **maximum** between $rts(X)$ and $ts(T)$, in this example $rts(X) = ts(T_3)$ cause $ts(T_3)$ of operation $r_3(X)$ is greater than $ts(T_2)$ of operation $r_2(X)$, so $r_2(X)$ is executed and the schedules goes on;
- If $cb(X)$ is *false*, then T_2 is put in a **waiting state** for the *commit* or the *rollback* of the *transaction* that was the last to *write* X;

3.6.2 Case 1b - Read too late

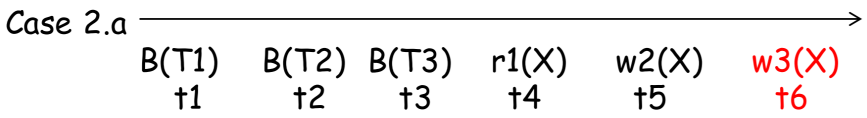


Consider $r_1(X)$ with respect to the **last write** on X, $w_2(X)$:

- The **physical time** of $r_1(X)$ is t_4 that is *greater* than the *physical time* of $w_2(X)$ that is t_3;
- The **logical time** of $r_1(X)$ is $ts(T_1)$ that is *less* than the *logical time* of $w_2(X)$ that is $wts(X) = ts(T_2)$;

So there is **incompatibility**, and the action of $r_1(X)$ *of* T_1 cannot be executed, so T_1 *rollbacks* and a new execution of T_1 starts with a new *timestamp*.

3.6.3 Case 2a - Write ok



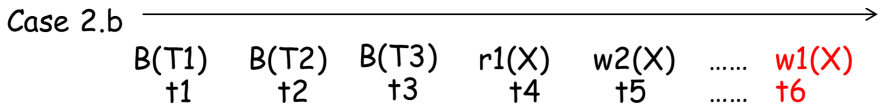
Consider $w_3(X)$ with respect to the **last read** on X, $r_1(X)$, and the **last write** on X, $w_2(X)$:

- The **physical time** of $w_3(X)$ is t_6 that is *greater* than the *physical time* of $r_1(X)$ and $w_2(X)$ (t_4 *and* t_5);
- The **logical time** of $w_3(X)$ is t_6 that is *greater* than the *logical time* of $r_1(X)$ and $w_2(X)$;

So there is **no incompatibility** between *logical* and *physical time*, so we proceed as follows:

- If $cb(X)$ is *true* and **no active transaction wrote** X: we set $wts(X) = ts(T_3)$, we set $cb(X) = false$ and the action $w_3(X)$ *of* T_3 is executed and the *schedule* goes on;
- If $cb(X)$ is *false*, then T_3 is put in a **waiting state** for the *commit* or the *rollback* of the *transaction* that was the last to *write* X;

3.6.4 Case 2b - Thomas Rule



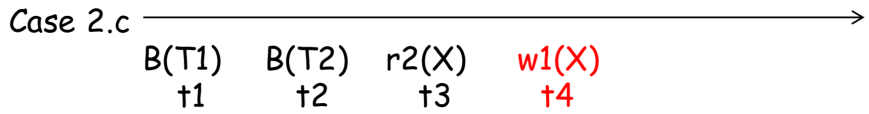
Consider $w_1(X)$ with respect to the **last read** $r_1(X)$ on X:

- The **physical time** of $w_1(X)$ is t_6 that is *greater* than the *physical time* of $r_1(X)$, t_4;
- The **logical time** is *equal* cause the two operations belongs to the same *transaction*;

So there is **no incompatibility** between *logical* and *physical time*, but on the **logical time dimension** $w_2(X)$ comes after the write $w_1(X)$ so the execution of $w_1(X)$ would correspond to an **update loss**. Therefor:

- If $cb(X)$ is *true*, we simply ignore $w_1(X)$ (not executed), in this way the effect is to *correctly* **overwrite** the value *written* by T_1 on X, with the value written by T_2 on X, so we pretend that $w_1(X)$ came before $w_2(X)$;
- If $cb(X)$ is *false*, then T_1 is put in a **waiting state** for the *commit* or the *rollback* of the *transaction* that was the last to *write* X;

3.6.5 Case 2c - Write too late



Consider $w_1(X)$ with respect to the **last read** $r_2(X)$ on X:

- The **physical time** of $w_1(X)$ is t_4 that is *greater* than the *physical time* of $r_2(X)$, t_3;
- The **logical time** of $w_1(X)$ is $ts(T_1)$ that is *less* than the *logical time* of $r_2(X)$, $rts(X) = ts(T_2)$;

So there is **incompatibility**, and the action of $w_1(X)$ *of* T_1 cannot be executed, so T_1 is **aborted**, and a new execution of T_1 starts with a new timestamp.

3.6.6 Deadlock with timestamps

The methods of *timestamps* doesn't avoid the risk of **deadlock**, but the probability is *lower* than in the case of *locks*. The *deadlock* is related to the use of the **commit-bit**, and the **deadlock problem** is handled with the same techniques used in *2PL method*.

3.6.7 Conflict-serializability and timestamps

There are **conflict-serializable** *schedules* that are not accepted by the *timestamp-based scheduler*.

- If the *schedule* S is *processed* by the **timestamp-based scheduler** without using the **Thomas rule**, then the schedule obtained from S by removing all *actions* of *rollbacked transaction* is **conflict-serializable**;
- If the *schedule* S is *accepted* by the **timestamp-based scheduler** using the **Thomas rule**, then S may be not **conflict-serializable**;
- If the *schedule* S is *accepted* by the **timestamp-based scheduler** using the **Thomas rule**, then the *schedule* obtained from S by removing all *actions* ignored by the *Thomas rule* and all *actions* of *rollbacked transaction* is **conflict-serializable**;

3.6.8 Timestamps and 2PL

- There are *schedules* that are accepted by *timestamp-based schedulers* that are **not in 2PL**;
- There are *schedules* that are accepted by *timestamp-based schedulers* and are also **strict 2PL**;
- There are **strong strict 2PL schedules** that are not accepted by *timestamp-based scheduler*;

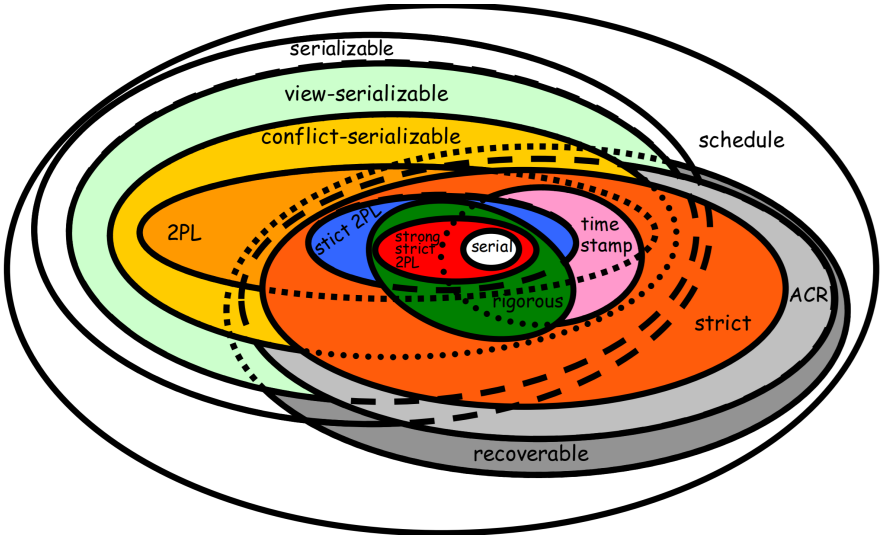


Figure 3: All types of schedules including timestamp

	waiting stage	serialization order	need to wait for commit	deadlock
2PL	yes	by conflicts	solved by SS2PL	risk
TS	killed and restarted	by timestamps	buffering write actions waiting cb(X)=true	less probable

Figure 4: Comparison between Timestamps and 2PL

3.6.9 Multiversion timestamp

The idea is to **don't block the *read action***, and this is made by introducing *different version* of an element X: $X_1, ..., X_n$, so that every *read* can be always be executed, provided that the right version is chosen (according to the *logical time* determined by the *timestamp*):

- Every **legal write** $w_i(X)$ generates a **new version** X_i;
- To *each version* X_h of X the *timestamp* $wts(X_h) = ts(T_h)$ is *associated*, denoting the **timestamp** of the transaction that **wrote** that version;
- To *each version* X_h of X the *timestamp* $rts(X_h) = ts(T_h)$ is *associated*, denoting the **highest timestamp** among those *transactions* that **read** X_h;

The *scheduler* uses **timestamps** as follows:

- When executing $w_i(X)$, if a *read* $r_j(X_k)$ such that $wts(X_k) < ts(T_i) < ts(T_j)$ already occurred, then the **write is refused** (cause a T_j older than T_i has already *read* a version of X that precedes X_i), otherwise the **write is executed** on a new version X_i of X and $wts(X_i) = ts(T_i)$;
- When executing a $r_i(X)$, the *read* is executed on the *version* X_j such that $wts(X_j)$ is the **highest write timestamp** among the versions of X, and **less or equal** to $ts(T_i)$, so $wts(X_j) \leq ts(T_i)$ and there is no version X_h such that $wts(X_j) < wts(X_h) \leq ts(T_i)$;
- For X_j with $wts(X_j)$ such that **no active transaction** has *timestamp* **less** than j, the version of X that precede X_j are deleted from *oldest* to *newest*;
- To ensure *recoverability*, the commit of T_i is delayed until all *commits* of the *transactions* T_j that *wrote* versions *read* by T_i are executed;

The *scheduler* uses suitable **data structures**;

- For each version X_i the *scheduler* maintains a $range(X_i) = [wts, rts]$ where **wts** is the *timestamp* of the *transaction* that *wrote* X_i, and **rts** is the *highest timestamp* among those of the *transaction* that *read* X_i (if none, $rts = wts$);
- We denote the *set*: $ranges(X) = \{range(X_i) \mid X_i \; is \; a \; version \; of \; X\}$
- When $r_i(X)$ is processed, the *scheduler* uses the set $ranges(X)$ to find the version X_j such that $range(X_j) = [wts, rts]$ where **wts** is the highest, and less or equal to the *timestamp* of T_i, and if $ts(T_i) > rts$ then the rts of $range(X_j)$ is set to $ts(T_i)$;
- When $w_i(X)$ is processed, the *scheduler* uses the set $ranges(X)$ to find the version X_j such that $range(X_j) = [wts, rts]$ where **wts** is the highest, and less or equal to the *timestamp* of T_i, and if $rts > ts(T_i)$, then $w_i(X)$ is **rejected**, else is **accepted** and the version X_i with $range(X_i) = [wts, rts]$ with $wts = rts = ts(T_i)$ is **created**;

4 Cap 4: Recovery

The **transaction manager** is mainly concerned with *isolation* and *consistency*, while the **recovery manager** is mainly concerned with *atomicity* and *persistency*. In fact the recovery manager is responsible for:

- **Beginning** the execution of *transactions*;
- **Committing** *transactions*;
- Executing the **rollback** of *transactions*;
- Restore a **correct state** of the *database* following a *fault condition*;

It use a special *data structure* called **log file**. **Failures** can be of two types:

- **System failures**: system crash, system error or application exception, local error condition of a transaction, concurrency control;
- **Storage media failures**: disk failures, catastrophic events;

The **log file** records the *actions* of the various *transactions* in a **stable storage** (*failure resistant storage*). *Read* and *write* operations on the **log** are executed as the *operations* on the *database* (on the *buffer*), and writing on the *stable storage* is done generally with *force operation*. Of course *stable storage* is an **abstraction**, in fact *stability* is achieved through **replication**. The **log** is a *sequential file* assumed to be *failure-free*. The operations on the *log* are:

- **Append a record** at the end;
- **Scan sequentially forward**;
- **Scan backward**;

The *log* records the *actions* of the *transactions* in **chronologically order**, and there are two types of records in the *log*:

- **Transaction records** (*begin, insert, delete, update, commit, abort*);
- **System records** (*checkpoint, dump*);

It's important to not confuse **transaction actions** with the **actions on the secondary storage**, in fact the *actions* of the *transaction* are executed when they are recorded in the *log* even if their effects are not yet registered in the *secondary storage*.

4.1 Transaction Records

- **O** = Element of the database;
- **AS** = After State, value of O after an operation;
- **BS** = Before State, value of O before an operation;

For each *transaction* T, the **transaction records** are stored in the **log** as follows;

- **Begin**: $B(T)$;
- **Insert**: $I(T, O, AS)$;
- **Delete**: $D(T, O, BS)$;
- **Update**: $U(T, O, BS, AS)$;
- **Commit**: $C(T)$;
- **Abort**: $A(T)$;

4.2 System Records

4.2.1 Checkpoint

The goal of the **checkpoint** is to register in the *log* the set of **active transaction** $T_1, ..., T_n$, so as to *differentiate* them from the **committed transactions**. The **checkpoint** CK operation executes the following *actions*:

- For each *committed transaction* after the last *checkpoint*, their **buffer pages** are copied into the *secondary storage* through **flush**;
- A *record* $CK(T_1, ..., T_n)$ is written on the **log** through **force**, where $T_1, ..., T_n$ are all *active transactions* **uncommitted**;

It follows that, for each *transaction* T such that $Commit(T)$ precedes $CK(T_1, ..., T_n)$ in the log we can avoid the **redo** in case of *failure*. This operation is executed *periodically* with **fixed frequency**.

4.2.2 Dump

The **dump** is a copy of the **entire state of the database**, and is executed *offline* (so all the *transactions* are suspended). It produces a **backup**, that is saved in a *stable storage*. This operation write through *flush* a *dump record* in the *log*.

4.3 Undo and Redo operations

- The **undo operation** restore the *state* of an *element* O at the **time preceding** the execution of an *action*;
- The **redo operation** restore the *state* of an *element* O at the **time following** the execution of an *action*;

The outcome of a *transaction* is established when either the **Commit**(**T**) record or the **Abort**(**T**) record is written in the **log**. The $Commit(T)$ is written *synchronously* from the *buffer* to the *log*, while the $Abort(T)$ is written *asynchronously*. when a **failure** occurs for a *transaction* that is **uncommitted** we need to **undo** the *actions* to ensure *atomicity*, instead if is **committed**, we need to **redo** the *actions* to ensure *durability*.

4.4 Writing records in the log

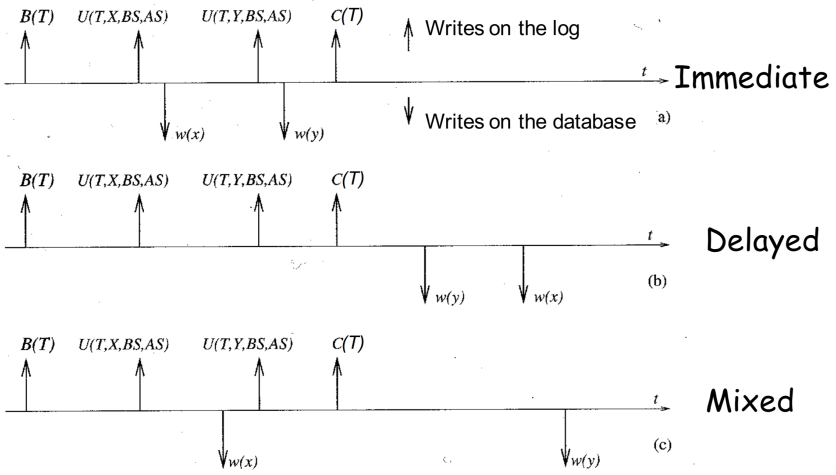
The **recovery manager** follows two *rules*:

- **WAL** or **write-ahead log**:
  - The *log records* are written from the **buffer** to the **log** before the **corresponding records** are *written* in the *secondary storage*;
  - This is important for the *effectiveness* of the **Undo operation**, cause the old values can always be *written back* to the *secondary storage* by using the *BS value*. In other words, **WAL** allows to **undo write operation** executed by *uncommitted transaction*;
- **Commit-Precedence**:
  - The *log records* are written from the **buffer** to the **log** before the *commit* of the *transaction*;
  - This is important for the *effectiveness* of the **Redo operation**, cause if a *transaction* committed before a *failure*, but its *pages* have not been written yet in *secondary storage* we can use *AS value* to **write** such pages. In other words, **Commit-Precedence** allows *committed transactions* whose *effect* have not been registered yet in the *database* to be **redone**.

4.5 Writing records in secondary storage

For each operation of **Update, Insert, Delete**, the *recovery manager* must decide on the strategy for *writing* in the *secondary storage*, and there are three possible methods, all coherent with the **WAL** and the **Commit-Precedence** rules:

- **Immediate effect**:
  - The **update operations** are executed *immediately* on the *secondary storage* after the *corresponding records* are *written* in the *log*;
  - The *buffer manager writes* the effect of an operation by a *transaction* T on the *secondary storage* **before** *writing* the **commit record** of T in *log*;
  - So all pages of the *database* modified by a *transaction* are certainly written in the *secondary storage*, and **Redo is not needed;**
- **Delayed effect**:
  - The **update operations** by a *transaction* T are executed on the *secondary storage* only after the *commit* of the *transaction*, so only after the *commit record* of T has been written in the *log*, and **Undo is not needed**;
  - The **log records** are *written* in the *log* before the *corresponding data* are *written* in *secondary storage*;
- **Mixed effect**:
  - For an *operation* O, both *immediate effect* and *delayed effect* are possible, depending on the choice of the *buffer manager*, and both **Redo and Undo needed**;



4.6 Warm, Cold restart

Depending on the type of *failure* we have two different types of *restart*:

- **Warm Restart**: in case of *system failure*;
- **Cold Restart**: in case of *disk failure*;

In **warm restart**, we will assume the *mixed effect strategy*, and is constituted by five steps:

- We go *backward* through the *log* until the **most recent checkpoint record** in the *log*;
- We set $s(Undo) = \{active\ transaction\ at\ checkpoint\}$, $s(Redo) = \{\}$;
- We go *forward* through the *log* adding to $s(Undo)$ the *transaction* with the corresponding *begin record*, and moving those with the *commit record* to $s(Redo)$;
- In the **undo phase**, we go *backward* through the *log* again, *undoing* the *transaction* in $s(Undo)$ until the *begin record* of the *oldest transaction* in the set of *active transactions* at the **last checkpoint**;
- In the **redo phase**, we go *forward* the *log* again, *redoing* the *transaction* in $s(Redo)$;

Instead **cold restart** is constituted by three phases:

- Search the most recent **dump record** in the log, and load the *dump* into the *secondary storage*;
- We reapply all *actions* in the *log* in the order determined by the *log*;
- We obtain the *database state* before the *crash* and we execute the **warm restart** procedure;

5 Cap 5: File Organization

5.1 Pages and records

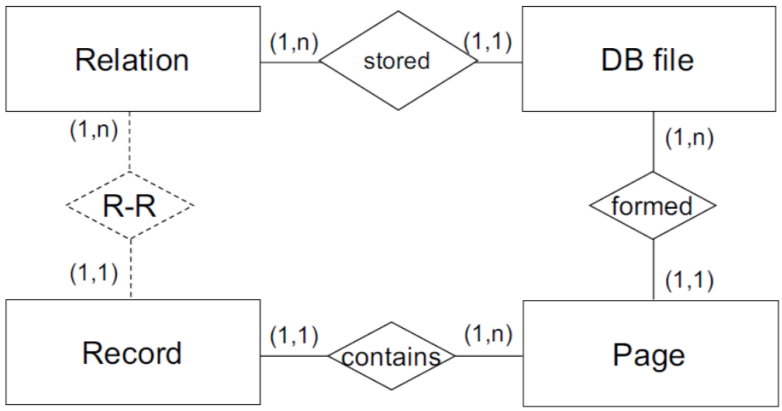
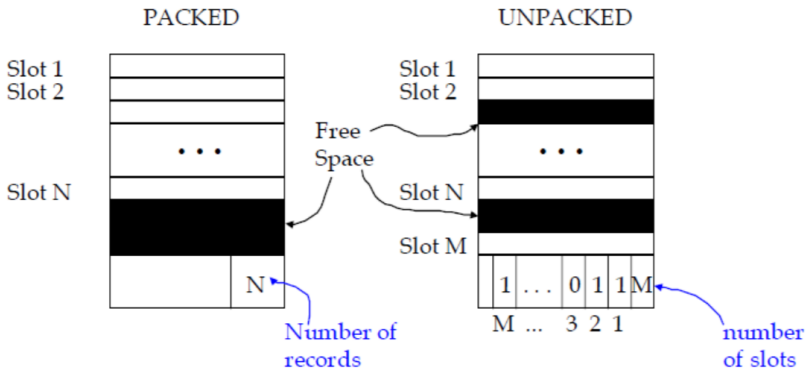


Figure 5: Note that R-R is a derived relationship

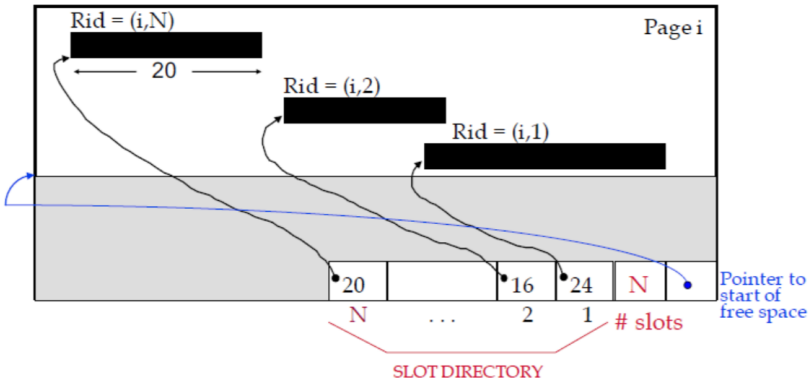
A **page** has a dimension of a *block* (unit of transfer from/to *buffer pool*), has an *address* (or **page id**) and is physically constituted by a set of **slots**, each one with an *address*; the page may contain a *header*, with *pointers* to other *pages*. A **slot** is a memory space that may contain *one record*, and has a **slot id** that identifies it in the context of the *page*. Each **record** has an identifier called *record id* or **rid**: $rid = < page\ id,\ slot\ number >$. Usually a **record** has also a *header*.

5.1.1 Page with fixed length records



- **Packed organization**: moving a *record* changes its *rid*;
- **Unpacked organization**: identifying a *record* require to access the *bit array* to check whether the *slot* is free (0) or not (1);

5.1.2 Page with variable length records



No problem in moving **record** to the *same page*. **Deleting** a *record* means to set to -1 the value of the corresponding *slot*, and move the *record space* to the *free space*. When a **record** moves to *another page*, we can store in the old position the *address* of the **new position** (example: switch the $rid = (i, 2)$ to $rid = (j, h)$ where j is the *new page*, and h is the *position* in the page j).

5.1.3 Format of a record

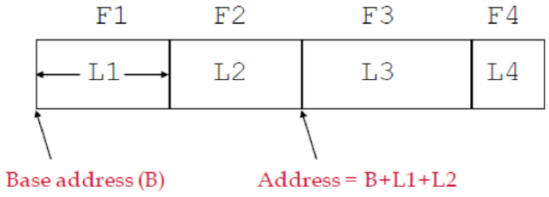


Figure 7: Fixed length record

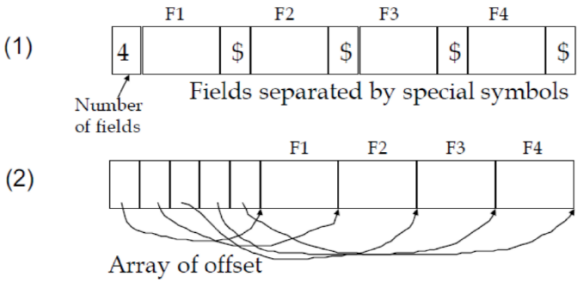


Figure 8: Variable length record

In the **variable length record**, the (2) alternative allows direct access to the *fields*, and efficient *storage of nulls* (two consecutive pointers that are equal correspond to a null).

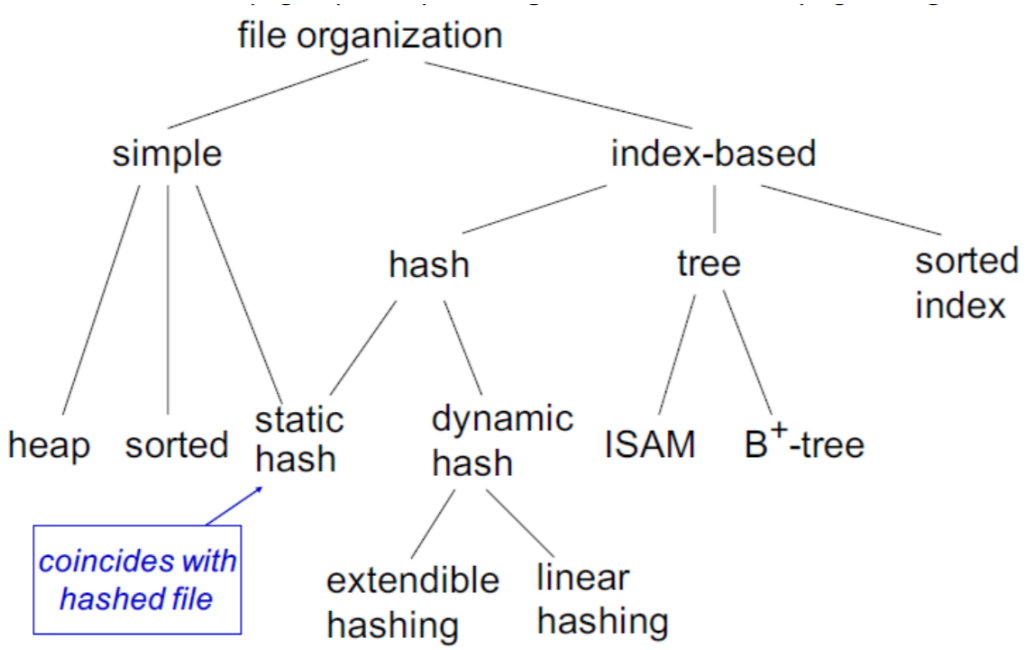
If a **record** doesn't fit in a *page*, the *record* is split in **fragments**, each on with suitable size as to fit in one page. A *record* with more than one *fragment* is called **spanned**. Every *record* and every *fragment* requires some extra *info*:

- A *bit*, that will tell if it is a **fragment** or a **whole record**;
- If it is a **fragment**, a *bit* that will tell if is the *first* of the *last fragment* of the **record**;
- **Pointers** to *previous* and *next fragment*, if such *fragments* exist;

5.2 Simple file organizations

A **file** is a collection of *pages*, each one containing a collections of *record*. Usually one file is used for one relation, but there are cases in which a file is used for more than one relation. A file organization should support the following operations:

- **Insert/Delete/Update** a *record* in one *page* of the collection;
- **Read** the *record* specified by its *rid*;
- **Scan** all the *records* in all its *pages*, and possibly satisfying some condition;



5.2.1 Heap File

In the **heap file organization** the *file* representing the *relation* contains a set of *pages*, each one with a set of *records*, with **no special criterion or order** for both the *pages* and for the *records*. When the *relation* grows or shrinks, *pages* are allocated or de-allocated. In order to support the operations, it is necessary to keep track of:

- The **pages** belonging to the *file*;
- The **records** in the *pages* of the *file*;
- The **free space** in the *pages* of the *file*;

We can represent *heap files* in two way:

- **Lists**;
  - In **list-representation** when a new *page* is needed, the request is issued to the *disk manager*, and the *page* returned is put as a **first page** of the *list* with *free space*, instead when a *page* is not used anymore (so it's constituted by only free space) it's deleted from the *list*.
- **Directory**;
  - In **directory-representation**, the *directory* is a *list of pages*, where each *entry* contains a *pointer* to a *data page*. Each *entry* in the *directory* refers to a **page** and tells how much *free space* the *page* has.

Searching for a *page* with **free space** is more *efficient* in **directory-based representation** cause the number of *page* accesses is linear with the size of the *directory*, instead in **list-based representation** with the size of the *relation*.

5.2.2 File with sorted pages

The **records** are *sorted* within each *page* on a set of *fields*, a **search key**, while the *pages* are *sorted* according to the *sorting* of their *records* and are stored contiguously in a **sequential structure**, where the order of the *pages* reflects the sorting of the *records*.

5.2.3 Hashed File

The *pages* of the *relation* are organized into *groups*, where each group is called **bucket**. Each *buckets* consist of one *page* called **primary page**, and other *pages* called **overflow pages** linked to the *primary page*. A set of fields of the *relation* is chosen as the **search key**, when we are searching for a *record* R with a given value k for the *search key*, we can compute the *address* of the *bucket* containing R by the application of an **hash function** to k. Given a fixed number of **primary pages** N (so N is the number of the buckets), *sequentially allocated*, *never de-allocated* and with *overflow pages* if needed, the **address** of the *bucket* containing *record* with *search key* k is: $h(k)\ mod\ N$.

5.2.4 Cost Model

We provide an estimation of the **execution time** of the basic operations for the three **simple file organizations**. We will concentrate only on the **page access cost**, so, we will be interested in characterizing the *cost of an operation* for a certain *file organization* in terms of the **number of page accesses** *required* by the execution of the *operation* (c*e frega de sape solo quanti accessi a pagine per ogni operazione*). We define:

- **B**: number of *pages* in the file;
- **R**: number of *records* per page;
- **D**: time for writing/reading from/to secondary storage;
- **C**: time for processing one record (e.g. comparing a field with a value);

The operations on *data* and their *cost* are:

- **Scan** the *records* of the file;
  - Cost of loading the pages of the file;
  - CPU cost for locating the records in the pages;
- *Selection* based on **equality**;
  - Cost of loading the pages with relevant records;
  - CPU cost for locating the records in the pages;
  - The record is unique if equality is on relation key;
- *Selection* based on **range of values**;
  - Cost of loading the pages with relevant records;
  - CPU cost for locating the records in the pages;
- **Insertion** and **Deletion** of a *record*;
  - Cost of locating the page where the operation must occur;
  - Cost of loading and modification and writing back the page;
  - Cost of loading and modification and writing of other pages if needed;

	Heap File	Sorted File	Hashed File
Scan	B(D + RC)	B(D + RC)	1.25 B(D + RC)
Equality selection	B(D + RC)	1 + D $\log_2$B + C $\log_2$R binary D($\log_2\log_2$B) + C log2R interpolation avg	(D + RC) x #n° overflows
Range selection	B(D + RC)	D $\log_2$B + C $\log_2$R + (f_s x B – 1)(D + RC)	1.25 B(D + RC)
Insertion	D + C + D	1 + D $\log_2$B + C $\log_2$R + C + 2B(D+RC)	2D + RC
Deletion	D + C + D if rid B(D + RC) + XC + YD if selection	1 + D $\log_2$B + C $\log_2$R + C + 2B(D+RC)	cost of search + D + RC

Figure 6: Where X number of records to be deleted, Y number of pages with records to be deleted

5.2.5 Sorting

Sorting of data in *secondary storage*, also called **external sorting**, is very different from *sorting* an array in *main memory*, called **internal sorting**. We need to minimize disk I/Os, so we don't use the normal algorithms for sorting (like *quicksort*, *mergesort*, ...), in fact, if we have a *file* with N records, a *mergesort* requires $O(N\ log_2\ N)$ comparison, we don't want to go to the disk all these times. **Sorting** data in *secondary storage* is based on **pages** not on *records*.

5.2.6 2-way merge-sort

The steps of the **2-way sort** are:

1. Sort each page;
2. Merge two pages into one run;
3. Merge two runs into one run;
4. ...
5. Sorted!

So, if we have B pages in the file:

- the **total number of passes** is: $(\lceil log_2\ B \rceil + 1)$;
- the the **total cost** is $2 \times B(\lceil log_2\ B \rceil + 1)$;

And this is way better than the plain **old merge sort** cause $N >> B$. The **2-way merge sort** only uses 3 *frames* in the *buffer*, 2 for holding the *input record* and 1 for holding the *output records*. The idea now is: each pass, read as much data as possible into *buffer*, so we reduce the *number of passes*.

5.2.7 Multipass merge-sort

Suppose we have F *free frames* in the *buffer* that we can use for sorting **relation** R.

- Repeat until **no more pages** in R (**Pass 0**);
  - Bring F *pages* of *relation* R in the *buffer*, sort their *records*, and write the corresponding *pages* in a file called **run** (or *sorted sublist*);
- Repeat until we have just one **run**;
  - Repeat until we don't have runs to consider (**Pass i**);
    - (**Merge**) We **merge** the *records* of the first Z *runs* (where $Z < F$) not yet analyzed into a single new *run*. This is done by reading the *pages* of the Z *runs*, using one *frame* for each run, in fact when we use *frame i* we load in the *frame* the next *page* of the i-th run and we write the result in the *output file* (**new run file**), one *page* a time using one frame in the buffer.

If B is the number of *pages* of the *relation* R, and we use F *frames* of the *buffer*, then the number of **run** initially created is $S = B/F$. Each time we enter in the *repeat loop* is called a **pass**. The number of **passes** depends on Z (whose value is usually $F - 1$, cause we need $Z + 1$ *frames* at each execution of the *inner loop*). At pass i, the number of **runs** produced is S/Z^i. At every pass we *read* and *write* every *page* of the *relation*, so the **cost** in terms of *page accesses* is:

$$\mathbf{2 \times B(\log_Z S + 1) = 2 \times B(\log_Z (B/F) + 1)}$$

If $Z = F - 1$ we can approximate and the **cost** is: $\mathbf{2 \times B(\log_F B)}$

5.2.8 Recursive formulation of the multipass

- Base step:
  - If R **fits** in the F available in the buffer (so, $if B(R) \leq F$), then *sort* R in the *buffer*, using any **main memory algorithm** and *write* the *sorted* relation to the *secondary storage*.
- Inductive step:
  - If R **doesn't fits** in the *buffer*, we *partition* the pages of R, into $F-1$ groups: $R_1, ..., R_{F-1}$ and we **recursively sort** R_i for each $i = 1, 2, ..., F-1$. Then we *merge* the $F-1$ *sorted sublists* using one frame for the output and *write* the *sorted* relation to the *secondary storage*.

The problem is that this algorithm suffer of the problem that the CPU must wait for I/O. There are two possible improvement:

- **Double buffering**:
  - We keep a **second set of buffer frames**, and we process one set while waiting for disk I/O to fill the other set.
- **Replacement/Selection**:
  - The idea is to minimize the number of *initial runs* computed at pass 0 (so $S = B/F$ smaller) by trying to produce *runs of bigger size* through **priority queue** (also called *heap*, a *binary tree* where each node has a value than is less than the values of the children). With *random data* this methods increases the **average run size** by a factor of 2, so it has the same effect of *doubling the size of the internal sorting buffer*.
  - We build a **priority queue 1** in the *buffer* by reading the *records* from R until there is no more space in the *buffer*, and we prepare a **priority queue 2** initially *empty* in the *buffer*;
  - We repeat the following:
    - If queue 1 and 2 are *empty* we stop;
    - If queue 1 is *empty* we move queue 2 into 1, we *empty* queue 2 and we start a **new run**;
    - While queue 1 is *not empty*, we write the **minimum value** of the queue 1 into the **current run**, and we *remove* it from queue 1. If there are still *records* in R then we read a *new record* from R and if its key is *greater* than the last key written in the **run** we insert it into queue 1 else we insert it into queue 2;

5.3 Index organization

An **index** is any method that takes as *input* a property of a set of*records*, typically the value of one or more *field*, and finds the *records* with that property quickly. Any **index organization** is based on the value of one or more predetermined *fields* of the *records* of the *relation*, which form the **search-key**. Any subset of the *fields* of a *relation* can be taken as the *search key* of the *index*. The notion of **search key** is different from the one of **key** of the *relation*, that is a minimal set of field that *uniquely identifying* the *records* of the *relation*. An index-based organization comprises:

- **Index file**: containing **data entries**, each containing a *value k* of the *search key* and used to locate the *data records* in the *data file* related to the *value k* of the *search key*, and **index entry**, used for the management of the *index file*;
- **Data file**: containing the *data records*, that are the *records* of *relation* R;

There are 7 different **properties** of an **index**:

1. Organization of the index;
2. Structure of data entries;
3. Clustering / Non-Clustering;
4. Primary / Secondary;
5. Dense / Sparse;
6. Simple Key / Composite Key;
7. Single Level / Multi Level;

There are three different **organization of the index**:

- **Sorted Index**: the *index* is a *sorted file*;
- **Tree-based**: the *index* is a *tree*;
- **Hash-based**: the *index* is a *function* from *search key values* to *record addresses*;

There are three different possible **structures of a data entry** (we will call later this three options *alternative* (...) or *technique* (...)):

1. Data entry is a **data record** (hashed file);
2. Data entry is a **pair** $(\mathbf{k}, \mathbf{r})$, where r is a reference of a *data record* with *search key* k;
3. Data entry is a $\mathbf{pair}(\mathbf{k}, \mathbf{r-list})$ where $r-list$ is a list of reference to *data records* with *search key* k;

An *index* can be **clustered or unclustered**:

- An index, for data file F, is **strongly clustering** when its *data entries* are stored in the *data file* F, according to the *search key*, otherwise is **non-clustering**;

- An index is **weakly clustering** if every value of the *search key* appears at least one time in *data file*;

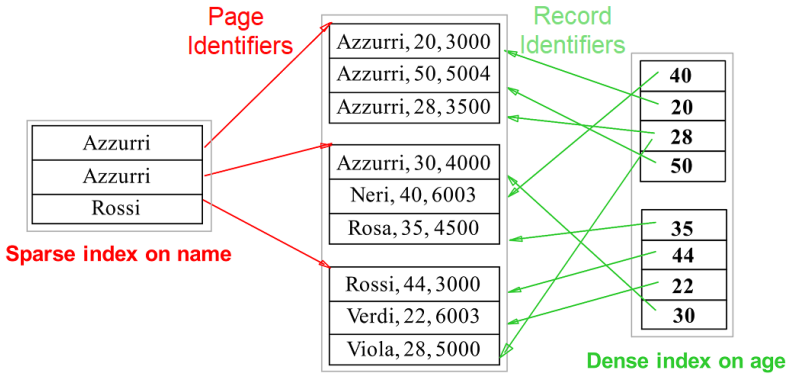
An index whose **data entries** are stored like (1) technique, is **clustered** by definition. In general there can be **at most one clustered index** per *data file*, cause the order of *data records* in the *data file* can be *coherent* at most one index *search key*.

We call **duplicate** two *data entries* with the *same values* of the **search key**. An *index* can be **primary or secondary**:

- A **primary key index** (or *primary index*) is an *index* on a *relation* R whose *search key* includes the **primary key** of R:
  - A **primary index** cannot contain any *duplicates*;
- If an *index* is not a *primary key index*, then is called **secondary index** (or *non-primary key index*);
  - A *secondary index* can contains **duplicates**, but if its *search key* contains a **key** (of course *non-primary*) is called **unique** and a **unique secondary index** doesn't contain *duplicates*.
  - A **secondary non-unique index** can be organized in two way for not contain *duplicates*:
    * If the *index* uses the **(3) technique**, so the *data entry* is a *pair* with a *list of reference*, and therefore every *relevant value* of the *search key* is stored only once in the index with a *list of rids*;
    * If the *index* uses the **(2) technique** and is **clustered**, and therefor for each *relevant value* k of the *search key* we have only one *data entry* in the *index*, pointing to the first *data record* R with value k for the *search key*, and since the index is *clustered*, the other *data records* with k for *search key* follow immediately R in the *data file*.

An index can be **sparse or dense**:

- An *index* is **dense** if every value of the *search key* that appears in the *data file* appears also in at least one *data entry* of the *index*. An index that uses *technique (1)* is always **dense**.
- A *dense index* is **strongly dense** if we have exactly one *data entry* for each *data record* of the *data file*, and the value of the *search key* in each *data entry* is the value held by referenced *data record*. In case of *techniques (2) and (3)* the *references* associated to *data entries* are **record identifiers**.
- An *index* is **sparse** if is not *dense*, and a **sparse index** is always *clustered*. Typically we have one *data entry* per *data page*, where the value of the *search key* of the *data entry* is the first data record in the corresponding data page, so if we use *techniques (2) and (3)* the *references* associated to *data entries* denote **page identifiers**.



A *search key* can be **single or composite** (and so an *index*):

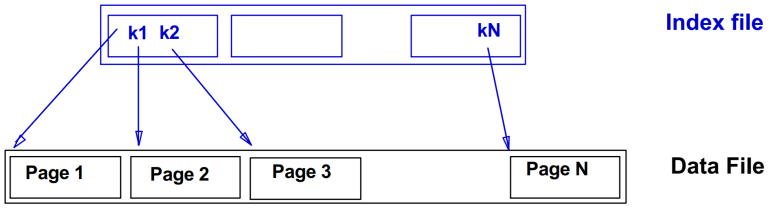
- A **search key** is called **simple** if it constituted by a *single field*, otherwise is called **composite**;
- If the **search key** is **composite**, then a *query* based on the *equality* predicate (**equality query**) is a *query* where the *value* of each *field* is **fixed**, while a *query* than fixes only some of the *fields* is a **range query**. A **composite index** supports a greater number of *queries* and a *single query* is able to extract more information, but is generally more subject to **update** than a *simple* one.

An *index* can be a **single level** or a **multi level**:

- A **single level index** is an *index* where we simply have a *single index structure* as well as the *indexed data file*.
- A **multi level index** is an *index* organization where an *index* is built on a structure that is in turn an *index file*.

5.3.1 Sorted Index

The idea is to create an *auxiliary sorted file* (so an **index**), which contains the *values* for the *search key* and *pointers* to *record* in the *data file*.



- **Clustering sorted index**: also called *primary sorted index* or **indexed sequential file**, in which the **data file** is a **sorted file**, with the *file sorted* on the same *search key* of the *sorted index*. Usually the *search key* coincided with the *primary key*.

- **Non-Clustering sorted index**: also called *secondary sorted index*, in which the **data file** can be *unsorted* or *sorted*, but in second case is *sorted* on attributes **different** from the *search key* of the *sorted index*.

We can have a **Clustering primary** (or *unique*) **sorted index** so on a *relation key* (*no duplicates*) or a **Clustering secondary** (or *non-unique*) **sorted index** so on *non-key* attributes.

The **Clustering primary sorted index** can be **dense or sparse**:

- If it is **dense** every value of the *search key* that appears in the *data file* appears also in at least one *data entry* of the *index* (if *strongly exactly one*);
  - **Clustering sorted index dense** requires more space and it is especially advantageous when it can *fit the main memory*, cause in this case we can find *any record* given its *search key* with only *one disk page access*;
- If it is **sparse** only some of the *data records* have a corresponding data *entry* with the same value of the *search key* in the *index file*, often there is one *data entry* per *page* in the *data file*.
  - **Clustering sorted index sparse** contains *one data entry* per *page* in the *data file,* so when we are searching a *record* with *key* k we need to search for the **largest key less or equal** to k and we follow the *associated pointer* to a *data page*, where we must search for the *record* with *key value* k;

All the **equality search** with *clustering primary sorted index* can be extended to deal with **equality range**, in we search for a *range* of **search key** values we can search the first value in the range, and then continue by moving forward, and this can be done either in the *sorted index* or in the *data file*. If instead we are in **weakly clustering** (all the data records with a fixed value for the search key appear on roughly as few pages as can hold them) then we hate to search in the *sorted index* considering one by one the values of the *range* and for each value access the *data pages*.

In the **Clustering secondary sorted index** we have **sorted data file** *with duplicates*. It can be **dense** or **sparse**:

- If the **clustering secondary sorted index** is **dense** we can have three alternative:
  - **With duplicates in the index and technique (2):** it is **strongly dense**, and for finding all the *records* with a *key* k we look for k in the *index file* and we follow the *pointer* to the first *data record* with *value* k and all the other k pointers follow immediately.
  - **With duplicates in the index and technique (3):** it is **strongly dense (?)**, and for finding all the *records* with a *key* k we look for k in the *index file* and we have an *associated list pointer* to the *data records* with value k.
  - **Without duplicates in the index and technique (2)**: it is **non-strongly dense**, in which the *pointer* associated to a *search value* is the first *data record* with that *search key*, and for find all the *records* with a key k and we follow the *pointer* to the first *data record* with *value* k and we find the other *data record* by *moving forward* in *data file*.

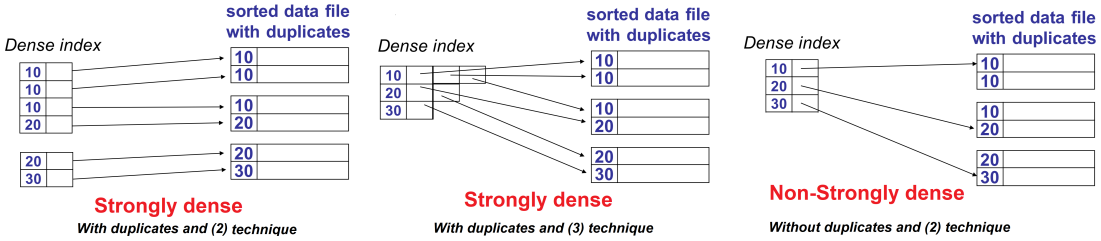


Figure 7: Clustering secondary sorted index dense

- If the **clustering secondary sorted index** is **sparse** we can have two alternative:
  - **With duplicates in the index**, usual rule for **sparse index**, so, one **data entry** per *page*, with *search key* equal to the *first key* in the *data page*, and in order to find the *records* with *search key* k, we find the last *entry* in the *index* that has a *key* values less or equal to k and we move *backward* in the *index* until we find the very first *data entry* of the *index* or we find an *entry* with a *key* value *strictly less* than k ;
  - **Without duplicates in the index**: *data entries* point to the *page* with the first instance of each value, and in order to find the *records* with *search key* k in the *data file* we look in the *index* for the first *data entry* whose *key* is either equal to k or less than k but with next *key* greater than k (look the entry 35 in the image) and we follow the *pointer* and if we find at least one *data record* with *key* k in this page we search forward until we find all *records* with *key* k;

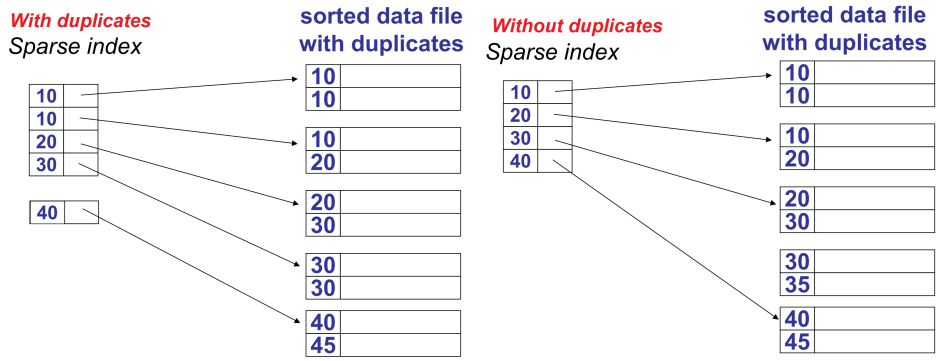


Figure 8: Clustering secondary sorted index sparse

We can have a **clustering sorted index with multiple level**, in fact by putting a *sorted index* on a *sorted index* we can make use of the *first level* (*points to the data file*) more efficient, the idea can be even iterated to a 3^{rd} level *sorted index*. The *first level* of the index can be *sparse* or *dense*, but *second* (points to *first level*) and the higher levels must be sparse, cause a *dense index* on a *index* would have essentially as many *data entries* as the *first level* so no advantages.

In the **Non-clustering sorted index** (or *secondary sorted index*), we have a data file **unsorted** or *sorted* but on *attributes* different from the *search key* of the *sorted index*. We can have a **Non-Clustering primary** (or *unique*) **sorted index** so on a *relation key* or a **Non-Clustering secondary** (or *non-unique*) **sorted index** so on *non-key* attributes.

Non-clustering primary sorted index can be **sparse** or **dense**, but since the *records* in the *data file* are not ordered in this case a *sparse index* doesn't make sense, so we use only *dense* index. A **non-clustering sorted index** supports well the operations of **equality search** on the *search key*, but the operation of the **range selection** is well supported only in case where we don't have to access to the *data file* (so in *index-only access*), in fact in every *non-clustering* with the *range selection operation* we need to access to a lot of *pages*, so the number of *page* accesses to the *data file* might grow considerably (*nei clustering basta andare avanti mentre qua dobbiamo accedere ad ogni pagina singolarmente*). The **multi-level index** can be applied also in *non-clustering index* and we have a *second level sparse* and a first level *dense*.

In **Non-Clustering secondary non-unique sorted index** instead we can have *duplicates* in *data files* and in our *index*. There are multiple possibilities:

- **Dense index with alternative 2**;
  - Where each **pointer** of one *index page* can go to many different *data pages*, instead of one or few consecutive *pages*. But we can have the problem of **excess overhead** (*disk space and search time*);
- **Dense index with alternative 3**;
  - This option is better than the first one, but we still have one problem: the **variable size records** in *index* (caused by the use of lists of *alternative 3*);
- **Dense index with alternative 2** but with the use of **buckets**;
  - The **buckets** organization, are used to link the *values* of the *search key* with all the *data records* holding such values, so with this organization we avoid the duplication of the *values* of the *search key* in the *index*. So we can see the *bucket* like a *middle structure* between *index file* and *data file*. *Buckets* are also useful for **query**, in fact some of them can be answered by operations on the sets of *pointers* in the *bucket*.

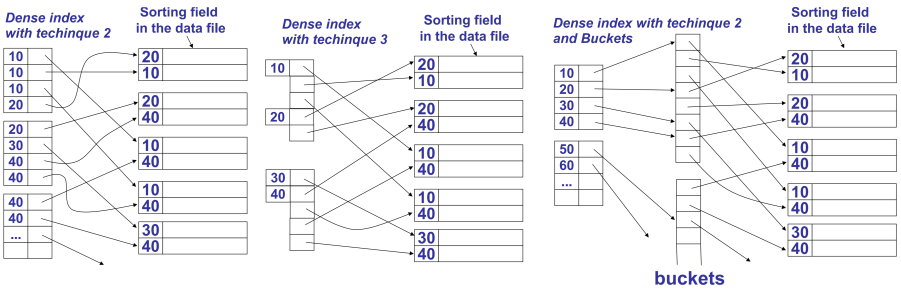


Figure 9: Non-Clustering secondary sorted index

5.3.2 Tree index

In a *sequential index* (like *sorted index*) the usage of *insertions* and *deletions* operation can lead to a reduction of *performance*. In the **tree index**, the *data entries* are organized according to a *tree structure*, based on the *value* of the **search key**.

- Every **node** of the *tree* coincides with a *page*;
- The *pages* with the *data entries* are the **leaves** of the *tree*;
- Any **search** starts from the *root* and ends on a *leaf* (so is important to minimize the *height* of the *tree*);
- The **links** between the *nodes* of the *tree* corresponds to *pointers* between *pages*.

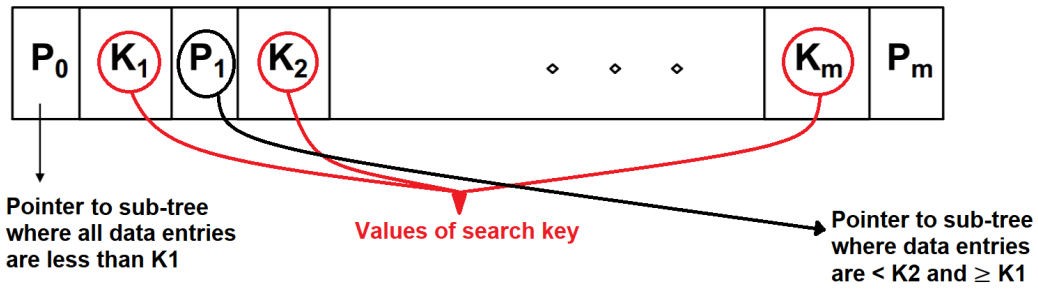


Figure 10: Typical structure of an intermediate node

There are two different types of tree index:

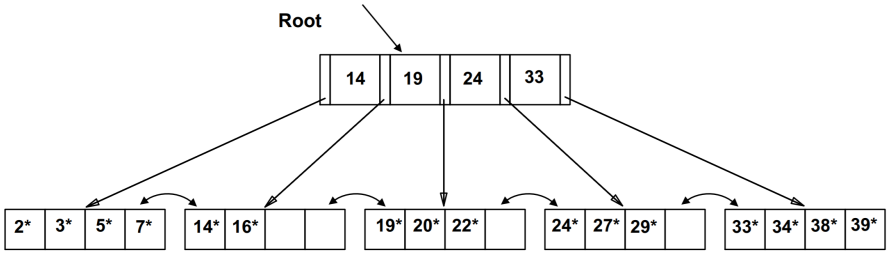
- **ISAM**: used when the *relation* (or the *index*) is **static** (so, no *insertion* and *deletion*);
- **B$^+$-Tree**: used in **dynamic** situation (with *insertion* and *deletion*);

The **ISAM** is a *balanced tree*, the name derives from **Indexed Sequential Access Method**. The *leaves* of the *tree*, contain the *data entries* and they can be scanned sequentially. This structure is *static* so there is no update on the *tree*. The **leaves** are allocated *sequentially*, and then the *intermediate nodes* are created. In order to **search**, we start from the *root*, and then we compare the *key* we are looking for with the *keys* in the *tree* until we arrive at a *leaf*. The **cost** is $log_F N$ where F is the **fan-out** (number or *children* of every *non-leaf nodes*, that is the same for all these *nodes* since the *tree* is *balanced*) and N is the *number of leaves*. *Insert* and *delete* operation are rare and involve only *leaves*.

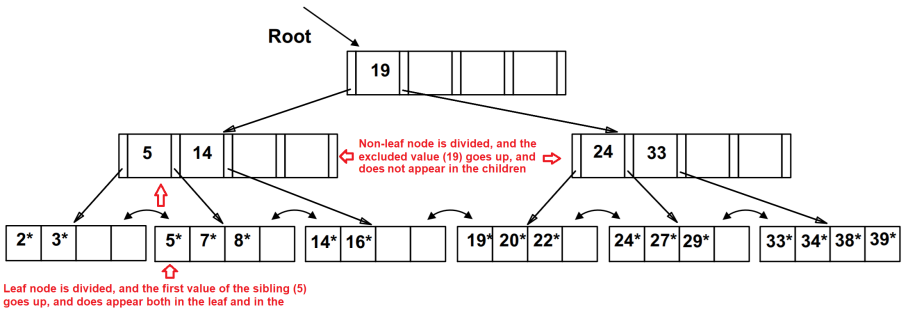
The **B$^+$-Tree** is a *balanced tree*, that overcome the problems of the **ISAM** with *insertion* and *deletion*. **B$^+$-Trees** are the ideal method for efficiently accessing data on **range operation** (especially whit a *clustering index*), and are also very effective for accessing data on *equality* (but they are not ideal). Every *page* has space for d *search key values* and $d+1$ *pointers*, and d is called **rank** of the *tree*. Every *node* n_i, contains m_i *search key values*, with $(d+1)/2 \leq m_i \leq d$, the only exception is the **root** that may have at least one *search key value*. The **leaves** are the *pages* with

the *data entries*, and they are linked through a *list* based on the order on the *search key*, so this list is very useful for *range queries*.

- **Searching**:
  - The number of **pages accesses** needed in a **search** for *equality operation* (on a *primary key*) is at most the *height* of the *tree*: $log_F N$ where F is the *fan-out* (average number of *children* per *node*) and N is the number of *leaves*, so the larger is F, less is the *cost*, and the *fan-out* depends on the size of the *block*;
- **Insertion**:
  - The **insertion operation** is made with an a *recursive algorithm*. We search the **leaf** where we need to insert the new *key*, if there is space we stop, else we **split** the *leaf* into two *equals parts* (two *nodes*). After splitting we need to insert the new *pointer* at the **higher level**, and this is made *recursively*.
  - If we are splitting a **leaf node**, we create a new *node* that will be the *sibling* of the *splitted node*, and after the division of the *keys*, the first value of the new *sibling* goes up to the *higher level* with the appropriate *pointer* (the new *key* appears both in the *sibling leaf* and in the *father*).
  - If we are splitting a **non-leaf node** after the *splitting* a value remains excluded by the *split*, and this values goes up to the *higher level* with the appropriate *pointer* (the new *key* appears only in the *father*).
  - Typically the tree increases in *breadth*, and not in *depth*, except when we need to insert into a full root.



Insertion of data record with search key 8, with rank d=4



- **Deletion**:
  - After the **deletion**, if the *node* does not have enough number of *keys* ($\geq \frac{d+1}{2}$), we need to apply one of this two techniques:
  - **Key Redistribution**: if one of the *adjacent sibling* of the *node* has more than the **minimum number** of *keys* then we can move one *pair* (*key and pointer*) to our *node*. With this *redistribution* possibly we have to change the keys at the **parent** of the *nodes* (*se il fratello di sinistra passa la sua chiave piu alta al nostro nodo, allora se questa diventa la key piu bassa del nodo devo modificare sopra la chiave del padre, stessa cosa se il fratello di destra passa la sua chiave piu bassa allora devo modificare la chiave del padre relativa al fratello del nodo*)
  - **Coalesce**: if there aren't *adjacent nodes* that can provide an extra *key* for our *node*, then we choose one of them and we **merge** them. After the *merging* we need to remove a pair *key-pointer* from the *parent* (cause the node doesn't exist anymore) and to adjust the *keys* at the *parent*, if the *parent* now does not have enough *keys* we recursively apply the *coalesce* until every *parent* is full enough. So, the *deletion algorithm* may result in lowering the *depth* of the *tree*, and sometimes *coalescence* is not implemented.

In the **clustered B$^+$-Tree** by assuming that we use technique (1) (so data entry is a data record, an hashed file) and with F as fan-out of the tree (for other variables look *5.2.4 cost model*):

- **Scan**: $1.5B(D + RC)$;
- **Selection on equality and on range**: $D\ log_F(1.5B) + C\ log_2 R$;
- **Insertion/Deletion**: $D\ log_F(1.5B) + C\ log_2 R + C + D$;

In the **unclustered B$^+$-Tree** by assuming that the *data file* is a *heap file* with technique (2) and a *dense index*:

- **Scan**: $0.15B(D + 6.7RC) + BR(D + C)$;
- **Selection on equality and on range**: $D\ log_F(0.15B) + C\ log_2(6.7R) + XD$, where X is the number of *records* that satisfy the *equality*;
- **Insertion**: $2D + C + D\ log_F(0.15B) + C\ log_2(6.7R) + D$;
- **Deletion**: $D\ log_F(0.15B) + C\ log_2(6.7R) + D + 2(C + D)$;

5.3.3 Hashed Index

There are two kind of **hashed index**:

- **Static Hashing**;
- **Dynamic Hashing**;

The **static hashing** coincides with **hashed file** in which we have a fixed number or *primary pages* N (*number of buckets*) that are *sequentially allocated*, *never de-allocated*, with *overflow pages* (if needed). The address of the *bucket* that contains the *record* with **search key** k is: $h(k) mod N$, where h is the **hash function** that should distribute the *values* uniformly into the *range* $0, ..., N - 1$. So,

the *buckets* contain the *data entries*, but since the number of **buckets** never changes with *insertion* and *deletion* we will have long *overflow chains* that are a problem for **efficiency**. If we assume that we use *technique (2)* and the data file is a *heap file*:

- **Scan**: $0.125B(D + 8RC) + BR(D + C)$;
- **Selection on equality**: $H + 2D + 4RC$, where H is the cost of identify the page thought the hash function;
- **Selection on range**: $B(D + RC)$;
- **Insertion**: $H + 4D + 2C$;
- **Deletion**: $H + 4D + 4RC$;

Dynamic hashing instead is divided in:

- **Extendible Hashing**;
- **Linear Hashing**;

The idea of the **extendible hashing** is using a *directory* of *pointer to buckets* and doubling only such *directory* in order to split only the *bucket* that is becoming full, and when the *directory* is doubled the *hash function* has to be adapted.

- **Global depth** of directory: number of bits needed to decide which is the bucket of a given entry;
- **Local depth** of a bucket: number of bits used to decide if an entry belongs to a bucket.

In order to find the *bucket* for a *key* k, we consider the last g bits of $h(k)$, g is called **global depth**, and each g in *directory* is linked to a *bucket*. The operation of **insert** can lead to two situations:

- If we need to **insert in a full bucket**, we doubling the *bucket* and we redistribute the *entries* of the old *bucket* according to the same *hash function* by using $c + 1$ bit where c is the **local depth**, so 1 bit more than the old *local depth*. After the *split* the *local depth* is incremented.
- If after a *bucket splitting*, the $localdepth > globaldepth$ the **directory must be doubled**. This means copying it and then setting up the *pointers* to the split image. When this happened we increment the *global depth*.

At most 2^{g-c} *directory elements* point to a *bucket* with *local depth* c, and if $g = c$ then exactly one *directory* points to the *bucket*, and *splitting* that would require **doubling the directory**. If the distribution of the **hash function** values is not *uniform* then the size of the *directory* can be very large, and in order to manage the *collisions* we may still require *overflow pages*. The **deletion** instead can make a *bucket empty*, in this case we *merge* it with its *split image*, and additionally if every entry in the *directory* points to the same *records* as its *split image* then we can halve the *directory*.

In the **linear hashing** the goal is to avoid the *directory*, so avoid one *access* while *searching*. The *primary pages* (N) are stored *sequentially*, all the *accesses* are organized in **rounds**, the variable $LEVEL$ (initially 0), tell us at which *round* we are at present.

- During **insertion/deletion** the *bucket* that have been allocated at the beginning of the *round* are split one by one, so that at the end of the *rounds* the number of the *buckets* is doubled;
- The variable $NEXT$ always points to the next *bucket* to split (so from 0 to $NEXT - 1$ have been already spitted). The method is flexible in choosing when the *split* will occur (when an *overflow page* is allocated, when a given condition occurs, ...);

Essentially we use a **family of hash functions**. and when we search for a *value* k we apply the *hash function* h_{LEVEL} to get the *bucket address* (T). If the *bucket* has not been split in this round ($T \geq NEXT$) then we look for the *data entry* $k*$ in *bucket* T otherwise we use the *hash function* $h_{LEVEL+1}$ to decide if we access the *bucket* T or its *split image*. The *family* is defined as: $h_i(v) = h(v)\ mod\ 2^i N$ where h is the **main hash function**. When the split occurs, data entries in bucket NEXT are redistributed according to h_{LEVEL} hash function. Differently from *extendible hashing* when a *split* occur during an **insertion** the *inserted data* is not necessarily stored in the *split bucket*. The *new bucket* (image of split bucket) is $b + N_{LEVEL}$, where N_{LEVEL} is the *number of buckets* in *current state*.

- If the **hash function** returns a number between $NEXT$ and N_{LEVEL} the bucket is not yet split;
- Instead if the **hash function** returns a number between 0 *and* $NEXT - 1$ the bucket is already split and we must use the new hash function (by looking at one more bit) to find the correct bucket where we will insert the new value;
- When all *buckets* are split, we go to a **different round**, so $LEVEL$ is incremented and $NEXT$ is reset to 0. This means that the *range* of the *hash function* is **doubled**, something similar to doubling the *directory* in *extendible hashing*;

So, in **Extendible Hashing** we split only the *most appropriate bucket*, so we have **less splitting**, instead in **Linear Hashing** we have an average *number of buckets* almost *empty low*, and we avoid the access to the *directory* so **one page access for every search**.

5.3.4 Comparing different file organizations

- **Heap File**:
  - Efficient in terms of space occupancy;
  - Efficient for scan and insert;
  - Inefficient for search and deletion;
- **Sorted File**:
  - Efficient in terms of space occupancy;
  - More efficient search respect to heap file;
  - Inefficient for insertion and deletion;

- **Clustered tree index**:
  - Limited overhead in space occupancy;
  - Efficient for insertion and deletion;
  - Efficient search;
  - Optimal support for search based on range;
- **Static hash index**:
  - Efficient search based on equality, insertion and deletion;
  - Optimal support for search based on equality;
  - Inefficient scan and search based on range;

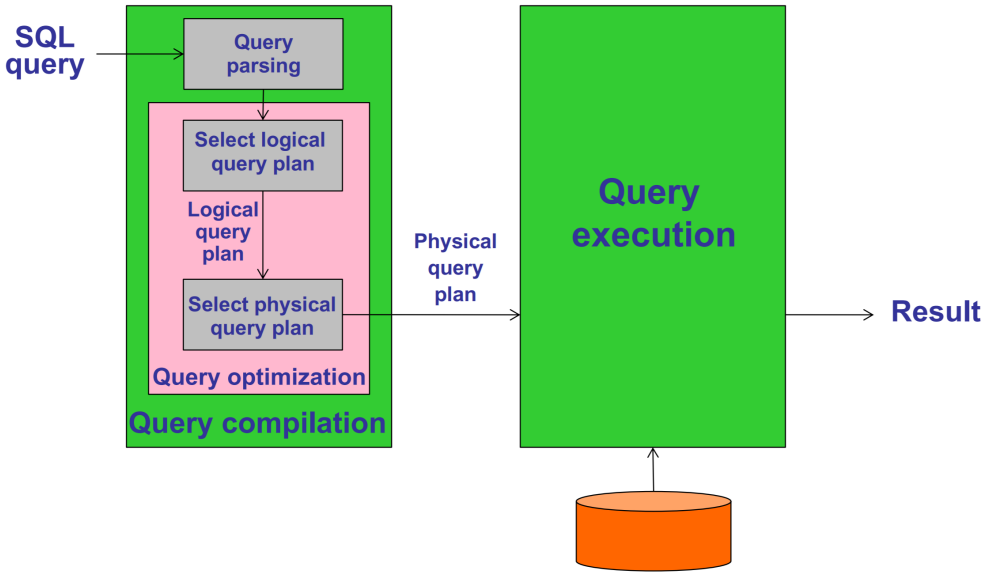
So no file organization is uniformly superior to others in every situation.

File Organization	*Scan*	*Search based on equality*	*Search based on range*	*Insertion*	*Deletion*
Heap file	BD	BD	BD	2D (we ignore time for space management)	Cost of search + D (we ignore time for space management)
Sorted File	BD	D $\log_2$B	D $\log_2$B + # of further pages	Cost of search + 2BD	Cost of search + 2BD
Sorted Index	D B/N	D $\log_2$ B/N	D $\log_2$ B/N + #of further pages	Cost of search + 2DB/N	Cost of search + 2DB/N
Clustered tree-based index (alternative 1)	1.5BD	D $\log_F$(1.5B)	D $\log_F$(1.5B) + #of further pages	Cost of search + D	Cost of search + D
Unclustered tree-based index (alternative 2)	BD (R +0.15)	D($\log_F$(0.15B) + #of further records)	D($\log_F$(0.15B) + #of further records)	D(3+ $\log_F$(0.15B))	Cost of search + 2D
Unclustered static hash-based index	BD(R+ 0.125)	D(1 + #of further records)	BD	4D	Cost of search + 2D

6 Evaluation of operators

6.1 Query processing by the SQL engine

The **SQL engine** is responsible of analyzing every component in *SQL*, any command, and executing it. We will focus on **queries**. We can see in the picture that the *query* goes as input to a **parser**, which will analyze the *query* and if the *query* is correct it builds an *internal representation* of the query. The **query compiler** produces at the end the **physical query plan**, that the **query executor** is going to interpret. Every choice about *optimization* is done in the *query compiler*, and only once we optimize the *query* we will pass everything to the *query executor*.



6.2 Evaluation of relational operators

We are going to considerate some **operators** from the *relational algebra* and some operators from *SQL* (not in algebra), and we are going to understand how an *SQL engine* evaluates a *query*. In principle an **operator** can be applied in two different modes:

- **Materialization-based**:
  - The *system* computes the *result* and stores it in a *temporary relation*;
- **Pipeline-based**;
  - Is an *implementation* of the operator that uses the notion of **iterator**, which is a mechanism for computing the result of an application of an *operator* in a *tuple-by-tuple*, on *demand fashion*. Every time an *iterator* is called, it is supposed to return a *tuple* in the result of the application of the operator. So you are not computing as a whole the result, you compute it in an incremental way.

An **iterator** is an *abstraction* of *objects* that are manipulated by three *functions*:

- **Open**: this *function* initializes all *data structures* used for getting the *tuples* that are the results of application of the *operator* and prepares the process of getting such tuples (so, this function is*: I want a new iterator on a relation*);
- **GetNext**: this *function* returns the *next tuple* of the *result*, and adjust the *data structures* so as to get subsequent *tuples*, and uses the variable $Found$ that is *true* if and only if a new *tuple* has been returned (so, this function is: *Give me the next tuple*);
- **Close**: this *functions* ends the *iteration* after all *tuples* have been obtained;

There are two kinds of **classification of algorithm for operators**. In the first type we will analyze in base of **how many passes** i need:

- **Sequential methods**:
  - **One-pass**:
    - These *algorithms* read the *data* only once from *secondary storage*, and usually they work when at least one of the *operands* fits in *main memory*. So, this *algorithm* will never examine twice each *page*.
  - **Nested-loop algorithms**:
    - These *algorithms* are based on a *loop* (analyze one *relation*) and inside there is another *loop* (analyze different one or the same *relation*). When specialized for the *join operator* they can be seen as one and half *algorithm* cause they work when only one of the two *operands* fits in available *main memory*.
  - **Two-pass**:
    - These *algorithms* read the *data* a first time from *secondary storage*, process them, and then write intermediate *data* on *secondary storage, read* them again for further processing. They work for *data* that is too large to fit in available *main memory*.
  - **Index-based**:
    - These *algorithms* base their strategy on the use of one or more *indexes*. There are also algorithms that are hybrid, like *two-pass* with *index*.
  - **Multi-pass**:
    - These *algorithms* uses three or more *passes* on the *data* and are natural generalization of the *two-pass* (for pass we mean how many times you *read* and *write page* on *memory*).
- **Parallel methods**
  - *Parallel algorithms* use many *processors* in a *shared-nothing architecture*.

In the second type of classification is based on which **strategy** i use:

- **Sorting-based**:
  - These *algorithms* requires to *sort* one or both *operands*. Some of *two-pass* and *multi-pass algorithms* use this technique.

- **Hash-based**:
  - These *algorithms* requires the use of one or more *hash functions*, even if the relation is not implemented with an hash index. Some of *two-pass* and *multi-pass algorithms* use this technique.
- **Index-based**:
  - These *algorithms* requires the use of one or more *indexes*, all *index-based algorithm* and also some of *two-pass* and *multi-pass algorithms* use this technique.

We will denote as $B(R)$ the **number of pages** used to **store** the *relation* R in the *secondary storage*. If we compare this with the number $QB(R)$, the **number of pages required** to *store* the *records* of R in the *secondary storage*, and if we suppose that the **average size of space per page** occupied by each *record* of R is $SR(R)$, the **size of a page** is SP and the **number of records** of R is $NR(R)$. Then:

$$QB(R) = NR(R) \times SR(R)/SP$$

So $B(R)$ is the *real number of pages* used to *store*, instead $QB(R)$ is the *required number of pages*.

- When the *relation* R occupies *pages* with only its *records*, and there is no *free space* in such *pages*, then $B(R)$ is the minimum number of *pages* holdings its tuples so $B(R) = QB(R)$. In this case we say that the *relation* is *clustered* (different from the notion of *clustered file* or *clustering index*);
- When the *pages* occupied by *relation* R have only *records* of R, and there is *free space* then $B(R) = QB(R) \times 100/(100 - C)$ where $C\%$ is the *free space* in each *page*;
- When the *relation* R is represented by a *clustered file*, and R is the *"child" relation*, so in every *page* we have an *average* of $C\%$ of space not occupied by R (so occupied by the *"father" relation* or free) then $B(R) = QB(R) \times 100/(100 - C)$;
- When the relation is represent by a *clustered file* and R is the *"father" relation* so in every page we have an average of $C\%$ of space occupied by R, then $B(R) = QB(R) \times 100/C$

In order to clear the term **cluster** we will recall the various interpretations:

- **Clustered file organization**: the records of two *relations* co-habits in the *same page*, one *record* of the *father relation* is followed by a set of related *records* of the *child relation*;
- **Clustered relation**: a *relation* is stored in a set of *pages* that exclusively or predominantly devoted to *store* the *records* of that *relation*;
- **Strongly clustering index**: An *index* on a *relation* R whose value are sorted coherently with the order used to store the *records* of the *relation* R;
- **Weakly clustering index**: An index is *weakly clustering* if all the *tuples* of the *indexed data file* with a fixed value for the *search key* used in the index appear on roughly as few *pages* as can hold them;

6.2.1 One-Pass Algorithm

- **Projection on relation R**:
  - We read the *pages* of R one at time into a *buffer frame (input frame)*, and we perform the operation on each *tuple*, and write the projected tuples into another *buffer frame (output frame)*. When the *output frame* is full we write the result in the *secondary storage*;
  - *Space in buffer*: one input frame and one output frame;
  - *Cost of I/O*: if the size of the relation R is B pages, the cost is B;
- **Selection on relation R** *(R unsorted and no index)*:
  - Equal to *projection*;
- **Selection on relation R** *(R sorted and no index)*:
  - In this case the *where* (*il comando where delle query de* sql) refers to the search key of which R is sorted. In this case we use *binary search* or *interpolation search*;
  - *Space in buffer*: one frame;
  - *Cost of I/O*: if the size of the *relation* R is B *pages*, the cost is $O(log_2 B)$ if we use *binary search*, if the *search key* is not a *key* of the relation we have to add as many *page accesses* as the number of *pages of interest*;
- **Duplicate elimination on relation R** *(R unsorted and no index)*:
  - We read each *page* of R one at a time and for each *tuple* we have to check if we already seen it, so we will use a *data structure* in order to keep a copy of every *tuple* seen, this *structure* can be an *hash table* or a *binary tree* (if we use a *list* the cost is $O(N)$ with N number of *tuples*). At the end we write into the result all the *pages* used for the *data structure*;
  - *Space in buffer*: if the buffers available are M, the number of pages needed to store are not greater than $M - 1$;
  - *Cost of I/O*: if the size of the *relation* R is B *pages*, the cost is B;
- **Duplicate elimination on relation R** *(R sorted and no index)*:
  - We read each *page* of R one at a time and we keep in the *buffer* the **group of tuples** with the value of the *sorting attributes* equal to the last value seen. For each *tuple* in the *page* we have to check if it is equal to one of the *tuples* kept in the *buffer*, if yes we ignore the *tuple* if not we add it to the *group*. When a *tuple* belongs to a new *group* we write the *pages* containing the *tuples* of the previous *group* into the result;
  - *Space in buffer*: if the buffers available are M, the number of pages needed to store are not greater than $M - 1$, in worst case the space used is M, if the sorting attributes includes all the attributes of the relation we only need one input frame and one additional frame;
  - *Cost of I/O*: if the size of the *relation* R is B *pages*, the cost is B;
- **Grouping on relation R** *(R unsorted and no index)*:

- We read each *page* of R one at a time and we create in the *buffer* one entry for each *group* (for each value of the *grouping attributes*). The *entry* is formed by the values of the *grouping attributes* and an accumulated value or values for each aggregation. The accumulated value depend on the *aggregation function* (min or max of the group, or the sum or the avg). When all the *tuples* of R have been read we produce the output by writing into the result all the *frames* containing the *tuples* for the *groups*;
- *Space in buffer*: if the buffers available are M, the number of frames needed to store one entry for each group is not greater than $M - 1$, in worst case the space used is M;
- *Cost of I/O*: if the size of the *relation* R is B *pages*, the cost is B;

- **Grouping on relation R** *(R sorted and no index)*:
  - We need to consider only the accumulated values for one *group* (current) in the *buffer*, we analyze the *tuple* of the *relation* by reading the *pages* one at time. When the *tuples* of one *group* is finished we write the *tuple* corresponding to the current *group* in the *output frame*. Whenever the *output frame* is full we write the result in *secondary storage*;
  - *Space in buffer*: one input frame and one output frame;
  - *Cost of I/O*: if the size of the *relation* R is B *pages*, the cost is B;

- **Bag Union**:
  - In order to compute the *bag union* of two relations R and S, we simply read every *page* of the two *relations* and write it into the *result*;
  - *Space in buffer*: one frame;
  - *Cost of I/O*: if the size of the *relation* R is B(R) *pages*, and the size of of relation S is B(S), the cost is $B(S) + B(R)$;

- Now we will analyze some *binary operators*, and we assume that $B(S) \leq B(R)$;

- **Set Union**:
  - We read the *pages* of S into M-2 *buffer frames*, we copy them into the *result* and build a *data structure* in the *buffer* with the *tuples* of S. We then read each *page* of R into the $(M-1)^{th}$ *buffer frame* one at time. For each *tuple* of R we use the *data structure* to decide if the *tuple* is in S, **if is not** then we copy it to the *output frame*, otherwise we skip it and delete it from the *buffer*. Whenever the *output frame* is full we write it into the result.

- **Set Intersection**:
  - Same as *union* but we check if the *tuple* is in S (in the *union if is not*), so if it is in S we copy it to the *output frame* and we delete it from the *buffer*, otherwise we skip it.

- **Set difference**:
  - If it is $R - S$ similar to *intersection*, if the tuple is in S we skip it and we delete it from the *buffer* otherwise we copy it to the *output frame;*

- If it is $S - R$ we read the *pages* of S into M-1 *buffer frames* and we build a *data structure* in the *buffer* with the *tuples* of S. We read each *page* of R into the M^{th} *frame* one at time and for each *tuple* of R, and we check if it is in S, if it is we delete it from the *buffer*. At the end we copy into the *result* the *pages* with all the *tuples* of S that are in the *buffer*.

- **Bag intersection**:
  - We read the *pages* of S into M-2 *frames* and we build a *data structure* in the *buffer* that associates to each *tuple* a **count** (how many times each tuple appears in s). Then we read each *page* of R into the M^{th} one at time, and for each *tuple* we use the *data structure* to decide if the *tuple* is in S. **If is not** we ignore the *tuple*, otherwise we decrements the *count* of the *tuple* in the *buffer* and we copy it into the *output frame*. When the *count* is 0 we *delete* it from the buffer. When the *output frame* is full we write it into the result.

- **Bag difference**:
  - Like *intersection*, if the *tuple* of R is in S, we decrements the *count*, and when the *count* is 0 we remove it from the *buffer*. At the end we copy into the *result* all the *tuples* in the *data structure* whose count is positive and the number of times we copy it equals that *count*.

- **Cartesian Product**:
  - We read the *pages* of S into M-2 *buffer frames*. We then read each *page* of R using the $(M-1)^{th}$, and for each *tuple* of R we concatenate it with each *tuple* of S, and write the *resulting tuples* in the M^{th} frame, the *output frame*, and when is full we write it into the *result*.

- **Natural Join**:
  - We assume that $R(X, Y)$ is joined with $S(Y, Z)$ where Y are all the *attributes* in common between R and S. In order to compute the *natural join*, we read the *pages* of S into M-2 *buffer frames*, and store their *tuples* into a *data structure* in the *buffer* such as an *hash table* or a *balanced binary tree* with the *attribute* Y as *search key*. We then read each *page* of R into the $M - 1^{th}$ frame, and for each *tuple* of R we find the tuples of S that agree with the tuple of R on all *attributes* in Y, by using the *data structure*. For each *matching tuple* of S, we form a tuple by joining it with the tuple of R, and we write the *resulting tuples* in the M^{th}, the *output frame*, and when this is full we write it into the *result*.

Operator	# buffer frames M required	Cost (Disk I/O)
selection, projection,	2	B
bag union	1	B
duplicate elimination, grouping	B + 1	B
union, intersection, difference, cartesian product, join	min(B(R),B(S)) + 1 or min(B(R),B(S)) + 2	B(R) + B(S)

Unsorted relations

Operator	# buffer frames M required	Cost (Disk I/O)
selection on sort key	2	log_2 B
selection not on sort key, projection	2	B
bag union	1	B
duplicate elimination	2 (if all attributes form the search key), B+1 (worst case)	B
grouping	2	B
union, intersection, difference	3	B(R) + B(S)
cartesian product, join	min(B(R),B(S)) + 2	B(R) + B(S)

Sorted relations

6.2.2 Nested-loop algorithms

Nested-loop are not used for *selection* and *projection*.

- **Duplicate elimination**:
  - We read one *page* P of R at time in an *input frame*, and we fill another input frame with all *subsequent page* of R (one at time). For each *tuple* of the page P (ignoring *duplicates*), we write it in the *output frame* if there is no other *subsequent pages* with the same *tuple*, otherwise we ignore all copies of that *tuple* in P. When the *output frame* is full we write it into the result.
  - *Space in buffer*: three frames;
  - *Cost of I/O*: this algorithm requires $B(B+1)/2$ pages accesses;
- **Block nested-loop**:
  - We can use more than 3 *buffer frames* in total, instead of looking one *page* a time in the *outer loop* we can load a *block* of M *pages* at time. For each of this *blocks* we perform the load of *subsequent pages* of the same *relation* in the *inner loop*.
  - *Cost of I/O*: this algorithm requires $B \times (3/2 + B/M) + M$ (*cost* when we have *one operand*);
- **Nested-loop join**:
  - This algorithm is also called **one and half pass algorithm** cause *one relation* is *read* once, and the other is read more than once. We will call p_R *and* p_S the number of *tuples* per *page* of R and S. In order to compute the *natural join* between $R(X,Y)$ and $S(Y,Z)$ there are several methods:
    - **Tuple-based nested-loop join**:
      - This is the simplest variant of *nested-loop join*, we scan the outer *relation* S, and for each *page* and for each *tuple*, we scan R, looking for the *tuples* joining with. The cost of this is very *high*: $B(S) + (p_S \times B(S) \times B(R))$;
    - **Page-based nested-loop join**:
      - We scan the outer *relation* S, and for each *page* P, we scan R looking for *tuples* joining with *those* in P. The cost of this algorithm is $B(S) + B(S) \times B(R)$;
    - **Block nested-loop join**:
      - The idea is to use M *buffer frames* plus two, one for *inner loop* of R, and the other as *output frame*;
      - For each *block* of M *pages* of S: read *pages* of S into into *buffer* and organize their *tuples* into *data structure* where *search key* is the common *attributes* of the two *relations*;
      - For each *page* p of R: read p into *buffer frame* and for each *tuple* of b, find in the *structure* the *tuples* of S in the *buffer* that *join* with it;
      - Then store in *output* the *tuples* that are the *join* of the *tuple* of R with each of these *tuples* of S;
      - The *cost* is approximated to: $B(S) + \frac{B(R) \times B(S)}{M}$ (*cost* when we have *two operands*);

6.2.3 Two-pass algorithms

Two-pass algorithms are a modification of *one-pass algorithm* designed to cope with the situation where the *relations* are larger than what the *one-pass algorithms* can handle. The *data* are read from *operands* into *main memory*, processed in some way and written out to *disk* again, and then somehow *re-read* from *disk* to complete the operation. There are two kind of *two-pass algorithms*:

- **Based on sorting**;
- **Based on hashing**;

We will analyze now the **algorithms based on sorting**. If we have a *large relation* R (or both) such that B(R) or $B(R) + B(S)$ is greater than M *buffer frames available*, then we can use the algorithm constituted by this two passes:

1. In the *first pass* we repeatedly the following:
   - Read M *pages* of the *relation* into *buffer*;
   - Sort these *pages* in *main memory* and forming a *sub-list*;
   - Write the *sorted sub-list* in M *pages* of *secondary storage*;
2. The *second pass* processes the *sorted sub-list* in some way to execute the *desired operation* and computing the *result* on one *output buffer frame*;

- **Duplicate elimination using sorting**:
  - After building *sorted sub-lists* for R, we use the *available buffer frame* to hold one *page* for each *sub-list*, then we copy into output the *minimum tuple* among the ones under considerations, ignoring all *tuples* identical to it.
  - *Space in buffer*: the algorithm requires $\sqrt{B(R)} + 1$ buffer frames;
  - *Cost of I/O*: this algorithm requires $3B(R)$, one B(R) for create *sub-lists*, one to write each of the *sorted sub-lists*, and one to read each page from the *sorted sub-lists*;
- **Grouping and aggregation using sorting**:
  - During the *pass 1* we produce the *sub-lists* of R, sorted by attributes of *grouping attributes*. In the *pass 2* we take the last value of *search key* v in current *available tuples* in *buffer*. We group all *tuples* with the same *search key* and we accumulate the *needed aggregates* (*sum and count for avg*), when there are no more *tuples* with *sort key* v, we put the new tuple in *output frame*. During the process if a *buffer frame* become empty we read another *page* from the same *sub-lists*.
  - *Space in buffer*: the algorithm requires $\sqrt{B(R)} + 1$ buffer frames;
  - *Cost of I/O*: this algorithm requires $3B(R)$;
- **Set Union using sorting**:
  - In the *pass 1* we produce all the *sorted sub-lists* for both *relations* R and S. In the *pass 2* we use one *buffer frame* for each *sub-lists* of R and S, and repeatedly find the first remaining *tuple* among all the *frames*. This *tuple* is copied to the *output frame* and all the copies of the *tuple* are removed from R and S. During the process if a buffer becomes empty, then we reload it with the next page from its sub-list.

- *Space in buffer*: the algorithm requires $\sqrt{B(R)+B(S)}+3$ buffer frames;
- *Cost of I/O*: this algorithm requires $3(B(R)+B(S))$;

- **Set Intersection using sorting**:
  - Like the *union*, but we copy the *tuple* in the *output frame* only if the *tuple* appears in both the *relations*;
- **Bag intersection using sorting**:
  - Like the *union*, but we copy the *tuple* in the *output frame* a number of times equal to the minimal number between the appears of the *tuple* in S and in R.
- **Set Difference using sorting**:
  - Like the *union*, but we copy the *tuple* in the *output frame* only if the *tuple* appears in R but not in S;
- **Bag difference using sorting**:
  - In the *pass 1* we produce all the *sorted sub-lists* for both *relations* R and S. In the *pass 2* we use one *buffer frame* for each *sub-lists* of R and S, and repeatedly find the first remaining *tuple* among all the *frames*. This *tuple* is copied to the *output frame* a number of times which is the *difference* between the number of *appears* in R and number of *appears* in S and all the copies of the *tuple* are removed from R and S. During the process if a *buffer* becomes empty, then we reload it with the next *page* from its *sub-list*.
  - *Space in buffer*: the algorithm requires $\sqrt{B(R)+B(S)}+3$ buffer frames;
  - *Cost of I/O*: this algorithm requires $3(B(R)+B(S))$;
- **Simple sort-based join algorithm**:
  - This *algorithm*, differs from the general patter since in the *pass 1* we sort using the Y of the two *relations* through *two-pass multi-way merge sort* using M *buffer frames* (so $B(R)$ $and B(S)$ are $\leq M^2$). In the *pass 2* we compute the *join*, and in the *first formulation* we will assume that for no values y of Y the *tuples* of R and S with that value occupy more than M-1 *frames*. In this case we repeatedly do the following:
    - Find the least value y of the *join attributes* Y that is currently at the front of the frames for R and S;
    - If y appears only in one *relation*, we remove the *tuples* with y as *Y-value*, else we identify all the *tuples* of both R and S, with y as *Y-value*, using the other M-3 *buffer frames*, and put in the *output frame* all the *tuples* corresponding to the *join* of all the *tuples* of R with all the *tuples* of S with y as *Y-value*;

    If instead for some value y the number of *tuples* of R and S occupy more than M-1 *frames*:

    - If the tuples from one relation (for example R), that have y as *Y-value* fit in M-2 frames, we load these *pages* of R into the *frames*, and read the *pages* of S that hold *tuples* with y, one at time, into one *frame*.

        * If neither relations has sufficient few tuples with y as *Y-value*, that they will fit in M-1 frames, then we will use these frames to perform the nested-loop join of the tuples with y as *Y-value*, from both relations. In either case, it may be necessary to read pages from one relation. and ignore them for a later re-analysis.
    - *Space in the buffer:* $\sqrt{max(B(R), B(S))}$;
    - *Cost of I/O*: $5(B(R) + B(S))$;

- **Sort-merge join algorithm:**
    - If we know in advance that the case of too many *tuples* with a *common attribute* do not occur we can use the following algorithm. After the creating of the *sub-lists* with *sorting key* Y, for both R and S we will bring the first *page* of each *sub-list* into the *buffer*. Repeatedly find the last y as *Y-value*, among the available *tuples* of all the *sub-lists*. Identify all the *tuples* of both *relations* that have y as *Y-value* using some of the M *available frames*, if there are fewer than M *sub-lists*. We put in the *output frame* the *join* of all *tuples* from R with all *tuples* from S, sharing the common *Y-value*.
- *Space in the buffer:* $\sqrt{B(R) + B(S)} + 3$;
- *Cost of I/O*: $3(B(R) + B(S))$;

Now we will analyze the **two-pass algorithms based on hashing**. The idea is: if the *data* is to big to be processed in *one-pass* we *hash* all the *tuples* of the *operands*. For all the *common operations* there is a way to select the *hash key* so that all the *tuples* that need to be considered when we perform the *operation* has the same *hash value* computed by the *hash function*, and therefore are in the same *bucket*. We then perform the *operation* by working on on *bucket* at time. If there are M *free buffer*, we can pick M or M-1 as the number of *buckets*, so this *algorithm* begins by *partitioning* the *relation* into M-1 *buckets* in secondary storage of about equal size.

- **Duplicate elimination using hashing**:
    - After *partitioning* the *relation*, we can carry out the *second pass*. Since two copies of the same *tuple* will *hash* to the same *bucket*, we will examine one *bucket* at time in isolation and we produce R_i, that is the *portion* of R that *hashes* to the i^{th} bucket. Then we can take as the answer the *union* of such R_i. When we analyze the *single bucket* we can use the *one-pass algorithm* for *duplicate eliminate* (if R_i fit in the *buffer*);
    - *Space in the buffer:* $\sqrt{B(R)} + 1$, and the algorithm works if $B(R) \leq (M-1)^2$;
    - *Cost of I/O*: $3B(R)$;
- **Grouping and aggregation using hashing**:
    - We *partition* the *relation* by using an *hash function* that depends only on the *grouping attributes*. After we can carry out the *second pass*, where we use the *one-pass algorithm* to process each *bucket* in turn. When we analyze a *bucket*, we form the *tuple* of the *result* derived from each *group* stored in that *bucket*, knowing that all such *tuples* are stored in such *bucket*;
    - *Space in the buffer:* $\sqrt{B(R)} + 1$, and the algorithm works if $B(R) \leq (M-1)(M-1) \times A$ where A is the average number of tuples per group;

- *Cost of I/O*: $3B(R)$;

- **Union intersection and difference using hashing**:
  - We *partition* each *operand* R and S *relation* by using one *hash function*. Both the *relations* will have M-1 *bucket* after the *first pass*. We then load in the *buffer* the various pairs of *buckets* (R_i and S_i will be considered together, since they correspond to the same value of the *hash function*), and we compute the result for this *bucket* by the *one-pass algorithm* depending on the *operator* we have to execute;
  - *Space in the buffer:* $\sqrt{min(B(R), B(S))} + 2$;
  - *Cost of I/O*: $3(B(R) + B(S))$;

- **Join using hashing**:
  - We *partition* each *operand* R and S *relation* by using one *hash function*. Both the *relations* will have M-1 *bucket* after the *first pass*. We then load in the *buffer* the various pairs of *buckets* (R_i and S_i will be considered together, since they correspond to the same value of the *hash function*), and we compute the join for this *bucket* by the *one-pass join algorithm*;
  - *Space in the buffer:* $\sqrt{min(B(R), B(S))} + 2$;
  - *Cost of I/O*: $3(B(R) + B(S))$;

- **Hybrid hash-join algorithm**:
  - If $B(S) << M-1$, we can *partition* S into k *buckets* where $k < M-1$. In the *first phase* we *hash* S and R. When we *hash* S, we keep in *memory* m *buckets* (where $m < k$), and the other *buckets* $(k-m)$ in the *secondary storage*. We then read R one *page* at time in an *input frame* and we use $k-m$ *buckets*. If a *tuple* of R *hashes* to one of the first m *buckets* we compute immediately the *join* with the *tuples* of the corresponding *bucket* in *main memory* using one *output frame*. If instead a *tuple* of R *hashes* to one of the corresponding $k-m$ *buckets* we sent the *tuple* in the right *buffer frame* R_{m+i}. In the second phase we compute the *join* of the $k-m$ pairs of *buckets* by using the *one-pass technique*.
  - *Space in the buffer:* $>> \sqrt{min(B(R), B(S))} + 2$;
  - *Cost of I/O*: $(3 - 2M/B(S)) \times (B(R) + B(S))$;

So, the **hash-based algorithms** for *binary operations* have a *buffer* requirement that depends only on the smaller of the two arguments rather than on the sum of the arguments sizes as for **sort-based algorithms**. *Sort-based algorithms* allow us to produce a result in **sorted order**, and we can take advantage of that *sorting*, but since we have to execute a *sorting algorithm* they are more *costly*. *Hash-based* depend on the *buckets* being of equal size, but since in practice there is generally a small variation, it's not possible to use *buckets* that on average occupy M *pages*, so we must limit them to a smaller figure.

6.2.4 Index-based algorithms

Index-based algorithms are algorithms that make use of *indexes*. There are *index-based* that use *one-pass* and other *nested-loop schema*. We need to introduce the notion of **conformance** of an *index* to a *condition*. An index *conforms* to a condition C when it can be used effectively to evaluate C. A simple index is *conform* to *attr* **op** *value* if:

- If it is a *tree index*, the search key is *attr*, and the *op* is: $<, \leq, =, <>, \geq, >$;
- If it is a *hash index*, the search key is *attr*, and the *op* is: $=$;

We say that a **prefix** of a *search key* is a initial not-empty segment of *attributes* for a composite *search key*, example for search key $< a, b, c >$, $< a >$ *and* $< a, b >$ are prefix, while $< a, c >$ *and* $< b, c >$ are not prefix. An index is said to be *conform* to the conjunction $(attopvalue) and (attopvalue) and ...$ if:

- The index is *tree-based*, and there exist a *prefix* P of the search key such that, for each attribute att in P, there is a term in the conjunction (these terms are called **primary terms**);
- The index is a *hash index*, and for each attribute of the search key there is a term $(att = value)$ in the conjunction;

Indexed-based algorithms for selection are special *one-pass algorithms*, in fact they are even more *efficient* than classical *one-pass algorithm*. If we consider the query:

- Select *
- From R
- Where <att op value>

We will distinguish between various cases depending on the method used to represent R:

1. The *attribute* is a *key of the relation*, op is $=$, and we have an *index* on such attribute:
   - If the *index* is a **hash index**, then the index *conforms* to the condition and we can use the *index*. The cost of the *selection* is 1 (or 2 if we count one access for the bucket and one for overflow page);
   - If the *index* is a **tree-based index**, then the index *conforms* to the condition and we can use the *index*. We refer to the cost of *equality search* for tree-based, if we don't know the *fan-out*, we can assume that the cost is 3-4 page accesses.
2. The *attribute* is *not a key of the relation*. op is $=$, and we have a *weakly clustering hash index* on such attribute:
   - Since it is *weakly*, the *tuples* with a given value of the *attribute* will be found in few *pages* (one or two). We denote $T(R)$ the number of *tuples* of R, and $V(R, A)$ the number of distinct tuples in the *projection* of R, on attribute A (so V(R,A) = T(R) if A is a key of R);
   - If the condition $=$ is on *attribute* A, and there is an *hash index* on A, then the index *conforms* to the condition. The *cost* of the *selection operation* is $1 + B(R)/V(R, A)$ or $2 + B(R)/V(R, A)$, cause $B(R)/V(R, A)$ is the number of *pages* required to hold the tuples with a certain value for A;

3. The *attribute* is *not a key of the relation*. op is $=$, and we have a *clustering tree index* on such attribute:
   - Since it is *clustered*, the *data file* is ordered with the same criterion of the *index*, and all the *records* can be found in few *pages*;
   - The index *conforms* to the condition, and we can use the *index*. The average *cost* of the *selection operation* is: $log_F 1.5B + B(R)/V(R, A)$. If we don't know the *fan out*, we can assume the *cost* is $4 + B(R)/V(R, A)$;
4. The *attribute* is *not a key of the relation*, op is $=$, and we have a *non-weakly clustering index* on such attribute:
   - Since the *index* is *non-weakly* then each entry can point to a *qualifying record* on a different *page*, and the *cost* could be one page per *qualifying record*.
   - We can do better by first sorting the *rid* in the *index data entries* by their *page-id component*, so that when we bring in memory a *page* of R, all *qualifying records* in this page are retrieved sequentially. The cost of *retrieving* the qualifying records is now the number of *pages* of R, that contain qualifying records.
5. The *condition* involves one *attribute* and the *op* is: $<$ or $>$ or a *range equality*, and we have a *clustering tree index* on such attribute:
   - The index *conforms* to the condition, and thus we can use the *index*.
6. The condition is a *complex condition*:
   - A *selection* with a *complex condition* C can sometimes be implemented by splitting the *condition* into *atomic condition*, and using a *index-based selections* for *atomic conditions* is possible.

- **Index based projection** also called **index-only scan**:
  - If we compute the *projection* on $A_1, ..., A_n$ on *relation* R, and we have a *dense index* (in particular, a B^+-Tree), whose *search key* includes all *attributes* $A_1, ..., A_n$ then we can use the *index*, in particular the so-called **index-only scan**;
  - An **index-only scan** for a *tree-based index* is a scan of the *leaves* of the *tree*, that is much efficient than the scan of the *data file* (usually bigger than the *leaves*). If $A_1, ..., A_n$ form a *prefix* of the *search key*, then during the *index-only scan* we can efficiently (through *one-pass algorithms for sorted relation*): eliminate those attributes that are not among the *target attributes* in the *projection*, and eliminate *duplicates* if needed.
- **Index-based join**:
  - There are two different kind of *index-based join*, the first is called **index-nested loop algorithm**, in which we want to compute the **natural join** between $R(X, Y)$ *and* $S(Y, Z)$ and **S has an index on Y**. We consider one *page* of R at time, and in each *page*, for each *tuple* t we uses the *index* to find all *tuples* of S having $t[Y]$ as *Y-value*, and we join each of these *tuples* with t. The cost is B(R) plus T(R) times the cost of accessing S, through the index. So, for each access to S (for each tuple t of R), we have the *cost* of accessing the index, plus the cost of reading an *average* of $T(S)/V(S, Y)$ *tuples* with the value for Y given by t.

- If the index is *non clustering*, the number of *page accesses* to S is $T(R) \times T(S)/V(S,Y)$. Instead if the index is *clustering* it is: $T(R) \times B(S)/V(S,Y)$;
- In the second case of *index-based join* we consider the **natural join** between $R(X,Y)$ and $S(Y,Z)$ under the assumption that **one (or both) of the indexes is sorted**. In this case we can perform the *ordinary sort-join algorithms* but with advantage of avoiding the sorting of one of the *relation*, the one which have the *sorted index*, so we access the *data files* only when it is necessary. We can perform only the final step of the *simple sort-join*, and this is called **zig-zag join**, whose names comes from the fact that we jump back and forth between the *indexes* (**not the relations**) finding the *Y-values* in common;
- In case of *sorted clustering index* the retrieval of all *tuples* of the corresponding *relation* with a given *search key* will result in a number of *page accesses* proportional to the number of *pages* of the *relation* needed to store such *tuples*. In case of *non-clustering sorted index*, the retrieval of all *tuples* of the corresponding *relation* with a given *search key* will result in a number of *page accesses* proportional to the number of such *tuples*;

6.2.5 Multi-pass algorithms

Two-pass algorithms can be generalized to algorithms that using as many *passes* as necessary can process *relations* of arbitrary size, these are called **Multi-pass algorithm**. We can consider two generalization like the *two-pass algorithms*:

- **Sort-based**;
- **Hash-based**;

In order to continue we need to remember the **recursive multi-pass sort-merge algorithm**:

- **Base step**:

  If R fits in the M *frames* available in the *buffer* (so $B(R) \leq M$) then sort R in the *buffer* and write the *sorted relation* into the *secondary storage*;

- **Inductive step**:

  If R doesn't *fit*, *partition* the *pages* of R into M-1 groups: $R_1, ..., R_{M-1}$ and *recursively sort* these groups, and then *merge* using M-1 *buffer frame* as input and one *buffer frame* for the output, and write the *sorted relation* to *secondary storage*.

To describe the structure of a **multi-pass sort-based algorithm** we distinguish between **unary and binary operations**. Let's analyze first the case of **unary operations**, by assuming that there are M *available buffer frames*.

- In the *first phase*, we *partition* R into M-1 parts, and we *sort* each of them by using the *multi-pass merge-sort algorithm*;
- In the *second phase*, we load one *page* at time of the *sorted sub-lists* of R,and we perform the *operation* on the *tuples* at the front of the *sub-lists* using a *buffer frame* for the output. For *duplicate elimination* we output one copy for each distinct *tuples*, for *aggregation* we combine the *tuples* with a given value of the *grouping attributes* depending on the *aggregation function*;

In case of **binary operations** instead:

- In the *first phase*, the partition into M-1 parts is done on the two relations R and S, in such a way that R is divided into M_R parts and S in M_S parts such that $M_R + M_S \leq M - 1$. Each part is sorted using the *multi-pass merge-sort algorithm* so as to produce the *sorted sub-lists*;
- In the *second phase*, we read each pair of corresponding *sorted sub-lists* and we operate in the *buffer* using one *output buffer frame*, and we do the computation in base on the *operation*;

In the *binary operation*, in the *first phase*, we can divide the M-1 *buffer frames* between relations R and S as we wish. One method is to divide the *buffer frames* in proportion to the *pages* of the two *relation*: R gets $M_R = (M-1) \times \frac{B(R)}{B(R)+B(S)}$ and S gets the rest. If K is the number of *passes*:

Operator	# buffer frames M required	Page accesses (Disk I/O)
duplicate elimination, grouping	$B^{1/K} + 1$	$(2K-1)B$
union, intersection, difference, join	$(B(R)+B(S))^{1/K} + 1$	$(2K-1)(B(R)+B(S))$

We will now analyze the **Multi-pass hash-based algorithms**.

- **Base step**:

  For an *unary operation*, if the *relation* fits in M *buffer frames*, read it and perform the *operation*. If is a *binary operation*, if either *relation* fits in M-1 *buffer frames*, perform the *operation* by reading this *relation* into *buffer* and then read the second *relation* one at time at buffer M^{th};

- **Inductive step**:

  If no *relation* fits in the *buffer*, we *hash* each *relation* into M-1 buckets, and we *recursively* perform the *operation* on each *bucket* (*unary*) or corresponding *pair of buckets* (*binary*), and we accumulate the *output* from each *bucket of pair*.

Obviously, at every *pass* we use a **different hash function**. The cost of *page accesses* is the same as the *multi-pass sort-based algorithm*:

Operator	# buffer frames M required	Page accesses (Disk I/O)
duplicate elimination, grouping	$B^{1/K} + 1$	$(2K-1)B$
union, intersection, difference, join	$min(B(R),B(S))^{1/K} + 1$	$(2K-1)(B(R)+B(S))$

6.2.6 Parallel algorithms

We call **shared-nothing architecture**, when all *processors* have their own memory and their own *secondary storage*, and do not share resources. The communication in shared-nothing is via **communication network**. We assume that *communication cost* is much less than read/write a *page* from/in *secondary storage*. *Parallel evaluation* in relational algebra is based on the fact that a *relation* R is split into P chunks $R_0, ..., R_{P-1}$ stored at P nodes. There are three strategies for *partitioning* a *relation* R:

- **Round-Robin**: *tuple* t_i goes to *chunk* $(imodP)$;
- **Range-based partitioning on attribute A**: *tuple* t goes to *chunk* i if $v_{i-1} < t.A < v_i$;
- **Hash-based partitioning**: uses a *hash function* for distributing the *tuples* of a *relation*, and this function works on all values of the *tuple*;

We will now analyze all the **operators**:

- **Parallel algorithm for selection**:
  - We are interested in: $\sigma_{A=V}(R)$ or $\sigma_{v1<A<v2}(R)$, if the tuples of relation R are distributed evenly among the P processor's disks the time for complete the operation is $B(R)/P$;
  - In **Round-Robin**: all *servers* do the work;
  - In **Range-based**: only one *server* do the work;
  - In **Hash-based**: if the *hash function* works on A, then one *server* for $\sigma_{A=V}(R)$ else all server for $\sigma_{v1<A<v2}(R)$;
  - The *selection* could radically change the *distribution* of *tuples* in *result* compared to *distribution* of R;
- **Parallel algorithm for projection**:
  - Equal to *selection*, but the *projection* cannot change the *distribution* of tuples;
- **Parallel algorithm for duplicate elimination**:
  - If we use an *hash-based partitioning*, then all the *duplicates tuples* are placed in the *same processor*, so we can apply a standard algorithm to each *processor*, and the *elapsed time* is: $B(R)/P$;
- **Parallel algorithm for grouping**:
  - If we use an *hash-based partitioning*, then all the *tuples* belonging to the *same group of* R are placed in the same *processor*, so we can apply any *algorithm* for *grouping* locally, and the *elapsed time* is: $B(R)/P$;
- **Parallel algorithm for binary operators**:
  - Similar to *duplicate elimination*, if the *same hash function* has been used then we can take the *union, intersection, or difference* of R and S by working in *parallel* on the portions of R and S at each *processor*. If instead we use a *different hash function*, we should first *hash the tuples* of R and S at each *processor* in *parallel* using a *single new hash function* and proceed according to the specific *operator*;

- **Parallel algorithm for set union**:
  - If we have to distribute the *tuples*, then we use the *hash function*, when the *frame* of the *bucket* i at *processor* j is full, we ship the content of the *frame* to processor i. *Processor* i receives all the *tuples* of R and S belonging to *bucket* i. After, each *processor* performs the *union* of the *tuples* of R and S belonging to its *bucket*. The *result* will be distributed over all the *processors*. If the *hash function* truly randomize the placement of *tuples* in *buckets* we should have the same number of *tuples* of the *result* to be at each *processor*;
- **Parallel algorithm for set intersection, set difference, bag intersection, bag difference**:
  - Same as the *union*;
- **Parallel algorithm for join and grouping**:
  - In order to compute the *natural join* of two *relations*, we do the same as the *union*, but we use an *hash function* that depends only on the *attributes* that we will join ($R(X,Y)$ $S(Y,Z)$ *so* Y), so in this way all the *joining tuples* are sent to the *same processor (bucket)*. The *join* is made through one of the *algorithm* already discussed;

7 Query processing

7.1 Query Parsing

Query parsing is done in two phases:

- The **parse phase**:
  - The *SQL query* is analyzed and represented as a *parse tree*;
  - The *Parse tree* is pre-processed with the goal of performing the following actions:
    - All *views* are substituted by their definition;
    - Every element of the *query* should be valid elements of the schema;
    - Resolving all ambiguities of attributes, if possible;
    - Type checking;
  - If the *parse tree* is valid we can go on the *convert phase*, otherwise an error is issued;
- The **convert phase**:
  - In this phase the *SQL parse tree* is converted into an **extended rational algebra expression tree** by using:
    - **Union, intersection and difference**: we use subscript S if works on *sets*, B if works on *bags*;
    - **Selection and projection**: (indicated as σ, π) we assume that renaming is done with *projection* of *bags*;
    - **Cartesian product and join**: (*natural join*: $\bowtie$, *theta-join*: $\bowtie_C$ made on *bags* (even all variants of join);

* **Duplicate elimination**: (indicated as δ) we turn a *bag* into a *set*;
* **Grouping**: corresponding to *GROUP-BY* and the *aggregation* indicated as γ;
* **Sorting**: corresponding to *ORDER-BY*;

An **extended rational algebra expression tree** is a *tree* where *non-leaf nodes* are labeled by operators of the extended relational algebra and leaves are database relations.

7.2 Selecting logical query plan

7.2.1 Rules

Relational algebra rules are **equivalence-preserving** transformation rules.

- $R \bowtie S = S \bowtie R$;
- $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$;
- $R \times S = S \times R$;
- $(R \times S) \times T = R \times (S \times T)$;
- $R \cup S = S \cup R$;
- $(R \cup S) \cup T = R \cup (S \cup T)$;
- $R \cap S = S \cap R$;
- $(R \cap S) \cap T = R \cap (S \cap T)$;
- $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$;

The last law is valid only for sets not for bags.
Selection:

- $\sigma_{p1}[\sigma_{p2}(R)] = \sigma_{p2}[\sigma_{p1}(R)]$;
- $\sigma_{p1 \wedge p2} = \sigma_{p1}[\sigma_{p2}(R)] = \sigma_{p2}[\sigma_{p1}(R)]$;
- $\sigma_{p1 \vee p2} = [\sigma_{p1}(R)] \cup_S [\sigma_{p2}(R)]$;

Last two laws are only for sets;
Projection: Let X and Y a set of attributes, and X is a subset of Y then:

- $\pi_X[\pi_Y(R)] = \pi_X(R)$;

Selection and Natural Join combined, with P predicate with only R attributes, Q predicate with only S attributes:

- $\sigma_P(R \bowtie S) = [\sigma_P(R)] \bowtie S$;
- $\sigma_Q(R \bowtie S) = R \bowtie [\sigma_Q(S)]$;
- $\sigma_Q(R \bowtie S) = \sigma_Q(R) \bowtie \sigma_Q(S)$;

Projection and Natural Join combined,

- With X a subset of R attributes, Z attributes in predicate P (subset of R attributes):
  - $\pi_X[\sigma_P(R)] = \{\sigma_P[\pi_X(R)]\}$, if Z subset of X;
  - $\pi_X[\sigma_P(R)] = \pi_X\{\sigma_P[\pi_{XZ}(R)]\}$, otherwise;
- With X a subset of R attributes, Y a subset of S attributes, Z intersection of R and S attributes:
  - $\pi_{XY}(R \bowtie S) = \pi_{XY}\{[\pi_{XZ}(R)] \bowtie [\pi_{YZ}(S)]\}$;

Other rules:

- $\pi_{XY}[\sigma_P(R \bowtie S)] = \pi_{XY}\{\sigma_P[\pi_{XZ'}(R) \bowtie \pi_{YZ'}(S)]\}$, where $Z' = Z \cup \{attributes\ used\ in\ P\}$;
- $\pi_{XY}[\sigma_P(R \times S)] = \pi_{XY}\{\sigma_P[\pi_X(R) \times \pi_Y(S)]\}$
- $\sigma_P(R \cup S) = \sigma_P(R) \cup \sigma_P(S)$;
- $\sigma_P(R - S) = \sigma_P(R) - \sigma_P(S)$;
- $R \bowtie_C S = \sigma_C(R \times S)$;
- $R \bowtie S = \pi_L[\sigma_E(R \times S)]$, where E is the condition that equates each pair of attributes from R and S with the same name, and L is the list of attributes that contains the union of the attributes of R and the attributes of S;

Rules for duplicate elimination where δ denote the duplicate elimination operator:

- $\delta(R \times S) = \delta(R) \times \delta(S)$;
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$;
- $\delta(R \cap_B S) = \delta(R) \cap_B \delta(S) = \delta(R) \cap_B S = R \cap_B \delta(S)$;

7.2.2 Guidelines

1. Repeat if and until possible:
   - Push *selections* over *projections*;
   - Group the *selections*;
   - Push *selections* over *Cartesian product*;
2. Eliminate useless *projections*;
3. Push *projections* down the *tree*, or add new *projections*;
4. Try to remove *duplicate eliminations*, or move them to a more convenient position of the *tree*;

5. Try to combine *selection* with *product* so it becomes an *equi-join* which is more efficient than two operations separately;
6. For each portion of *sub-tree* that consist of *nodes* with the same associative and commutative *operator*, group the *nodes* with these *operators* into a single *node* with many children;
7. *Natural-join*, *theta-join*, and *products* can be combined under certain circumstances:
    - If we replace *natural-join* with *equi-join,* so on common attributes;
    - If we add a *projection* to eliminates copies of *attributes* involved in a *natural join* that has become an *equi-join*;
    - The *theta-join* must be commutative;

7.3 Selecting Physical query plan

When we work at the *logical query plan level* we make *transformation* that are beneficial independently from the current statistics. When we work at **physical query plan level**, we aim at *transformations* that are effective with respect to the current *volume of data* we have. So we need to *estimate* the size of the result of the execution of the *operators*. The DBMS keeps statistic for every relation R:

- T(R): *tuples* of R;
- S(R): *bytes* in each *tuple* of R;
- B(R): *pages* of R;
- V(R,A): *distinct values* in R for *attribute* A;

For the size estimate of a **projection**, we should remember that the number of *tuples* doesn't change with *projection*, but the *size* of each *tuple* may decrease, and therefor the number of *pages* may decrease. $T(W) = f \times T(R)$.

When the **selection** condition C is the **AND** of several *atomic conditions* we can treat the *selection* as a cascade of simple *selections*, each of which check for one of the *conditions*. The effect will be that the *size* estimate for the result is the *size* of the original relation multiplies by the *selectivity factor*:

- 1/3 for any *inequality*;
- 1 for <>;
- $1/V(R, a)$ for any *attributed* compared to a *constant* in *condition* C;
- If the exercize gives you the $V(R, a)$ you can compute the cost of the selection as: $T(W) = T(R)/V(R, A)$;

When the *selection* condition is an **OR**, we can assume the sum of the number of *tuples* that satisfy each. A more precise estimate is to take the smaller of the size of R and the sum of the number of the *tuples* that satisfy each.

The estimation of the size of the **Natural join** of two *relations* will require two assumptions:

1. **Containment of values sets**:
   - If R and S are two *relations* with *attribute* Y in common, and $V(R,Y) \leq V(S,Y)$ then every Y-value of R will be a Y-value in S:

$$V(R1,A) \leq V(R2,A) \Rightarrow Every\ value\ A\ in\ R1\ is\ in\ R2$$

2. **Preservation of value sets**:
   - If we join a *relation* R with another *relation*, then an *attribute* A that is not a *join attribute* doesn't loose values from its set of possible values. So if A is an attribute in R but not in S, then $V(R \bowtie S, A) = V(R, A)$;

We have three cases, and we call X the *attributes* of R1, and Y the *attributes* of R2:

1. $X \cap Y = \varnothing$ so the *size estimation* is $R1 \times R2$;
2. $X \cap Y = A$ so for the *first assumption*:
   - If $V(R1,A) \leq V(R2,A)$ so every A value in R1 is in R2;
   - If $V(R2,A) \leq V(R1,A)$ so every A value in R2 is in R1;
   - So the size estimation is: $T(W) = \frac{T(R2) \times T(R1)}{max\{V(R1,A), V(R2,A)\}}$
3. $X \cap Y = A_1, ..., A_n$ but valid also for *equi-join*, the estimate of the number of *tuples* of the *join* is computed by multiplying T(R1) and T(R2), and then dividing by the larger of $V(R1,y)$ *and* $V(R2,y)$ for each *attribute* y in common to R1 and R2;

Size estimation for other types of **join**:

- The *equi-join* can be treated as a *natural join*;
- *Theta-joins* can be treated as if they were a *selection* following a *product* with additional observation:
  - An *equality condition* can be estimated using the method for *natural join*;
  - An *inequality comparison* between two *attributes* (like $R.a < S.b$) can be handled as for the *inequality* of the form $R.a < constant$, so a *selectivity factor* of 1/3.

Size estimation for other *operators*:

- For **bag union**, the *size* is the sum of the two *relations*;
- For **set union**, the *size* is approximated to the sum of the larger plus the half of the smaller;
- For **intersection**, the *result* can be as 0 as many as the smaller, so an approximation is half of the smaller;
- For **difference**, a good *estimation* is $T(R) - T(S)/2$;

- For **duplicate elimination**, a good *estimation* is the smaller between $T(R)/2$ and the product of all $V(R, a_i)$, that is the maximum number of distinct *tuples* in R;
- For **grouping and aggregation**, a good estimation is the smaller between $T(R)/2$ and the product of all $V(R, a_i)$, that are all the grouping attributes;

In order to turn a *logical query plan* into a *physical query plan* we can consider many different **physical plans derived**, and estimating the *cost* of each, and then pick the *physical query plan* with the **lowest estimated cost**. The *number of physical query* plans derivable from a single *logical plan* is enormous, so the *DBMS* applies *heuristics* for limiting the number of *physical plans* to be considered, these are the guidelines:

- Use *indexes* for *selection* of the form $A = c$ (or, $A > c$) on a stored *relation*, if available;
- If the *selection* involves one *condition* of the form $A = c$ plus other *conditions* on a stored *relation*, then use the *index*, if available and apply a further *selection operator* (a **filter**);
- If an argument R of the *join* has an *index* on the *join attributes*, then check whether it advantageous to use an *index-based join* with R in the *inner loop*;
- If an argument R of the *join* is *sorted* on the *join attribute*, then prefer a *join algorithm based* on *sorting* rather than a *hash-based* one;
- When computing the *union* or the *intersection* of three or more *relation*, *group* the smaller *relations* first;
- Apply specific algorithms for deciding *join order*;

Deciding the **order of the joins** is very important. Many of the *join algorithms* are *asymmetric*, so the roles played by the two argument *relation* are different, so the *cost* depends on which relation plays which role.

- For **instance**, the *one-pass join* reads one *relation* (called **build relation**) in the *buffer*, and the other *relation* one *page* at time (called **probe relation**). When we have a *join* of two *relations* we select the one whose **estimated size** is smaller as the left argument, and this is a good choice for *one-pass*, *nested loop* and *index-based join*.
- If the **join** is **not binary** that's more difficult. We will use a *greedy algorithm* for solving this problem (so doesn't guarantee to find the *optimal solution*), and this will result in a **left-deep join tree**.
- **Basis**:
  - Start with the pair of *relations* whose estimated *join size* if the *smallest*. The *join* of these *relations* becomes the **current tree**;
- **Induction**:
  - Find, among all *relations* not yet in the **current tree**, the *relation* that when *joined* with the *tree*, yield the relation of *smallest estimated size*. The new **current tree** has the the old *current tree* as its *left argument* and the *selected relation* as its *right argument*;

In order to complete the **physical query plane** we need to:

- Selection of algorithms to implement the *operations* of the *query plan* that have not been decided yet;
- Deciding when *intermediate result* will be *materialized* (stored in *secondary storage*) or *pipelined* (created in *main memory* and passed to other *operations*)
- Building a **comprehensive tree** using appropriate notation, to be passed to the *query execution engine*.

Materialization is an oblivious approach, we store the *result* of an *operation* in *secondary storage* for later usage. Instead **Pipelining** means that several *operations* are running at once, and the tuples produced by one *operation* are passed directly to the *operation* that uses it, without storing the intermediate tuples on the *secondary storage*. *Pipelining* is implemented through a networks of *iterators*.

- **Unary operators**, like *selection* and *projection*, are excellent candidate for *pipelining*;
- **Binary operators**, means that we don't store the *entire result* in *secondary storage*, and we store *relevant data* of the *operation* in the *buffer*.

For the **physical plan tree** we use a *tree* where:

- We indicate the *algorithm* used for the *operators* in the *nodes* corresponding to the *operators*;
- We indicate that a certain *intermediate relation* is *materialized* by a double line crossing the *edge* between the *relation* and its *consumer*, all the other *edges* are *pipelined*;
- In the *leaves* we indicate how we access the *relation* of the *database*;

Each relation in the *leaf* of the **logical query plan** will be replaced by one of the following **scan operator**:

- **TableScan(R)**: scan the *relation* R;
- **SortScan(R,L)**: R is scanned and *sorted* on the basis of *attribute list* L;
- **IndexScan(R,C)**: R is accessed through an *index* on *attribute* A, where A is the *attribute* mentioned in the *condition* C;
- **IndexScan(R,A)**: The entire *relation* R is retrieved via an *index* on *attributes* A (*index-only scan*);
- For *selection*, we can use **Filter(C)**;
- For sorting *intermediate relations*, we use **Sort(L)** where L is a *list of attributes*;
- For other *operations*, we indicate the *algorithm* used, and the number of *buffer frames* expected;