Advanced Data Management

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

[[CM0623-2] FOUNDATIONS OF ARTIFICIAL INTELLIGENCE
Academic Year 2022 - 2023

Student Nicola Aggio 880008

June 28, 2023

Contents

List of Figures

List of Tables

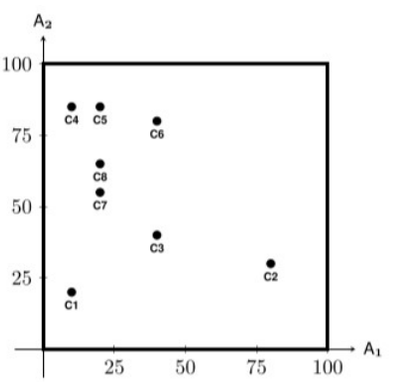
Chapter 1

Multidimensional data organizations

1.1 Types of data and queries

Let's consider multidimensional data representing points or regions in a k-dimensional space.
An example could be the following: consider a set of 8 records with two attributes A1 and A2 of type integer, which represent the latitude and longitude of cities, whose name will be used to denote the corresponding record, as shown in Picure 1.1. Picture 1.1 provides a 2-dimensional representation of the points.

City	A_1	A_2
C1	10	20
C2	80	30
C3	40	40
C4	10	85
C5	20	85
C6	40	80
C7	20	55
C8	20	65



The problems that will be considered are:

- **primary organization**: how to divide stored data across pages;
- **secondary organization**: how to quickly find the region containing the points in a specified rectangular area.

Both problems depend on the type of queries that are supported:

- point/region search: check if a point/region is present;
- spatial range search: points/regions in a rectangle or ball;
- k-NN: find the k-closest points/regions with respect to a selected point/region.

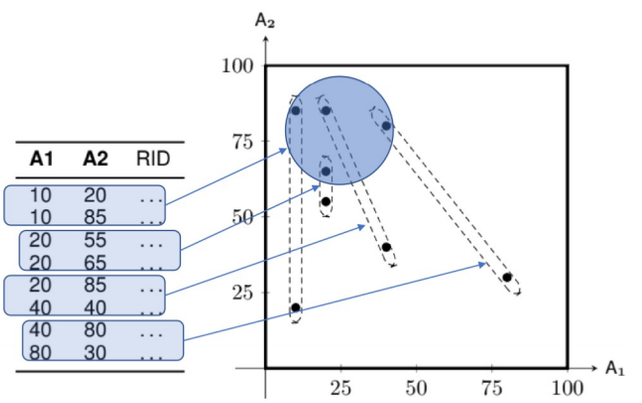
1.2 Multidimensional data organizations

There exist different multidimensional data organizations.

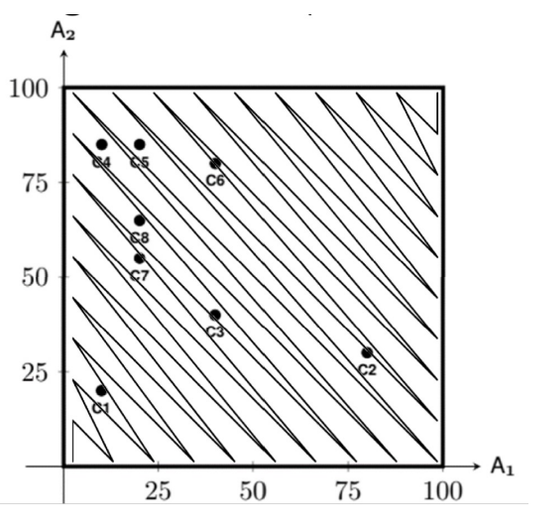
1.2.1 Linear order based

This organization let us to use traditional indexes in order to map positions, but it has a strong **precondition**: total order on the multidimensional data. Some examples of possible orders could be:

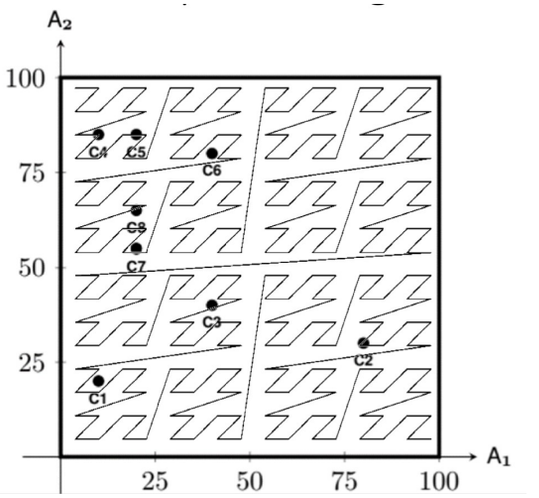
- multi-attribute lexicographic order: in this case the points are store in a B+-tree. This order is useful for point search, but almost useless for range search and k-NN.



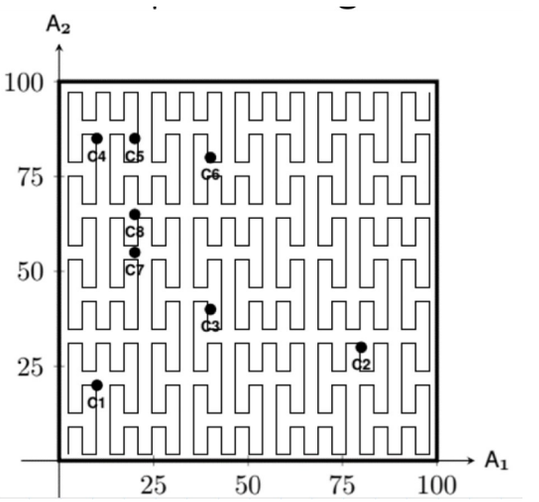
- diagonal order: in this case all the points are touched and it provides different type of granularity.



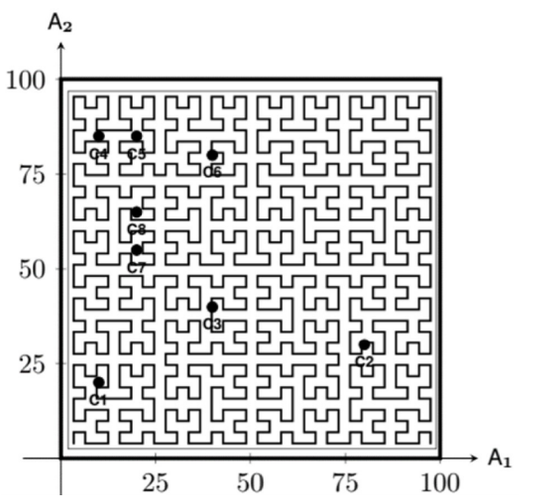
- Z-order



- Peano space filling curve



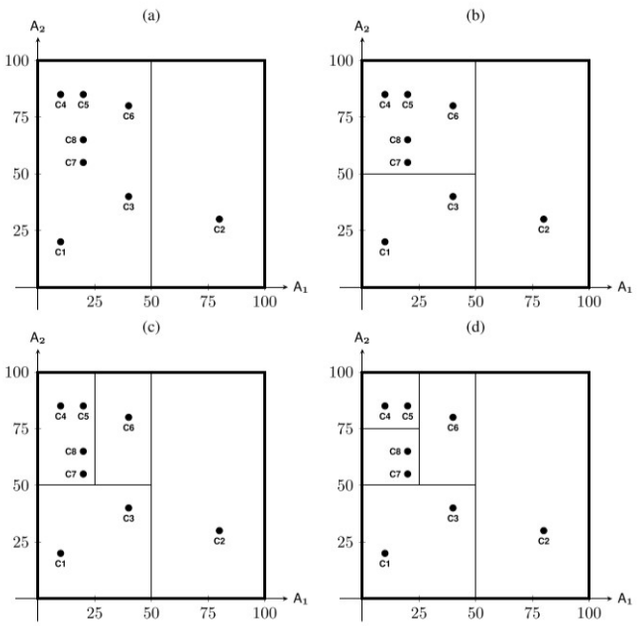
- Hilbert space filling curve



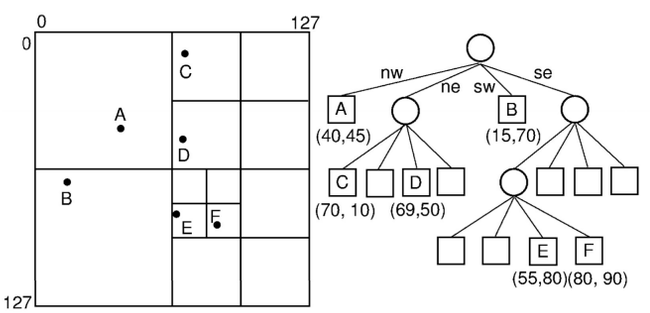
1.2.2 Space partitioning

In this case the focus is both on finding the partition of the space that results in (non-overlapping) regions that contain records that can be stored in a page and on how to quickly find the region containing the points for a specific query.
Let's focus on the different types of space partitioning:

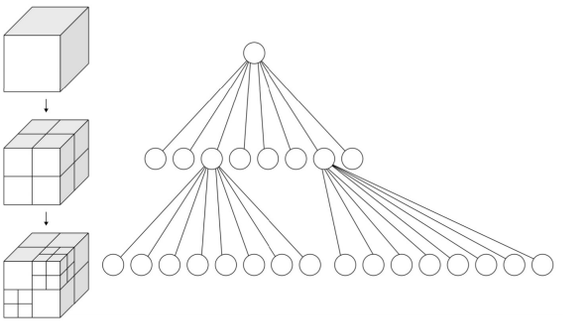
- classical partitioning: in this case the split is made according to a value of separation d for a specific attribute. When there is a new overflow from a page during data loading, a new split is done, but changing the attribute.



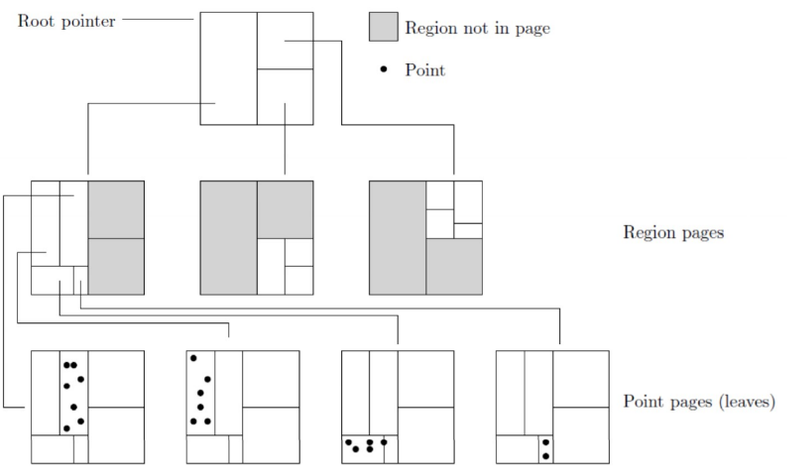
- point region quadtrees: this organization is used to organize bidimensional data, and it can be considered as the evolution of a binary tree into 2-dimension. Its limitation are:
  - it is not balanced;
  - it creates regions that are not needed;
  - the higher the dimension, the higher the number of regions that are created.



- octrees



- KDB-trees: they inherit the idea of having a threshold to split the domain, but they use coordinates at time and they circle the order of the splitting strategy. In order to find the correct position, starting from the root is easier than starting from the leaves, because in this case an additional tree must be kept in order to manage the partition volume.



1.2.3 G-trees

G-trees combine the ideas of data partitioning and of B+-trees as follows: data space is divided into non-overlapping regions of variable size identified by an appropriate code, then a total ordering is defined for partition codes, and they're stored in a B+-tree. If we consider a 2-dimensional case, assuming that data pages may contain 2 points, then a **partition code** is a binary string constructed as follows:

- the initial region is identified by the empty string;
- the first split is made along the X-axis, and the two partitions it produces are "0" and "1". Points $0 < x \leq 50$ are in partition "0", while points $50 < x \leq 100$ belong to partition "1";
- when a partition of the previous step is split along Y-axis, then the new partition codes become "00" and "01" and so on...

An example of G-tree is showed in Picture 1.1.

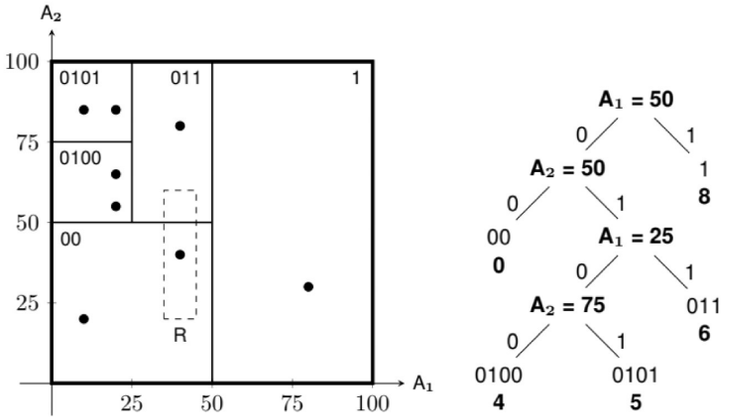


Figure 1.1: Example of G-tree

The Picture 3.1.2 represents the partition codes and the B+-tree which is used to store them.

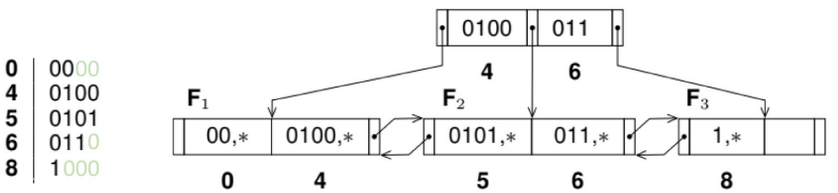


Figure 1.2: Example of G-tree

Operations

Now we focus on some possible operations which can be performed using these G-trees:

- **point search**: let M be the maximum length of the partition number of the G-tree, then the search for a point P with coordinates (x, y) proceeds as follows:

  1. the partition tree is searched for the code S_p of the partition that contains P, if it is present
  2. the G–tree is searched for the partition code S_p to check if P is in the associated page.

- **spatial range search**: a spatial range search looks for the points P_i with coordinates (x_i, y_i) such that $x_1 \leq x_i \leq x_2$ and $y_1 \leq y_i \leq y_2$, i.e. they are in the query region $R = \{(x_1, y_1), (x_2, y_2)\}$. The query result is found as follows:

  1. the G–tree is searched for the leaf node F_h of the partition containing the lower left vertex (x_1, y_1) of R
  2. the G–tree is searched for the leaf node F_k of the partition containing the upper right vertex (x_2, y_2) of R

3. for each leaf from F_h to F_k the elements S are searched such that $R_S = RegionOf(S)$ overlaps with the query region R, where $RegionsOf(S)$ is a function that maps the code S of a partition R in the coordinates of the lower left and upper right vertices of the partition

- **point insertion**:

1. the G–tree is searched for the leaf node F of the partition R_p that should contain it. Let S_p be the code of R_p
2. if R_p is not full, insert P, otherwise R_p is split in R_{p1} and S_{p1}, with codes $S_{p1} = S_p$"0" and $S_{p2} = S_p$"1". If the new strings have a length greater than M, M takes the value $M + 1$
3. the points in R_p and P are distributed in $RP1$ and R_{P2}.
4. the element (S_p, p_{Rp}) in the leaf F is replaced by (S_{p1}, p_{Rp1}) and (S_{p2}, p_{Rp2})

- **point deletion**:

1. let F be the leaf node with the partition R_p containing P, S_p the partition code of R_p, and S' the partition code of R' obtained from R_p with a split and therefore different from S_p for the last bit only
2. P is deleted from R_p and then two cases are considered:
   - R' has been split:
     * If the partition R_p becomes empty, then S_p is deleted
     * Otherwise the operation terminates
   - R' has not been split:
     * If the two partition cannot be merged, the operation terminates
     * Otherwise the two partition are merged

1.2.4 R*-trees

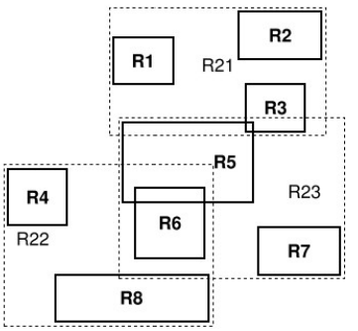
- an R*-tree is a dynamic tree structure perfectly balanced as a B+-tree, used for retrieval of multidimensional data according to ita spatial position;

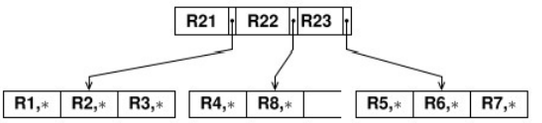
- terminal nodes of a R*-tree contain elements of the form (R_i, O_i), where:

  - R_i is the rectangular data;
  - O_i is its reference to the data nodes; for simplicity we denote O_i as *.

- non terminal nodes contain elements of type (R_i, p_i), where

  - p_i is a reference to the root of a subtree;
  - R_i is the minimum bounding rectangle countaining all rectangles associated with the child nodes.





The **properties** of R*-tree are:

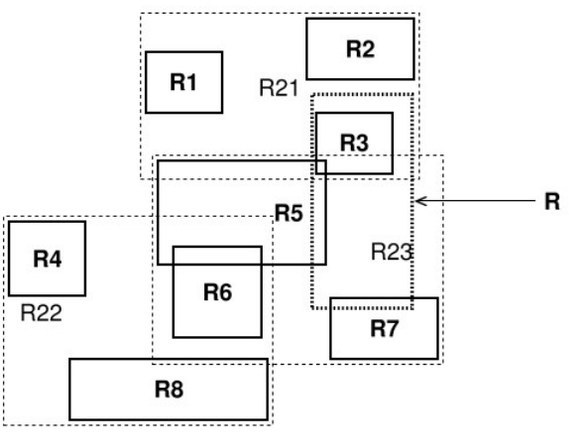
- each node has a number of elements between m = the minimum number of elements in a node, and M = the maximum;
- all leaf nodes are at the same level;

Differences with B+-trees:

- there is no sort order in R*-trees, unlike B+-trees;
- there may be overlap between regions associated with different elements of the same level, unlike B+-trees.

Operations

- **search overlapping data regions**: suppose we want to search all the overlapping regions to the region R:
  - the root is visited in order to look for elements (R_i, p_i), with R_i that overlaps with R;
  - for each element (R_i, p_i) found, the search proceeds in the subtree rooted in p_i: when a leaf node is reached, the data regions R_i in the search results are those with R_i that overlaps with R.

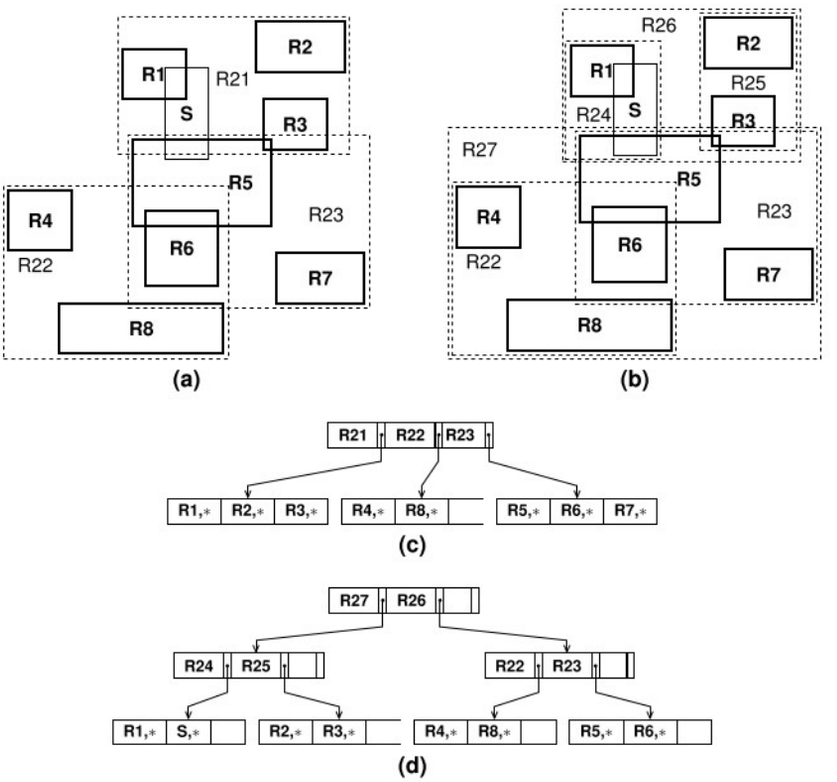


In the example, R_3, R_5 and R_7 overlaps with R.

- **insertion**: let S be a new data region to insert. The operation is similar to inserting a key in a B+-tree, since S is stored in a leaf node, and if there is an overflow, the node will be split into two nodes. In the worst case the division can propagate to the parent node up to the root. However, there is a significant difference with the insertion in B+-trees: in R*-trees, since the regions may overlap, the new data region S may overlaps with more of them, so it could be inserted in more leaf nodes. In this sense, the choice of the region in an internal node can be made according to the degree of overlap with S. After having selected the node N where to insert S, if an overflow does not occur, the region is recalculated and its value propagates in the parent node, otherwise:

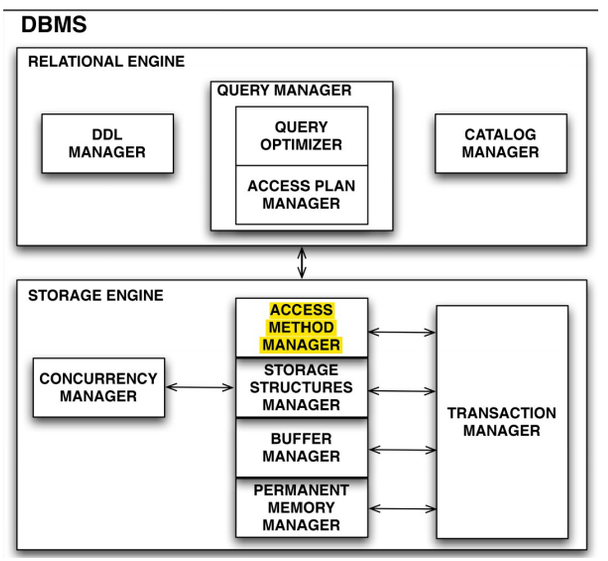
  1. if it is the first overflow, then a fraction p of the $M + 1$ entries are removed from the node and reinserted in the tree, in a sort of dynamic reorganization of the tree
  2. otherwise, the $M + 1$ elements are divided between two nodes, and two new elements are inserted in the parent node, and the process goes on to propagate the effects.

  Example: assume to insert region S (a): the process starts with the root node of the R*-tree in (c). Let the region $R21$ be selected for the insertion. Following the associated pointer, the leaf node to the left is considered, and since this node does not have enough space to contain S an overflow occurs. Being the first, we proceed with the reinsertion of R1. The region $R21$ is updated (not shown in the figure) and the reinsertion takes place in the same leaf node, causing another overflow and then a subdivision. Suppose that we get $R1, S$ and $R2, R3$. Let $R24$ and $R25$ be the regions containing $(R1, S)$ and $(R2, R3)$ (b). Consequently, two new elements have to be inserted into the root node to replace $R21$ (c). Since in the root node there is not enough space to contain four elements, there is an overflow. In the root the reinsertion it is not applied, but a subdivision is made. The result is that the old root node is replaced by two new nodes, one containing $(R24, R25)$ and the other $(R22, R23)$. Let $R26$ be the region containing $(R24, R25)$ and $R27$ be the region containing $(R22, R23)$. A new root is added with elements $R26$ and $R27$ (d).


R1
R2
R21
S
R3
R5
R4
R22
R6
R23
R7
R8
(a)
R26
R1
R2
R25
R24
S
R3
R27
R5
R4
R22
R6
R23
R7
R8
(b)
R21
R22
R23
R1,*
R2,*
R3,*
R4,*
R8,*
R5,*
R6,*
R7,*
(c)
R27
R26
R24
R25
R22
R23
R1,*
S,*
R2,*
R3,*
R4,*
R8,*
R5,*
R6,*
(d)

Chapter 2

Access methods management



The access method manager provides to the relational engine the operators used by its modules to execute the commands for the definition and use of DBs.

2.1 Storage engine

A DBMS is typically divided into:

- the **relational engine**, which includes modules to support the execution of SQL commands and interacts with the storage engine;
- the **storage engine**, which includes modules to execute the operations on data stored in the permanent memory.

Normally, the storage engine is not accessible to the user, who will interact with the relational engine.
While the interface of the relational engine depends on the data model features, the interface of the storage engine depends on the data structures used in permanent memory. To give a better idea of the **interface of a storage engine**, we will

consider the case of the relational system **JRS**, which stores relations in heap files and provides B+– tree indexes to facilitate data retrieval. The operators on data exported by the storage engine are procedural and can be grouped into the following categories:

- operators to create the DBs;
- operators to start and end a transaction;
- operators on heap file and indexes;
- operators about access methods available (= ways of retrieving records from a table) for each relation.

We now focus on the operators on DBs, on heap files, on indexes and on the operators about access methods.

2.2 Operators on DBs

The operators are:

- *createDB*, which creates a DB in the specified path;
- *createHeapFile*, which creates an heap file in the DB in the specified path;
- *createIndex*, which creates an index on a relation attribute, which could be a key, or on multiple attributes.

A DB, an heap file or an index can be deleted using *dropDB*, *dropHeapFile* and *dropIndex*

2.3 Operators on heap files

A DB table is stored in a heap file, in which there are operators to insert, delete, retrieve or update records with a specified RID, or to get the number of pages used and the number of the record. A **table** is a set of records where each record contains the same number of fields. Heap files support the *scan* operation to iterate through the records of a file.

2.4 Opeators on indexes

An index is a set of records (Value, RID), organized as a B+-tree: the Value is the search key, while RID is the identifier of the record with the search key Value. Any number of indexes can be defined on a relation, and a search key can be multi-attribute. The operators available on indexes are those to insert or delete elements, or to get data about the B+–tree used to store them, such as the number of leaves, minimum and maximum search key Value.

2.5 Access methods operators

> The Access Methods Manager provides the operators to transfer data between permanent memory and main memory in order to answer a query on a database.

Permanent data are organized as collections of records, stored in heap files, and indexes are optional auxiliary data structures associated with a collection of records. The operators provided by the Access Methods Manager are used to implement the operators of physical query plans generated by the query optimizer.
Records of a heap file or of an index are accessed by scans. A **heap file scan operator** simply reads each record one after the other, while an **index scan** provides a way to efficiently retrieve the RID of a heap file records with a search by key values in a given range. The records of a heap file can be retrieved by a serial scan or directly using their RID obtained by an index scan with a search condition. A heap file or index scan operation is implemented as an **iterator**, also called cursor, which is an object with methods that allow a consumer of the operation to get the result one record at a time.
When a heap file iterator is created, it is possible to specify the RID of the first record to return, while for index operators a hey ragne is specified.

2.6 Example of query execution plan

Let's consider some examples of programs that use access methods operators of the storage engine to execute simple SQL queries: in particular, the programs show a possible **query plan** that might be generated by the query optimizer of the relational engine (we will see that the query execution plan does not produce an execution plan of this type, but a physical plan).
Picture 2.6 show the execution plan of the given query using the heap files, while in Picture 2.6 we assume that there's an index *idxCity* on the attribute *City* os *Students*. Clearly, the second execution plan is more efficient.

```
SELECT  Name
FROM    Students
WHERE   City = 'Pisa';
```

```
HeapFile Students = HF_open("path", "bd", "Students", transId);
ScanHeapFile iteratorHF = HFS_open(Students);
while ( !iteratorHF.HFS_isDone() ) {
      Rid  rid = iteratorHF.HFS_getCurrent();
      Record  theRecord = Students.HF_getRecord(rid);
      if ( theRecord.getField(4).("Pisa") )
          System.out.println(theRecord.getField(1));
      iteratorHF.HFS_next();
}
Students.HF_close();
iteratorHF.HFS_close();
```

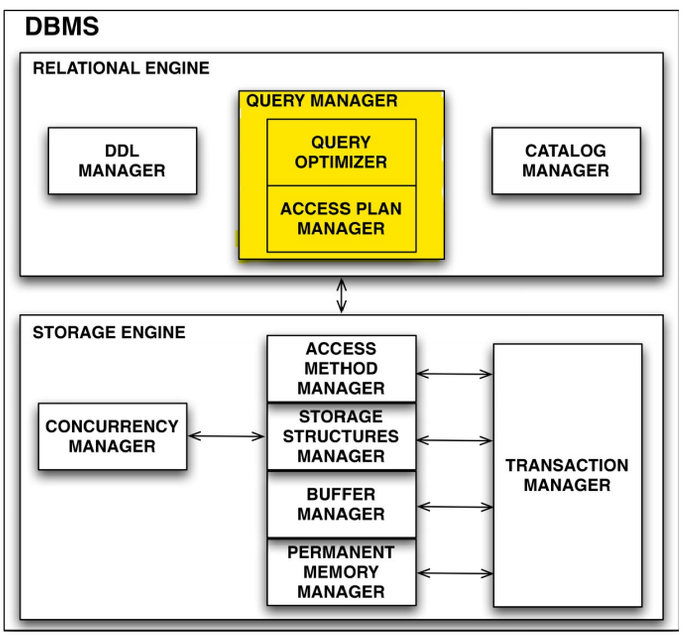
Figure 2.1: Example of query plan with heap files

```
HeapFile Students = HF_open(″path″, ″bd″, ″Students″, transId);
Index indexCity = I_open(″path″, ″bd″, ″IdxCity″, transId);
ScanIndex iteratorIndex = IS_open(indexCity, ″Pisa″, ″Pisa″);
while ( !iteratorIndex.IS_isDone() ) {
        Rid rid = iteratorIndex.IS_getCurrent().getRid();
        Record theRecord = Students.HF_getRecord(rid);
        System.out.println(theRecord.getField(1));
        iteratorIndex.IS_next();
}
interatorIndex.IS_close();
Students.HF_close();
indexCity.I_close();
```

Figure 2.2: Example of query plan with indexes

Chapter 3

Implementation of relational operators



This chapter focuses on the Query Manager, one of the critical components of any DB system, and, in particular, on the algorithms for evaluating relational algebra operations, and how to estimate their cost and size.

3.1 Assumptions and notation

3.1.1 Physical data organization

- each relation has attributes without null values, and it is stored in a heap file, or with the primary organization index sequential.
- to make the operations on relations more efficient, **indexes** on one or more attributes are used. The indexes are organized as a B+–tree and those for non-key attributes are inverted indexes. We also distinguish two types of indexes:
  - a **clustered index** is built on one or more attributes of a relation sorted on the index key. A relation can only have one clustered index;

– an **unclustered index** is built on one or more attributes which are not used to sort a relation

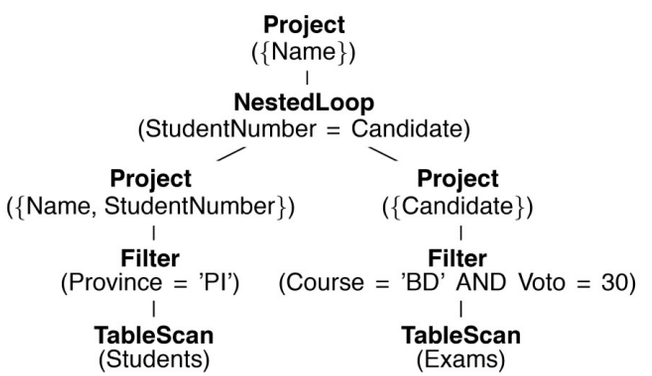
3.1.2 Physical query plan operators

> The **query optimizer** has the task of determining how to execute a query in an "optimal" way, by considering the physical parameters involved, such as the size of the relations, the data organization and the presence of indexes.

The problem is particularly difficult because, as we will see later, each relational algebra operator can be implemented in different ways and the query optimizer must use appropriate strategies to estimate the costs of the alternatives and choose the one with lowest cost.
The algorithm chosen by the optimizer to execute a query is represented as a tree, which is called **physical query plan**. The nodes of this tree are physical operators, each of which is a particular implementation of an operator of the **relational algebra extended on multisets**. This extension of relational algebra allows to manage multisets, which is what SQL table represent, and for this reason it supports the following operations:

- **projection with duplicates**: $\pi^b_X(O)$, with X attributes of O;
- **duplicate elimination**: $\sigma(O)$;
- **sorting**, used by the SQL statement ORDER BY: $\tau_X(O)$;
- **multiset union, intersection and difference**.

An example of physical query plan:



Each DBMS has its own **physical operators** and, for simplicity, we will consider those of the JRS system.
NOTE:

- for each physical operator we estimate the data access cost C and the cardinality of its result E_{rec};
- physical operators, like the operators of relational algebra, return collections of records with a type that depends on the operator.

Physical query plan execution

Physical operators are implemented as **iterators** that produce the records of the result one at a time on request. An iterator behaves as an object with state, and methods *open*, which initializes the process of getting records, *isDone*, which tests if the iterator has more data to return, *next*, *reset* and *close*. For example, a physical query plan with root *Plan* is executed as follows:

```
Plan.open();
whilenot Plan.isDone() do
        print(Plan.next());
Plan.close();
```

Physical operators are either **blocking** or **non-blocking**. A blocking operator, when is opened, must call next exhaustively on its operands before returning its first (or next) record (e.g. the sort operator)

3.1.3 Cost model

The **cost estimate** C of executing a physical operator is given by the number of pages read from or written to the permanent memory to produce the result

3.2 Physical operators for relation (R)

The records of a relation can be retrieved with any of the following operators:

- **TableScan(R)**: it returns the record of R in the order they're stored. This operator belongs to all primary organizations. The cost is:

$$C = N_{\text{pag}(R)}$$

- **SortScan(R,A_i)**: it return the records of R sorted in ascending order according to the attribute A_i.
  - sorting is done with *merge sort* algorithm
  - in general, the operator's cost depends on $N_{\text{pag}(R)}$, the number of pages B available in the buffer and the implementation of the *merge sort* algorithm;
  - this operator is available for all possible primary organizations.

  We assume that the cost is:

$$C = 3N_{\text{pag}(R)}$$

  , under the condition that $N_{\text{pag}(R)} < B^2$

- **IndexScan(R,I)**: it returns the records of R sorted in ascending order according to the attribute A_i values of the index I. The cost depends both on the type of index and on the type of attribute on which it is defined:

$$C = \begin{cases} N_{\text{leaf}}(I) + N_{\text{pag}(R)} & \text{if I is clustered} \\ N_{\text{leaf}}(I) + N_{\text{rec}(R)} & \text{if I is unclustered} \\ N_{\text{leaf}}(I) + \left\lceil N_{\text{key}}(I) \times \phi(\lceil N_{\text{rec}}(R)/N_{\text{key}}(I) \rceil, N_{\text{pag}(R))} \right\rceil & \text{if I is an inverted index} \end{cases}$$

Note that $N_{\text{leaf}}(I)$ represents the cost of accessing the index, while the other addend represents the cost for accessing the data, and it depends on the nature of the index I.

- **IndexSequentialScan(R,I)**: it returns the records of R, stored with primary organization *index sequential* I, sorted in ascending order on the primary key values. The cost is:

$$C = N_{\text{leaf}}(I)$$

NOTE: in all the cases, the cardinality of the result is:

$$E_{\text{rec}} = N_{\text{rec}(R)}$$

3.3 Physical operator for projection with duplicates (π^b)

- **Project(O,A_i)**: it returns the records of the projections of the records of O over the attributes A_i. The cost is:

$$C = C(O)$$

, if there are no subqueries.

The cardinality of the result is:

$$E_{\text{rec}} = E_{\text{rec}(O)}$$

This operator is always available.

- **IndexOnlyScan(R,I,A_i)**: it returns the sorted records of $\pi^b_{A_i}(R)$ by using the index I on the attributes A_i. The cost is:

$$C = N_{\text{leaf}}(I)$$

, while the cardinality of the result is:

$$E_{\text{rec}} = N_{\text{rec}(R)}$$

3.4 Physical operators for duplicate elimination (δ)

- **Distinct(O)**: returns the record of O sorted and without duplicates, i.e. it converts O from a multiset to a set. The crucial precondition is that the records of O are sorted. The cost is:

$$C = C(O)$$

- **HashDistinct(O)**: it eliminates the duplicates of O using an hash technique. Suppose that we have $B + 1$ pages in the buffer to perform duplicate elimination, then this technique is composed of two phases that use two different hash functions:

  - the *partitioning phase*, in which the first hash function h_i is applied to each record of O to distribute the records uniformly in the B pages: the result are B files, each containing a collection of records with the same hash value;
  - the *duplicate elimination phase*, in which the second hash function h_2 is applied to all record attributes of each page of each partition in order to eliminate the duplicates.

  The cost is:

$$C = C(O) + 2 \times N_{\text{pag}(O)}$$

We notice that the *HashDistinct* technique has the same cost of the *Distinct* tecnhique with the sorting of the operand records, but it does not procude a sorted result.

3.5 Physical operators for selection (σ)

- **Filter(O, ψ)**: it returns the records of O satisfying the condition ψ. The cost is:

$$C = C(O)$$

  This operator is always available.

- **IndexFilter(R,I,ψ)**: it returns the records of R satisfying the condition ψ with the use of the index I, defined on attributes ψ ; more specifically:

  - it uses the index to find the sorted set of RIDs satisfying the condition, using **RIDIndexFilter(I, ψ)**;
  - retrieves the records of R using **TableAccess(O,R)**.

The cost is given by:

$$C = C_I + C_D$$

, where C_I and C_D depend on the type of index and the type of attributes on which it is defined.

- **IndexSequentialFilter(R,I,ψ)**: it returns the sorted records of R, stored with the primary organization *index sequential* I, satisfying the condition ψ. The cost is:

$$C = \lceil s_f(\psi) \times N_{\text{leaf}()I} \rceil$$

- **IndexOnlyFilte(R,I,A_i,ψ)r**: it returns the sorted records of $\pi^b_{A_i}(\sigma_\psi(R))$, using only the index I. The cost is:

$$C = \lceil s_f(\psi) \times N_{\text{leaf}()I} \rceil$$

- **OrIndexFilter(R,I_i,ψ_i)**: it returns the records of R satisfying the disjunctive condition $\psi = \psi_1 \vee \psi_2 \vee ... \vee \psi_n$ using the index I_i for each ψ_i. More specifically:
  - it uses the index to find a sorted union of the RID lists of records matching each terms ψ_i;
  - retrieves the records of R.

  The cost is:

$$C = C_I + C_D$$

  , where

$$C_I = \left\lceil \sum_{k=1}^{n} C_I^k \right\rceil$$

  and

$$C_D = \lceil \Phi(E_{\text{rec}}, N_{\text{pag}}(R)) \rceil$$

- **AndIndexFilter(R, I_i, ψ_i)**: it returns the records of R satisfying the conjunctive condition $\psi = \psi_1 \wedge \psi_2 \wedge ... \wedge \psi_n$ using the index I_i for each ψ_i. More specifically:
  - it uses the index to find a sorted intersection of the RID lists of records matching each terms ψ_i;
  - retrieves the records of R.

The cost is:

$$C = C_I + C_D$$

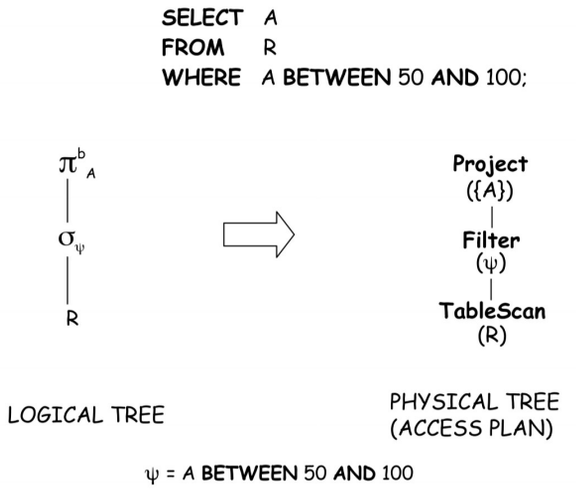
, where

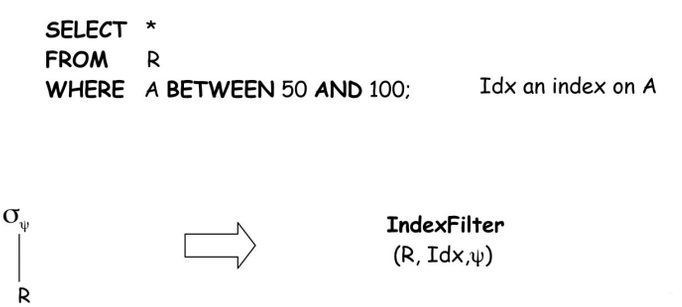
$$C_I = \left\lceil \sum_{k=1}^{n} C_I^k \right\rceil$$

and

$$C_D = \lceil \Phi(E_{\text{rec}}, N_{\text{pag}}(R)) \rceil$$

Picture 3.5 and 3.5 show two examples of access plans that are constructed from two simple queries: note that in the second query we can use IndexFilter because there's an index on the attribute A and , but we cannot use IndexOnlyFilter because we have "SELECT *".





3.6 Physical operators for grouping (γ)

The records of R are partitioned according to their values in one set of attributes A_i, called the *grouping attributes*. Then, for each group, the values in certain other attributes are aggregated with the functions f_i. The result of this operation is one record for each group. To execute the grouping of the operand O result, the following physical operators are used, which differ in the way the records of O are partitioned:

- **GroupBy(O,A_i,f_i)**: the records of O are sorted on the grouping attributes A_i, so that the records of each group are next to each other. Note that in the set f_i there are the aggregation functions present in the SELECT and HAVING clauses. The cost is:

$$C = C(O)$$

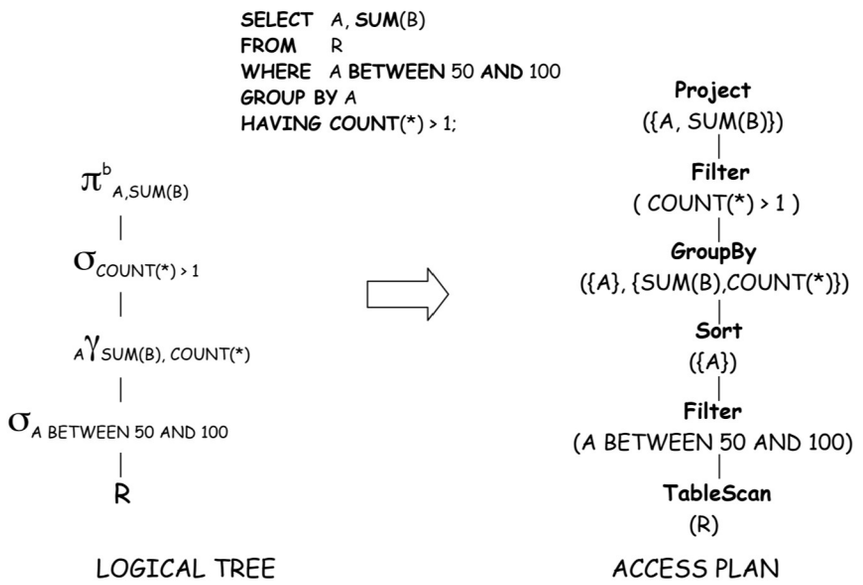
- **HashGroupBy(O,A_i,f_i)**: the records of O are partitioned using an hash function on the attributes A_i:

  - *partitioning phase*: using the hash function h_1, a partition is created;
  - *grouping phase*: using the hash function h_2 to all grouping attributes, the records of each partition are grouped.

  The cost is:

$$C = C(O) + 2 \times N_{\text{pag}(O)}$$

Picture 3.7 shows an example of GROUP BY query that is transformed into an access plan.



3.7 Physical operators for join ($\bowtie$)

The JOIN operation can be considered as cartesian product + filter operation: the cartesian product crates all the possible pairs, while the filter operation removes the pairs in which the primary key does not match. However, if we have two sets A and B, then the number of pairs is $|A \times B|$, and most of them are discarded, so it is an inefficient operation. For this reason, the two possible options to optimize the JOIN operation is:

- reduce the number of candidates using an index;
- reduce the cost of access, using the same number of candidates.

We only consider $O_E \bowtie_{\psi_J} O_I$, where:

- O_E is the external operand;
- O_I is the internal operand;
- ψ_J is the join condition.
- **NestedLoop(O_E, O_I ψ_J)**:

```
for each r ∈ O_E do
    for each s ∈ O_I do
        if ψ_J then add < r, s > to the result;
```

  The cost is:

$$C = C(O_E) + E_{\text{rec}(O_E)} \times C(O_I)$$

  , but the order of the loops matter! From the cost function we can see that all the pages of the internal operand are accessed for each page of the external operand, so it is more convenient to have as external operand the relation with more tuples.

- **PageNestedLoop(O_E, O_I, ψ_J)**: this operator exploit the buffer in order to reduce the number of disk accesses: in particular, the result of O_I is scanned once per page of O_E, instead of per each record of O_E. For this reason, the larger the page size, the better in terms of computational complexity.

```
for each page p_r of O_E do
    for each page p_s of O_I do
        for each r ∈ p_r do
            for each s ∈ p_s do
                if ψ_J then add < r, s > to the result;
```

  The cost is:

$$C = C(O_E) + N_{\text{pag}(O_E)} \times C(O_I)$$

  In this case, the algorithm cost is lower when the external relation is the one with fewest pages.

NOTE:

- both NestedLoop and PageNestedLoop do not have any knowledge about the predicates, so no index is used;
- PageNestedLoop is better than NestedLoop, being $N_{\text{pag}(O_E)} < N_{\text{rec}(R)}$, but it does not produce a sorted result.
- **IndexNestedLoop**(O_E, O_I, ψ_J): in this case we have some knowledge about the predicates, in particular we can use an index in order to filter the records and reduce the number of candidates. However, the preconditions are:
  - the join is an *equi-join*. It can be used not only for equi-join, provided that it has an access method for the predicate;
  - there is an IndexFilter on the internal operand.

  for each $r \in O_E$ **do**
  for each $s \in$ **IndexFilter**$(S, I, S.A_j = r.A_i)$ **do**
  add $< r, s >$ to the result;

  The cost is:

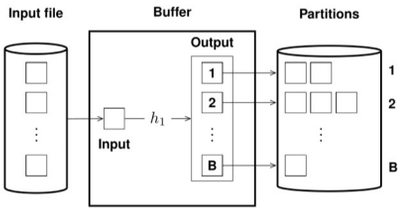
  $$C = C(O_E) + E_{\text{rec}(O_E)} \times (C_I + C_D)$$

  , where $(C_I + C_D)$ is the cost to retrieve the matching records of S with a record of O_E that depends on the index type.
- **MergeJoin**(O_E, O_I, ψ_J): in this case the precondition are:
  - the join is an *equi-join*;
  - O_E and O_I are sorted on the join attributes $O_E.A_i$ and $O_I.A_i$;
  - the join attribute $O_E.A_i$ is a key.

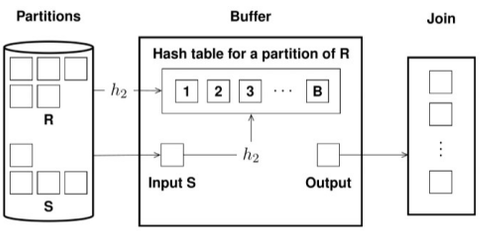
  The cost is:

  $$C = C(O_E) + C(O_I)$$

  This operator is efficient provided that O_E and O_I are sorted, otherwise we have to consider some additional costs.
- **HashJoin**(O_E, O_I, ψ_J): similarly to HashDisinct and HashGroupBy, this operator computes the join in two phases:
  - in the *partitioning phase*, the records of O_E and O_I are partitioned using the first hash function h_1 applied to the join attributes. If two tuples are joined, they belong to the same partition but in different pages.



– in the *probing phase*, for each partition B_i
  * the records of O_E are read and inserted into buffer hash table with B pages using the function h_2 applied to the join attribute values;
  * the records of O_I are read one page at a time, and which of them join with those of O_E is checked with h_2, and they're added to the result;



The costs are:

$$C_{\text{partitioning}} = C(O_E) + C(O_I) + N_{\text{pag}(O_E)} + N_{\text{pag}(O_I)}$$

$$C_{\text{probing}} = N_{\text{pag}(O_E)} + N_{\text{pag}(O_I)}$$

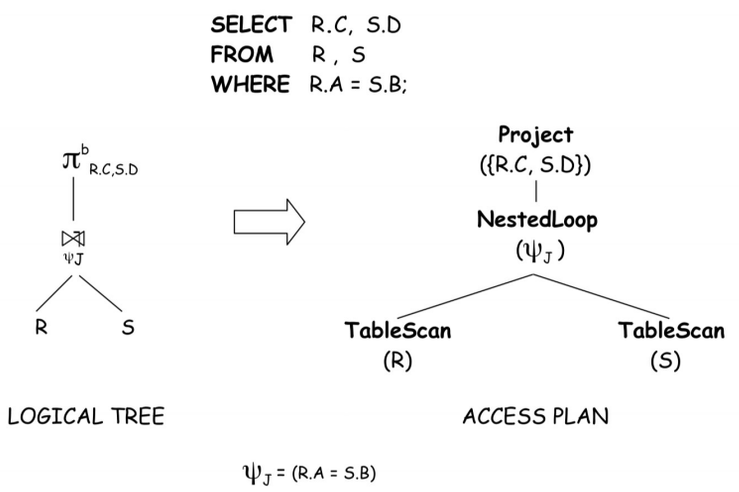
, so the total cost is:

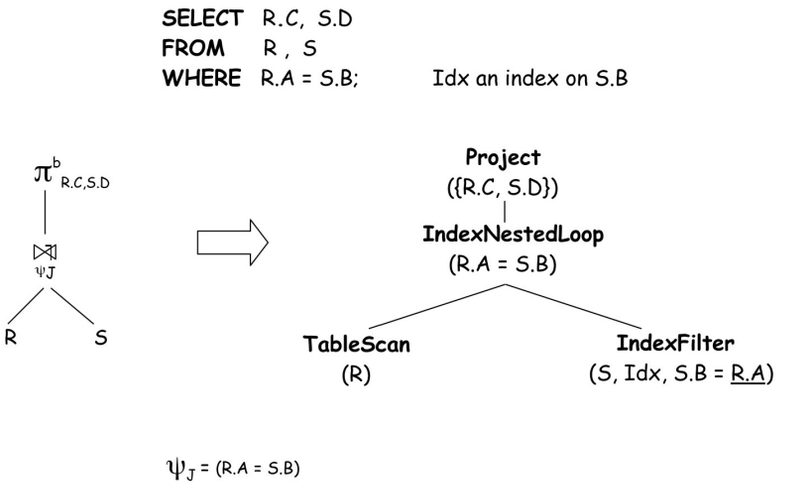
$$C = C(O_E) + C(O_I) + 2 \times (N_{\text{pag}(O_E)} + N_{\text{pag}(O_I)})$$

In general, since $O_E \bowtie_{\psi_J} O_I = \sigma_{\psi_J}(R \times S)$, the **cardinality of the result** is :

$$E_{\text{rec}} = \left\lceil s_f(\psi_J) \times E_{\text{rec}(O_E)} \times E_{\text{rec}(O_I)} \right\rceil$$

Pictures 3.7 and 4 show two examples of join queries that are trasformed into access plans.





3.8 Physical operators for set and multiset operations

- **Union**(O_E,O_I), **Except**(O_E,O_I) and **Intersect**(O_E,O_I) require that the operand are sorted and do not have any duplicate element. In this case the cost is:

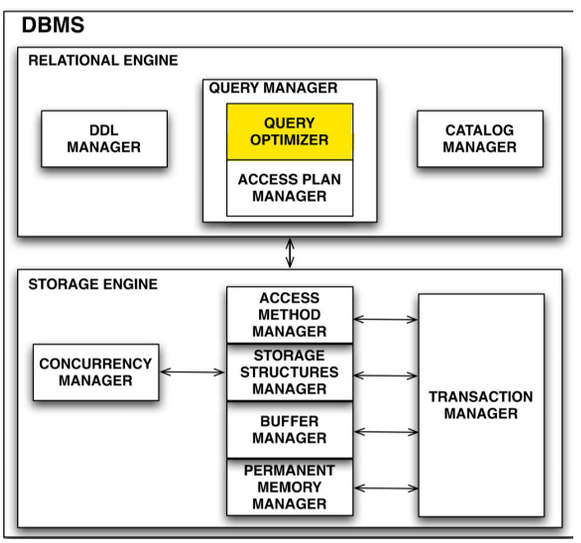
$$C = C(O_E) + C(O_I)$$

- **UnionAll**(O_E,O_I), **ExceptAll**(O_E,O_I) and **IntersectAll**(O_E,O_I) do not eliminate duplicates, so they implement the multiset operations. The cost is:

$$C = C(O_E) + C(O_I)$$

Chapter 4

Query optimization



In this chapter the focus is on query processing and on how to find the best plan to execute a query. In particular, we study the optimizer organization, a fundamental component of the **Query Manager**, which selects the optimal physical plan to execute queries using the operators and data structures of the Storage Engine. We will also show how functional dependencies, well known for relational schema design, are also important for query optimization.

4.1 Introduction

The **goal** of the optimizer is to find an efficient plan to execute a query. This problem is complex because:

- a query can be written in several equivalent ways;
- a relational algebra operator can be implemented with different physical operators.

, so the optimizer will use some heuristic methods to find a good solution quickly. The optimization can be:

- **dynamic**, i.e. the physical plan is generated at run time when the query is executed, taking into account some informations about the DB and the available data structures;

- **static**, i.e. the physical plan in generated at the program compilation time

4.1.1 Query processing phases

1. **query analysis**: the correctness of the SQL query is checked and it is transformed into relational algebra, i.e. the *initial logical query plan*

2. **query transformation**: the initial logical plan is transformed into an equivalent one that provides a better query performance

3. **physical plan generation**: alternative algorithms for the query execution are considered and the *physical query plan* with the lowest cost is chosen. This step makes query processing hard, because there's a large number of alternative solutions to consider in order to choose the best one

4. **query evaluation**: the physical plan is executed

4.2 Query analysis phase

1. lexical and syntax query analysis

2. semantic query analysis, in which the system check both the query semantic correctness and that the user has appropriate privileges

3. simplification of the **WHERE** condition using:

   - equivalence rules of boolean expressions;
   - elimination of contradictory conditions;
   - elimination of NOT conditions;

4. generation of the internal representation of the query as an **initial logical query plan**, represented as an expression tree of relational algebra operations. For example, the query:

```
SELECT  PkEmp, EName
FROM    Employee, Department
WHERE   FkDept = PkDept AND
        DLocation = 'Pisa' AND ESalary = 2000;
```

   is transformed into the following logical plan:

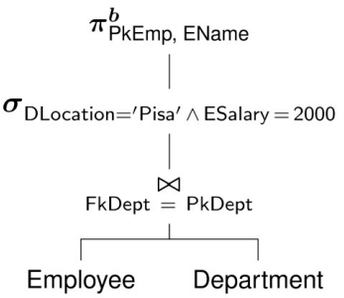


Figure 4.1: Initial logical query plan

4.3 Query transformation phase

In this phase, the *initial logical query plan* is transformed into an equivalent one using a set of equivalence rules, for example the *cascading of selections* or the *commutativity of selection and projection*. In general, the following rules are followed:

- selections are pushed below projections and they're grouped;
- selections and projections are pushed below joins;
- unnecessary projections are deleted;
- projections are grouped.

Picture 4.4 shows an example of transformation of the initial logical query plan of Picture 4.

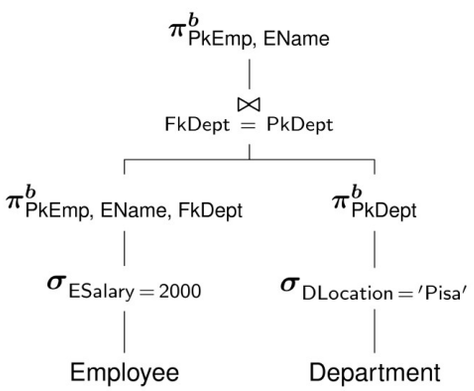


Figure 4.2: Transformation of the initial logical query plan

4.3.1 DISTINCT elimination

The DISTINCT operation requires SORT, which is costly, so we use the functional dependencies theory to discover if an interesting functional dependency can be inferred into the result of a query. In general, let A be the set of attributes of the result and K be the union of the attributes of the key of every table used in the query, then if $A -> K$, then the operator δ for duplicate elimination is unnecessary.

4.3.2 GROUP BY elimination

As DISTINCT, also GROUP BY requires the SORT operation. In this case, the elimination can happen:

- if there's only a single group;
- each group is composed of a single tuple

4.3.3 WHERE-subquery elimination

In general, to execute such queries, the optimizer generate a physical plan for the subquery, which then will be executed for each record processed by the outer query. Therefore, the presence of a **subquery** makes the **physical plan more expensive**, and for this reason techniques have been studied to transform a query into an equivalent one without the subquery, which the optimizer processes more efficiently. In particular, all the subqueries can be rewritten using EXISTS and NOT EXISTS, and in certain cases they can be eliminated with the introduction of JOIN.

```
SELECT  R1.A1,...,R1.An
FROM    R1
WHERE   [Condition C1 on R1 AND]
            EXISTS ( SELECT  *
                     FROM    R2
                     WHERE   Condition C2 on R2 and R1 );
```

is equivalent to the join query

```
SELECT  DISTINCT  R1.A1,...,R1.An
FROM    R1, R2
WHERE   Condition C2 on R1 and R2
        [AND Condition C1 on R1];
```

NOTE: by using the join we create all the possible pairs, so we must check for DISTINCT!
Complex queries are much easier to write and understand if **views** are used, either by creating them with **CREATE VIEW** or by using **WITH** (the second one creates temporary views available only to the query in which the clause occurs).In general, when a query uses a view, the optimizer generates a physical sub-plan for the SELECT that defines the view, and optimizes the query considering the scan as the only access method available for the result of the view.

4.4 Physical plan generation phase

The goal of this phase is to find a plan to execute a query, among the possible ones, which has the **minimum cost** on the basis of the available information on storage structures and statistics. The main steps of this phase are:

- generation of alternative physical query plans;
- choice of the physical query plan with the lowest estimated cost.

To estimate the cost of a physical query plan it is necessary to estimate, for each node in the physical tree:

- the cost of the physical operator;
- the size of the result and if the result is sorted.

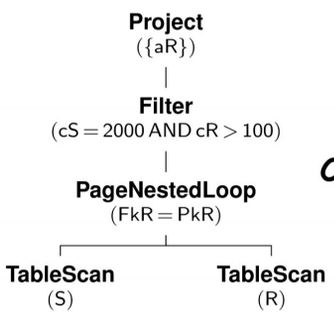
Let's see an example of alternative physical plans for a query and their cost. Let's consider a DB and the following query:

R(PkR :integer, aR :string, bR :integer, cR :integer)
S(PkS :integer, FkR :integer, FkT :integer, aS :integer, bS :string, cS :integer)
T(PkT :integer, aT :int, bT :string)

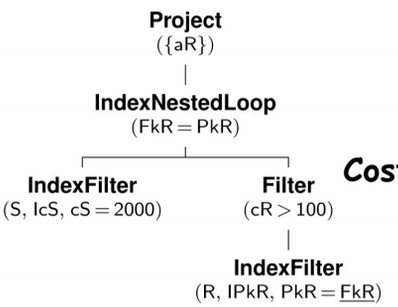
and the query:

```
SELECT   aR
FROM     S, R
WHERE    FkR = PkR  AND  cS = 2000  AND  cR > 100;
```

, and let's consider two alternative physical query plans: the first one uses the join operator **PageNestedLoop**, while the second one uses the available **indexes**.



Cost = 501000



Cost = 1010

Clearly, the second physical plan has better performances than the first one.

4.4.1 Single relation queries

If the query uses just one relation, the operations involved are the projection and selection.
If there are **no useful indexes**, the solution is to perform a scan + projection.
Otherwise, if there are useful index we can:

- **use of single-index**: the one of minimal cost is used, and it tests if the record satisfy the conditions and projection is applied. Note that the records of the result are sorted on the index attribute;
- **use of multiple-indexes**: the operation is completed by testing whether the retrieved records satisfy the remaining conditions and the projection is applied;
- **use of index-only**: if all the attributes of the condition of the SELECT are included in the prefix of the key of an index, the query can be evaluated using only the index with the query plan of minimum cost.

4.4.2 Multiple relation queries

Queries with two or more relations in the FROM clause require joins (or crossproducts), and finding a good physical query plan for these queries is very important to improve their performances.
In the following, for simplicity, we will consider an optimization algorithm based on the idea of generating and searching a **state space** of possible solutions to find

the one with minimal cost. The state space is constructed step-by-step starting with an initial state and repeatedly applying a set of operators to expand a state s into other ones, called the *successors* of s. Each state s corresponds to a relational algebra subexpression of a given query Q to optimize, and the successors of s are larger subexpressions of Q. The cost of a state s is that of the best physical plan to execute the expression associated to s.
The state space is represented as a tree of nodes, where the root is the empty subexpression, the first level nodes are the query relations or selections and projections on each of them; the following levels are alternative joins of the algebraic expressions of the previous level. The node of the "optimum" state is the expression the contains all the query joins, which is then extended with other operators (e.g. project, sort or group by) to become the final state of the query expression, with the associated minimal cost physical query plan.
As queries become more complex, the full search algorithm cannot find the overall best physical query plan in a reasonable amount of time because of the **exponential nature of the problem**. Therefore, several **heuristics** have been proposed in the DBMS literature that allow the query optimizer to avoid bad query plans and find query plans that, while not necessarily optimal, are usually "good enough". Some examples are *limitation of the number of successors*, *greedy search* and so on..

4.4.3 Other queries

- **DISTINCT queries**: assuming that the operation is performed by sorting, the physical plan for a SELECT ORDER BY is generated, and then extended with the physical operator **Distinct(O)**;

- **GROUP BY queries**: the optimizer produces a physical plan for the SELECT only, without considering the GROUP BY clause, which produces the result sorted on the grouping attributes. The physical plan is then extended with a physical operator for the **GroupBy** and the operator **Project** over the SELECT attributes. If the SELECT has also a HAVING clause, the physical plan is extended with a selection operator, and the operator **GroupBy** computes all the aggregate functions used in the HAVING and SELECT clauses;

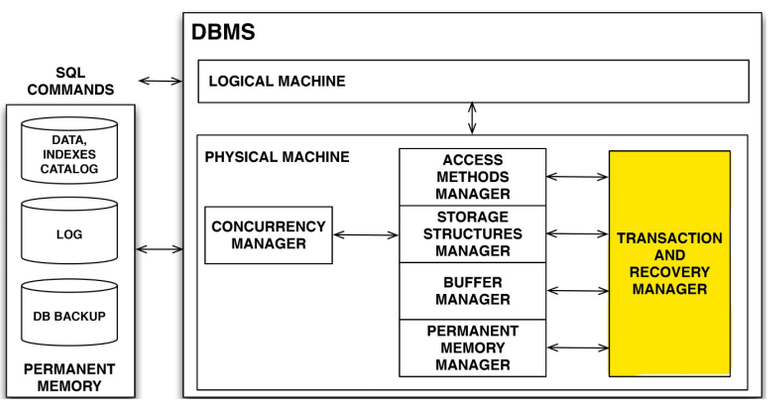
  Pre-grouping tranformation

  The standard way to evaluate join queries with grouping and aggregations is to perform the join first, but sometimes it is convenient to anticipate grouping operations. However, complex conditions are needed to perform this optimization, which also depends on the type of aggregate functions.

- **Queries with set operations**: we only consider the UNION operator. The optimizer is used to generate the physical plans for the two SELECT of the UNION, with duplicate elimination operators, which then become the operands of the **Union** operator.

Chapter 5

Transaction and recovery manager



One of the most important features of a DBMS are the techniques provided for the solution of the *recovery* and problems, to allow the users to assume that each of their applications is executed both as if there were no failures, and that there were no interferences with other applications running concurrently. The solutions of these problems are based on the abstraction mechanism called *transaction*. The correct implementation of transactions requires some of the most sophisticated algorithms and data structures of a DBMS. In this chapter we will focus on transactions as a mechanism to protect data from failures, while in the next one we will examine the aspects of transactions concerning the concurrency control to avoid interference.

5.1 Transactions

A database is said to be in a *consistent state* if all the integrity constraints are satisfied, and a *transaction* is a mechanism to properly group operations on the database into atomic units of work. A transaction is *correct* if it causes the DB to change from a consistent state to another consistent state, even in a system failure and by handling concurrent transactions.

5.1.1 Transactions from the programmer's point of view

A **transaction** is a sequence of operations on the DB with the following ACID properties:

- **atomicity**: only transactions terminated normally (*committed transactions*) change the database, otherwise the database remains unchanged;
- **isolation**: when a transaction is executed concurrently with others, the final effect must be the same as if it was executed alone;
- **durability** the effects of committed transactions on the DB survive system and media failures.

NOTE:

- atomicity, isolation and durability are provided by a DBMS, while consistency cannot be ensured by the system when the integrity constraints are not declared;
- the isolation property is sometime called the *serializability* property: when a transaction is executed concurrently with others, the final effect must be the same as a serial execution of committed transactions, i.e. the DBMS behaves as if it executes the transactions one at a time;
- the *Transaction and Recovery Manager* guarantees the atomicity and durability properties, while the isolation property is guaranteed by the *Concurrency Manager*

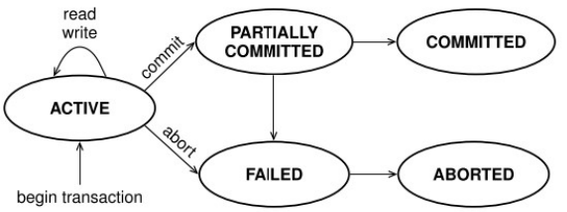
5.1.2 Transactions from the DBMS's point of view

A **transaction** for the DBMS is a sequence of read and write operations which start and end with the following transaction operations:

- *beginTransaction*: beginning of the transaction;
- *commit*: successful termination of the transaction and the system has to make its updates durable;
- *abort*: abnormal termination, the system has to undo its updates.

NOTE: the execution of the operation does not guarantee the successful termination of the transaction, because it is possible that the transaction updates cannot be written to the permanent memory, and therefore it will be aborted.
Picture 5.6 shows the different states of a transaction execution.

- a transaction enters into the *active* state immediately after it starts execution, where it stays while it is executing;
- a transaction moves to the *partially committed* state when it ends;



- a transaction moves to the *committed* state if it has been processed successfully and all its updates on the database have been made durable. In this state we have the warranty that durability is guaranteed;

- a transaction moves to the *failed* state if it cannot be committed or it has been interrupted after a transaction failure while in the active state;

- a transaction moves to the *aborted* state if it has been interrupted and all its updates on the database have been undone. In this state we have the warranty that atomicity is granted.

Note that the read/write and commit operations are executed by the system when required by the transaction, while the abort operation is executed by the system either when required by the transaction or when a system failure occurs, independently from the transaction requests.

5.2 Types of failures

We assume that the occurrence of a failure is always detected, and this causes:

- the immediate interruption of a transaction or of the whole system;

- the execution of specific *recovery procedures* to ensure that the DB only contains the updates produced by committed operations.

The following failures may occur:

- **transaction failures**: is an interruption of a transaction which preserves both the buffer and the permanent memory. It may occur either due to system errors or due to error in a constraint;

- **system failure**: is a system interruption, where the content of the buffer is lost, but content of the permanent memory remains intact. In this case the DBMS is restarted;

- **media failure**: is an interruption of the DBMS in which the content of the permanent memory is corrupted or lost. In this case the recovery manager uses a backup to restore the DB.

5.3 Database system model

Looking at 5 we notice that the permanent memory consists of three main components: the DB, the Log and the DB backup, which are used by the recovery procedure in the case of failures. Note that these three components are stored in distinct physical devices.
The *Transaction and Recovery Manager* performs the following tasks:

- execution of read, write, commit and abort operation on behalf of transactions and by cooperating with the *Permanent Memory Manager*, the *Buffer Manager* etc..;
- management of the log;
- execution of a *restart* command after a system failure, in order to guarantee that the DB only contains the updates of the committed transactions;
- execution of techniques to prevent data loss.

In the next sections the data structures and algorithms used by the recovery manager will be discussed. To simplify the presentation we assume that:

1. the database is just a set of pages;
2. each update operation affects a single page, and it is performed by modifying an in-memory copy of the page and then by writing it to disk only when the *Buffer Manager* decide to do it;
3. the operation of transferring a page from the buffer to the permanent memory is an atomic operation;
4. if different transactions are concurrently in execution, they read and write different pages.

5.4 Data protection

We saw that there exists different types of failures, but the different techniques that are used in these situations share the common principle of **redundancy**: to protect the database, the DBMS maintains some redundant information during normal execution of transactions in order to better reconstruct a consistent state, a process called **recovery**.

5.4.1 DB backup

The DBMS provides facilities for periodically making a backup copy of the DB.

5.4.2 Log

During the normal use, the history of the operations performed on the database from the last backup is stored in the **log**.

- for each transaction we write in the log when the transaction starts, commits, aborts and modifies a page;

- each log record is identified through the so called **LSN** (Log Sequence Number), that is assigned in a strictly increasing order. It can be viewed as the serial number of the position of the first character of the record in the file;

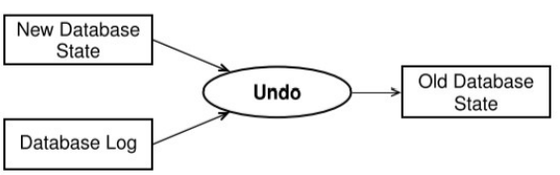
Operation	Data	Information in the log
beginTransaction		(*begin*, 1)
$r[A]$	$A = 50$	No record written to the log
$w[A]$	$A = 20$	$(W, 1, A, 50, 20)$ — Old and new value of A are written to the log
$r[B]$	$B = 50$	No record written to the log
$w[B]$	$B = 80$	$(W, 1, B, 50, 80)$
commit		(*commit*, 1)

NOTE:

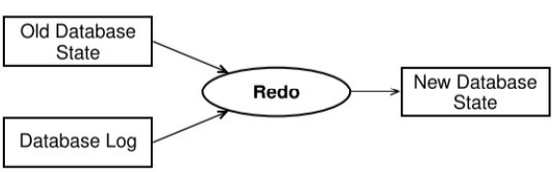
- the read operations do not affect the state of the database, and for this reason no record is written to the log;
- the exact content of the log depends on the algorithms of the transaction manager;
- in general a log is stored in a file buffered for efficiency reasons;
- we assume that the log is not buffered and each record is immediately written to the permanent memory

5.4.3 Undo and Redo algorithms

Recovery algorithms differ in the information they store in the log, in how they structure this information and, more importantly, in the time when the system transfers the pages updated by a transaction to the permanent memory. We say that a recovery algorithm requires an **undo** if an update of some uncommitted transaction is stored in the database. Should a transaction or a system failure occur, the recovery algorithm must undo the updates by copying the before-image of the page from the log to the database.



We say that a recovery algorithm requires **redo** if a transaction is committed before all of its updates are stored in the database. Should a system failure occur after the transaction commits but before the updates are stored in the database, the recovery algorithm must redo the updates by copying the after-image of the page from the log to the database.



A failure can happen also during the execution of a recovery procedure, and this requires the restart of the procedure. This means that for such procedures the *idempotency* property must hold. That is, even if the operation is executed multiple times the effect is the same as if it is executed once. For the assumption that the entire page is replaced, this property is automatically fulfilled.

5.4.4 Checkpoint

To reduce the work performed by a recovery procedure in the case of system failure, another information is written to the log, the so called **checkpoint** (CKP) event. There exists three methods of performing and recording checkpoints:

- **commit-consistent checkpoint**:

  1. the activation of new transactions is suspended
  2. the systems waits for the completion of all active transactions
  3. all modified pages present in the buffer (dirty pages) are written to the permanent memory and the relevant records are written to the log. Here the permanent memory is forced so that all the transactions terminated before the checkpoint have their updates
  4. the CKP record is written to the log
  5. a pointer to the CKP record is stored in a special file, called restart file.
  6. the system allows the activation of new transactions

  This strategy is simple but not efficient, because from (2) we derive that if we have long-time running transactions, we must wait a long time before the flushing of the dirty pages. This problem also affects the work that must be done in the case of restart. For this reason, we should perform this strategy when no transactions are running, but this condition is very demanding;

- **buffer-consistent checkpoint - Version 1**: in this case there's no waiting for active transactions termination, but the problem of the buffer flush operation (costly operation) is still present;

- **Fuzzy checkpoint (ARIES method)**

5.5 Recovery algorithms

The Recovery managers for the transactions management differ in the way they combine the undo and redo algorithms in order to recover the last consistent state of a DB. There are four possibilities: Undo-Redo, Undo-NoRedo, NoUndo-Redo, NoUndo-NoRedo. In the following sections we assume that a write in the log is forced to disk.

5.5.1 Use of the Undo algorithm

The use of the undo algorithm depends on the policy used to write pages updated by an active transactions to the DB:

- *deferred update* requires that updated pages cannot be written to the DB before the transaction has committed, and it is implemented by "pinning" the pages in the buffer until the end of the transaction. This policy is costly at the time of the commit, but it is more efficient from the point of view of the resource usage. An algorithm that adopts this policy is a *NoUndo* type: when a system failure occurs, no undo is necessary since the DB has not been changed.

- *immediate update* allows that updated pages can be written to the DB before the transaction has committed, and it is implemented by setting the page as "dirty" and by removing its pin. An algorithm that adopts this policy is a *Undo* type: if a system failure occurs, the updates of the DB must be undone.

To **undo** the updates of a transaction, the following rule must be observed:

Undo rule (Write ahead log)

If a database page is updated before the transaction has committed, its before-image must have been previously written to the log file in the permanent memory.

This rule allows an undo of a transaction in the case of abort by using the before-images from the log.

5.5.2 Use of the Redo algorithm

The use of the redo algorithm depends on the policy used to commit a transaction:

- *deferred commit* requires that all updated pages are written to the DB before the commit record has been written to the log. A transaction that implements this policy is a *NoRedo* type: when a system failure occurs, no redo of the updates is necessary. A downside of this policy is that the buffer manager is forced to flush to the permanent memory all the updated pages before the commit operation;

- *immediate commit* allows the commit record to be written to the log before all updated pages have been written to the DB. An algorithm that adopts this policy is a *Redo* type, since it is necessary to redo all the updates in the case of a system failure. On the other hand, the buffer manager is free to flush the unpinned pages to the permanent memory when it considers appropriate.

To **redo** the updates of a transaction, the following rule must be observed:

Redo rule (commit rule)

Before a transaction can commit, the after-images produced by the transaction must have been written to the log file in the permanent memory.

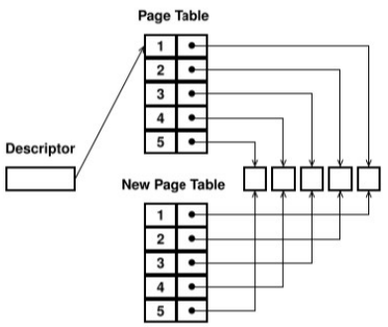
5.5.3 No use of Undo and Redo algorithms

- NoUndo algorithm requires that all the updates of a transaction must be in the database after the transaction has committed;
- NoRedo algorithm requires that all the updates of a transaction must be in the database before the transaction has committed

Therefore:

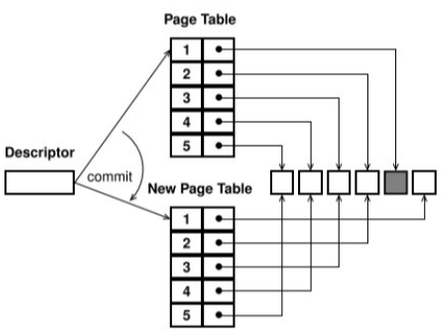
> A **NoUndo-NoRedo** algorithm requires that all the updates of a transaction must be in the database neither before nor after the transaction has committed.

This condition is preserved if the commit operation atomically writes all the updates of a transaction to the DB and mark the transactions as committed: this is feasible using the **shadow pages**.
The implementation of the shadow pages is based on the *Page table*, a permanent memory index that maps each page identifier to the physical address where the page is stored. Moreover, there is the *descriptor*, which is a record that contains a pointer to the Page table.



When a transaction starts, a copy of the Page table is created in the permanent memory (*New page table*), and it is used by the transaction. When a transaction updates for the first time a page:

1. a new database page is created, the *current* page with a certain address p, whereas the old page becomes a *shadow page*
2. the New Page Table is updated so that the first element contains the physical address p of the *current page*



When the transaction reaches the commit point, the system should substitute all the shadow pages with an atomic action, otherwise if a failure happen the database would be left in an incorrect state. This atomic action is implemented by executing the following steps:

1. the pages updated in the buffer are written to the permanent memory
2. the descriptor of the database is updated with an atomic operation

NOTE:

- no need for undo/redo, but this technique is difficult to be applied if the system manages concurrent transactions;
- if the number of abortion/rollbacks is not high, the undo-redo policy is the preferred one, since it is more efficient.

5.6 Recovery from system and media failure

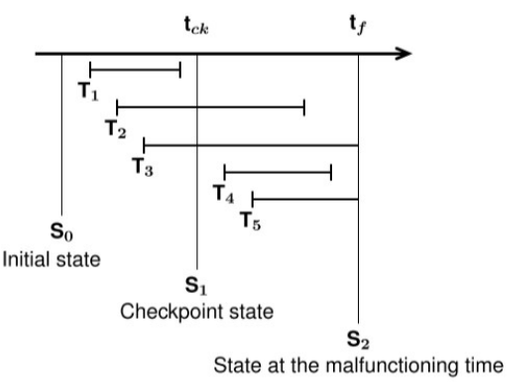
In the case of **system failures**, in order to recover the database, the *restart* operator is invoked to perform the following steps (*warm restart*):

- bring the database in its committed state with respect to the execution up to the to the system failure;
- restart the normal system operations.

And is described with two phases:

- in the **rollback** phase the log is read from the end to the beginning:
  - to undo, if necessary, the updates of the non terminated transactions;
  - to find the set of the identifiers of the transactions which are terminated successfully in order to redo their operations.
- in the **rollforward** phase the log is read onward from the first record after the checkpoint to redo all the operations of the terminated transaction

In this example, both T_3 and T_5 must be undone, while T_2 and T_4 must be redone.



rollback				
LSN	**Operation**	L_r	L_u	**Action**
16	(*commit*, 4)	$\{4\}$	$\{\}$	no action
15	$(W, 3, B, 20, 30)$	$\{4\}$	$\{3\}$	undo: $B = 20$
14	(*commit*, 2)	$\{4, 2\}$	$\{3\}$	no action
13	$(W, 5, E, 50, 30)$	$\{4, 2\}$	$\{3, 5\}$	undo: $E = 50$
12	(*begin*, 5)	$\{4, 2\}$	$\{3\}$	no action
11	$(W, 4, D, 5, 30)$	$\{4, 2\}$	$\{3\}$	no action
10	(*begin*, 4)	$\{4, 2\}$	$\{3\}$	no action
9	$(W, 2, C, 5, 10)$	$\{4, 2\}$	$\{3\}$	no action
8	(*CKP*, $\{2, 3\}$)	$\{4, 2\}$	$\{3\}$	no action
7	$(W, 3, A, 20, 30)$	$\{4, 2\}$	$\{3\}$	undo: $A = 20$
6	(*begin*, 3)	$\{4, 2\}$	$\{\}$	L_u is empty, end of phase

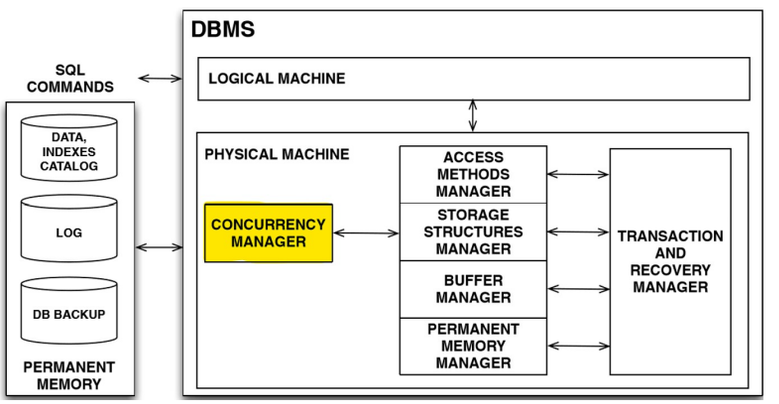
rollforward			
LSN	**Operation**	L_r	**Action**
9	$(W, 2, C, 5, 10)$	$\{4, 2\}$	redo: $C = 10$
10	(*begin*, 4)	$\{4, 2\}$	no action
11	$(W, 4, D, 5, 30)$	$\{4, 2\}$	redo: $D = 30$
12	(*begin*, 5)	$\{4, 2\}$	no action
13	$(W, 5, E, 50, 30)$	$\{4, 2\}$	no action
14	(*commit*, 2)	$\{4\}$	no action
15	$(W, 3, B, 20, 30)$	$\{4\}$	no action
16	(*commit*, 4)	$\{\}$	L_r is empty, end of phase

Picture 5.6 and 6.2 show the actions to be performed in the *rollback* and *rollforward* phases of the restart procedure: L_r indicates the set of transactions to be redone, while L_u indicates the set of transactions to be undone.
In the case of **media failure**, a *cold restart* is performed through the following steps:

- the most recent database backup is reloaded;
- a *rollback* phase is performed on the log;
- a *rollforward* phase is performed to update the copy of the database

Chapter 6

Concurrency management



When executing concurrent transactions, some interference may leave the DB in an inconsistent state. The *Concurrency Manager* is the system module that ensures the execution of concurrent transactions without interference during DB access.

6.1 Introduction

The problem about concurrent transactions is that their database operations are interleaved, i.e. operations from one program can execute in between operations from another program. This interleaving can cause programs to produce unpredictable results

Assuming that each transaction is *consistent*, a simple way to avoid interference among concurrent transactions is to allow only a **serial execution** of them, i.e. all the operations of a transaction are executed before any operation of the other. However, serial executions are impractical from a performance perspective, so we consider a **serializable execution**. An execution of a set of transactions is *serializable* if its effect is exactly the same as a serial execution of the committed transactions. Serializable executions are **correct**, because each serializable execution has the same effect of a serial one, and serial executions are correct.

In this sense, the goal of the *Concurrency Manager* is to establish an order among the operations of a set of transactions to make its execution serializable. Its correctness is proved by using the results of the **serializability theory**, which uses a structure called **history** (or **schedule**) to represent the chronological order in which the operations of a set of concurrent transactions are executed, and which goal is to define the properties that an history has to hold to be serializable.

6.2 Histories

From the DBMS's point of view, a transaction is seen as a set of read/write/commit and abort operations. For example, this program:

```
program T;
var x, y:integer;
begin
    x:= read(x);
    y:= read( y);
    x := x + y;
    Write(x);
end;
end {program}.
```

is seen as $r_1[x]$, $r_1[y]$, $w_1[x]$, c_1.
We make the following assumptions:

- a transaction is a set of read/write operations which terminates only with a commit or an abort;
- we do not consider the operations of insertion or deletion of records;
- a transaction reads/writes a specific record at most once.

We now introduce the concept of history:

> Let $T = \{T_1, T_2, ..., T_n\}$ a set of transaction. A **history** (or **schedule**) H on T is an ordered set of operations such that:
>
> 1. the operation of H are those of $T_1, T_2, ..., T_n$
> 2. H preserves the ordering between the operations belonging to the same transaction

Intuitively, the history H represents the actual or potential execution order of the operations of the transactions $T_1, T_2, ..., T_n$. An example of history involving three transactions could be the following one:

$$T_1 = r_1[x], w_1[x], w_1[y], c_1$$
$$T_2 = r_2[x], w_2[y], c_2$$
$$T_3 = r_3[x], w_3[x], c_3$$

$$H_1 = r_1[x], r_2[x], w_1[x], r_3[x], w_3[x], w_2[y], w_1[y], c_1, c_2, c_3$$

6.2.1 Equivalent histories

For **equivalent histories** we could mean that they produce the same effects on the database, namely that the values written to the database by the committed transactions are equal. Since a DBMS knows nothing of the computations made

by a transaction in temporary memory, a **weaker notion** of equivalence which takes into account only the order of operations in conflict made on the database is preferred.

> Two operations of different transactions are in **conflict** if they are on the same data item and at least one of them is a write operation.

This definition allows us to define three types of situations in which the operations are in conflict:

- Write-Read Conflict.
  Consider the following piece of a history: $H = ...w1[x], r2[x]....$ The transaction T_2 reads x updated by T_1 which has not yet committed. This type of read, called *dirty read*, can give rise to executions not serializable;

- Read-Write Conflict.
  Consider the following piece of a history: $H = ...r1[x], w2[x], r1[x], ...$ Transaction T_2 writes a data x previously read by the transaction T_1, still active, which then rereads it getting a different value even if in meanwhile T_1 has not updated it. This effect obviously can not be achieved by any serial execution of the two transactions;

- Write-Write Conflict.
  Consider the following piece of a history: $H = ...w1[x], w2[x]...$ The two transactions, while attempting to modify a data item x, both have read the item's old value before either of them writes the item's new value.

> Two histories H_1 and H_2 are **c-equivalent** (conflict-equivalent), i.e. equivalent with respect to operations in conflict, if:
>
> 1. they are defined on the same set of transactions $T = \{T_1, T_2, ..., T_n\}$ and have the same operations;
>
> 2. they have the same order of operations in conflict of transactions terminated normally

Condition (1) requires that H_1 and H_2 contain the same set of operations to be comparable, while condition (2) requires that H_1 and H_2 have the same order of the operations in conflict of committed transactions

6.3 Serializable history

A history H on the set $T = \{T_1, T_2, ..., T_n\}$ is **serial** if it represents a serial execution of T in some order.
We can now define a stronger condition that is sufficient to ensure that a history is serializable:

> A history H on the transactions $T = \{T_1, T_2, ..., T_n\}$ is **c-serializable** if it is c-equivalent to a serial history on $T_1, T_2, ..., T_n$.

NOTE that each c-serializable history is serializable, but there are serializable histories that are not c-serializable.

Serialization graph

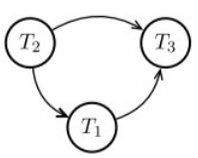
Another way of examining a history H and decide whether it is c-serializable or not is to analyze a particulare graph derived from H, called the **serialization graph**.

Let H be a history of committed transactions $T = \{T_1, T_2, ..., T_n\}$. The **serialization graph** of H, denoted $SG(H)$, is a directed graph such as:

- there is a node for every committed transaction in H;
- there is a directed arc from $T_i \rightarrow T_j$, with $(i \neq j)$ if and only if in H some operation p_i in T_i appears before and conflicts with some operation p_j in T_j.

We say that two transactions T_i and T_j conflicts if $T_i \rightarrow T_j$ appears in $SG(H)$. As an example, this is the serialization graph of the given history.

T_1	T_2	T_3
	$r_2[x]$	
	$w_2[y]$	
	c_2	
$r_1[x]$		
$w_1[x]$		
$w_1[y]$		
c_1		
		$r_3[x]$
		$w_3[x]$
		c_3



C-Serializability theorem

A history H is c-serializable if and only if its serialization graph $GS(H)$ is acyclic.

Since Picture 6.6 represents an acyclic graph, the history in Picture 6.3 is c-serializable!

6.4 Serializability with locking

From the analysis of the serialization graph it can be verified a posteriori if a history is c-serializable. Histories and serialization graphs, however, are abstract concepts, and during the execution of a set of transactions, the **serialization**

graph is not constructed. Conversely, the c-serializability theorem is used to prove that the **scheduling algorithm for the concurrency control** used by a scheduler is correct, i.e. that all histories representing executions that could be produced by it are c-serializable.

6.4.1 Strict two-phase locking

A simple scheduling algorithm to obtain serializability is the *Strict 2PL* protocol. The idea behind this approach is simple: each data item used by a transaction has a **lock** associated with it, which could be:

- a *read* or *shared* (S) lock;
- a *write* or *exclusive* (X) lock.

, and it follows two rules:

1. if a transaction T_i wants to read a data item, it first request a shared lock on the data item (first phase):
   - if no other transaction holds the lock, then the data item is locked;
   - if another transaction T_j holds a lock in conflict, then T_i must wait until T_j releases it
2. All locks held by a transaction T_i are released together when T_i commits or aborts (second phase). To guarantee the isolation of a transaction, lock releases cannot occur before commit/abortion time.

We can describe the *lock-granting policies* of Strict 2PL protocol using a **compatibility matrix**:

- a row corresponds to a lock that is already held on an element x by another transaction;
- the columns correspond to the mode of lock on x that is requested

	S	X
S	Yes	No
X	No	No

NOTE:

- requests to acquire or release locks are automatically inserted into transactions by the *Transaction Manager*;
- lock and unlock requests are handled by the scheduler with the use of the *Lock Table*, in which an entry contains the ID of the data item being blocked, the type of lock granted or requested, a list of transactions holding lock and a queue of lock requests.

C-Serializability of Strict 2PL

A *Strict 2PL* protocol ensures c-serializability.

However, the set of *Strict 2PL* histories is a subset of the c-serializable histories, i.e. there are c-serializable histories that are not *Strict 2PL*.

6.4.2 Deadlocks

Although being simple, *Strict 2PL* needs a strategy to detect *deadlocks*, a situation in which, for example, two transactions lock two different items and none of them can proceed. The deadlock problem can be solved with deadlock prevention techniques or with deadlock detection and recovery technques.

Deadlock detection

- a strategy to detect deadlocks uses a **wait-for graph**, in which :
  - the nodes are active transactions;
  - an arc from T_i to T_j indicates that T_i is waiting for a resource held by T_j.

  Using this representation, a **cycle** in the graph indicates that a **deadlock** has occurred, and one of the transactions of the cycle must abort;
- the standard method to decide which transaction to abort is to choose the **"youngest"** transaction by some metric (being young, it has probably done less work, so it is less costly to abort);
- checking for cycles requires a linear complexity algorithm, but the actual cost of the management of the graph in real cases discourage the use. For this reason, instead of wait-for graph we can use the **timeout strategy**: if a transaction has been waiting too long for a lock, then the scheduler simply presumes that deadlock has occurred and it aborts the transaction.

Deadlock prevention

- each transaction T_i receives a timestamp $t_s(T_i)$ when it starts: T_i is older than T_j if $t_s(T_i) < t_s(T_j)$;
- each transaction is assigned a priority on the basis of its timestamp: the **older** a transaction is, the **higher priority** it has.
- when a transaction T_i requests a lock on a data item that conflicts with the lock currently held by another active transaction T_j, two algorithms are possible:
  1. wait-die (or non-preemptive technique): an older transaction waits only for a younger one, otherwise the younger dies.
  2. wound-wait (or preemptive technique): an older transaction wounds a younger one to take its lock, otherwise the younger waits for the older one.

  In both cases when an aborted transaction T_i is restarted, it has the same priority it had originally: this property ensures that sooner or later every transaction will become the oldest, so it ensures that no *starvation* will occur.

  These two algorithms have the property of **not creating deadlocks**, but their behaviour is very different:

  - with the wait-die method, a transaction T can wait for data locked by a younger transaction or it restarts due to an older one T_0. Most likely is the second case and then the method favors the restarting rather than waiting;
  - with the wound-wait method, a transaction T can wait for data locked by an older transaction or restarts a younger one T_y. Most likely is the first case followed by the waiting rather than restarting.

- deadlock prevention methods are easier to implement than deadlock detection solutions, but on the other hand they lead to the abortion of transactions which could run without restarts.

6.5 Serializability without locking

While methods of concurrency control that use locks are called *pessimistic*, in the sense that they're based on the idea that a bad things is likely to happen, other methods, called *optimistic*, do not lock data because are based on the idea that bad things are not likely to happen. However, in this approach, when a transaction requests to commit, the system controls that no bad things has happened.

Snapshot isolation

- a transaction can perform any database operation without requesting permission, and taking advantage of *multiversions* of each data item;
- the transactions must request permission to commit;
- each transaction T_i reads the data of the DB version (*snapshot*) produced by all the transactions committed before T_i starts, but no effects are seen of other concurrent transactions.
- this approach grants that no concurrent writes are performed ("First-Committer-Wins" rule).

6.6 Multiple granularity locking

The concurrency control techniques seen so far, based on the idea of a single record lock, is not sufficiently general to treat transactions that operate on collections of records. For example, in some cases it could be convenient to lock an entire table, rather that a single record.
For this reason, other techniques are base on the idea that the data to lock can have different **granularities**, and among them is defined an inclusion relationship : DB $\rightarrow$ Files $\rightarrow$ Pages $\rightarrow$ Records $\rightarrow$ Fields.
The inclusion relation between data can be thought of as a **tree** of objects where each node contains all its children. If a transaction gets an *explicit* S or X lock on a node, then it has an *implicit* lock in the same lock mode on all the descendants of that node.
In order to manage locks on data of different granularities, *intention locks* are introduced:

Multiple-granularity and intention locks

Multiple-granularity locking requires that before a node in explicitly locked, a transaction must first have a proper intention lock on all the ancestors of that node in the granularity hierarchy.

The intention locks are the following:

- IS, or *intention shared lock*, allows requestor to explicitly lock descendant nodes in S or IS mode;
- IX or *intention exclusive lock*, allows requestor to explicitly lock descendants in S, IS, X, IX or SIX mode;
- SIX or *shared intentional exclusive lock*, implicitly locks all descendants of node in S mode and allows requestor to explicitly lock descendant nodes in X, SIX, or IX mode.

Note that the SIX lock is introduced in order to combine S and IX to simplify the lock manager, since they frequently appera together.
The compatibility matrix is:

	S	X	IS	IX	SIX
S	Yes	No	Yes	No	No
X	No	No	No	No	No
IS	Yes	No	Yes	Yes	Yes
IX	No	No	Yes	Yes	No
SIX	No	No	Yes	No	No

To deal with multiple granularity locks, it is necessary extend the *Strict 2PL protocol* with new rules, obtaining the protocol called **Multi-granularity Strict 2PL**:

1. a node can be locked by a transaction Ti in S or IS mode only if the parent is locked by Ti in IS or IX mode.
2. a node can be locked by a transaction Ti in X, IX or SIX mode only if the parent is locked by Ti in SIX or IX mode.

6.7 Locking for dynamic databases

So far we have considered only transactions that read or update existing records in the database. In reality, transactions can also **insert** or **delete** records into tables with **indexes**. These possibilities raise new problems to be solved to ensure serializability during the concurrent execution of a set of transactions, while continuing to adopt the Strict 2PL protocol.

Insertion and Deletion

The inserted or deleted records are called *phantoms* because they are records that appear or disappear from sets, i.e. they are invisible only during a part of a transaction execution.

An acceptable practical solution to this problem is **index locking**. When a set of records that match some predicate is locked, the database system also checks to see if there is an index whose key matches the predicate. If such an index exists, the structure of the index should allow us to easily lock all the pages in which new tuples that match the predicate appear or will appear in the future. The practical issue of how to locate the relevant pages in the index that need to be locked and what locks need to be acquired is discussed in the following section.

Concurrency Control in B+−trees

The concurrent use of a B+−tree index by several transactions may be treated in a simple way with *Strict 2PL protocol*, by considering each node as a granule to lock appropriately. This solution, however, would lead to a low level of concurrency due to locks on the first tree levels.
A better solution is obtained by exploiting the fact that the indexes are used in a particular way during the operation. In the case of a search, the nodes visited to reach the leaves, where the data is located, are locked in reading during the visit and unlocked as soon as the search proceeds from one node to the child. In the case of an insertion, during the visit of the tree when switching from one node A to a child B not full, the locks on A can be released because a possible propagation of the effect of the insert into a leaf stops at node B, called a (safe node). A node is safe for a delete operation if it is at least half full. The general case with node splits, merging and balancing is more complex and several tree locking techniques have been proposed.

Chapter 7

Distributed concurrency control

Distributed concurrency control ensures the correct execution of operations (more generally, transactions) that affect data that are stored in a distributed fashion on different database servers. In case of data replication, a typical application of a concurrency control protocol is synchronizing all replicas of a record (on all database servers that hold a replica) when a write on the record was issued to one of the database servers. That is, if there a n replicas and one of them has been updated the remaining $n - 1$ replicas have to be updated, too.
Specifications of concurrency control protocols use the term **agent** for each stakeholder participating in the protocol; each agent can furthermore have different **roles**, where the most important one is the **coordinator**. The coordinator is the database server that communicates with all other agents and is responsible for either **leading** the distributed operation to success or **aborting** it in its entirety. **Concurrency protocols** hence offer a solution to the **consensus problem**: the consensus problem requires a set of agents to agree on a single value.

7.1 Two-Phase Commit protocol

The **two-phase commit** requires **all** participating agents to agree to a proposed value in order to accept the value as the currently globally valid state among all agents