Databases Technologies:

- Physical Organization and
- Query Management

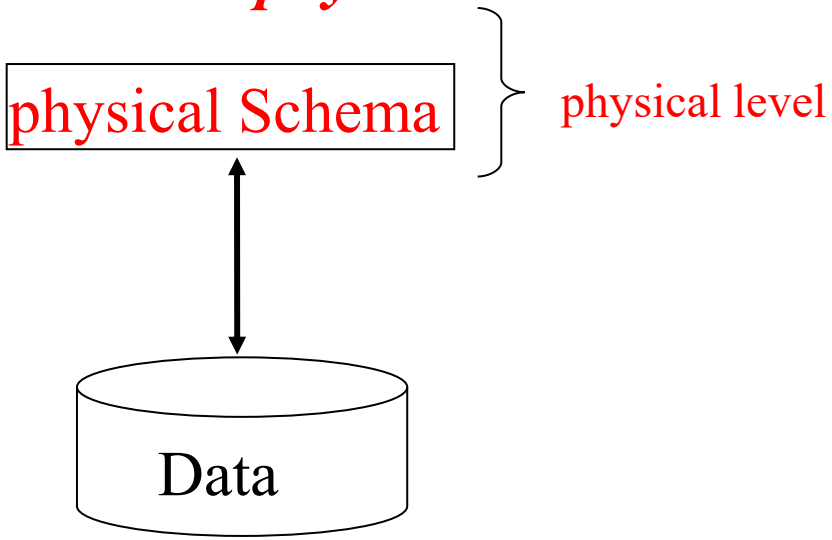
BD Technology: Why studying it?

- The DBMS offer their services in a "transparent" way:
  - We've so far been able to ignore many aspects of DB implementation
  - we considered the DBMS as a "black box"
- Why open it?
  - understanding how it works can be useful for a better and more efficient/effective use
  - some services are offered separately

Abstraction levels in DBMS

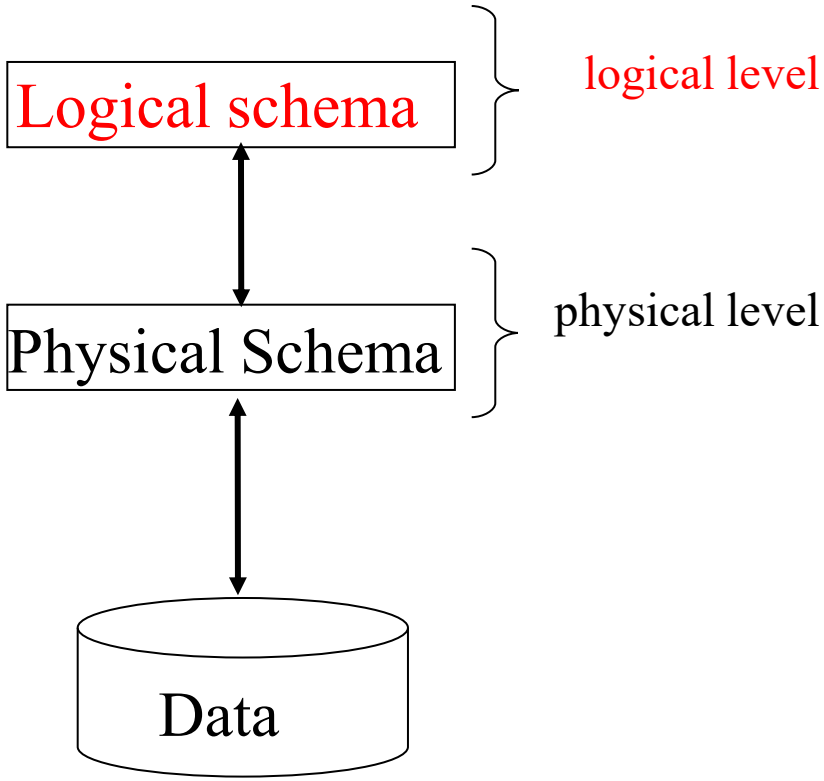
In DBMSs the database is divided into three distinct levels of data description, for each of which there is a logical schema (not conceptual).

- ***physical level:*** At this level it is described how data are physically organized into permanent memories and which auxiliary data structures are defined for making their use easier. The description of these aspects is called ***physical schema*** or ***internal schema.***



Abstraction levels in DBMS

- ***Logical Level:*** At this level it is described the structure of the data sets and the relationships between them, according to a certain data model, without any reference to their physical organization in the permanent memory. The description of the data base structure is called ***logical schema***.

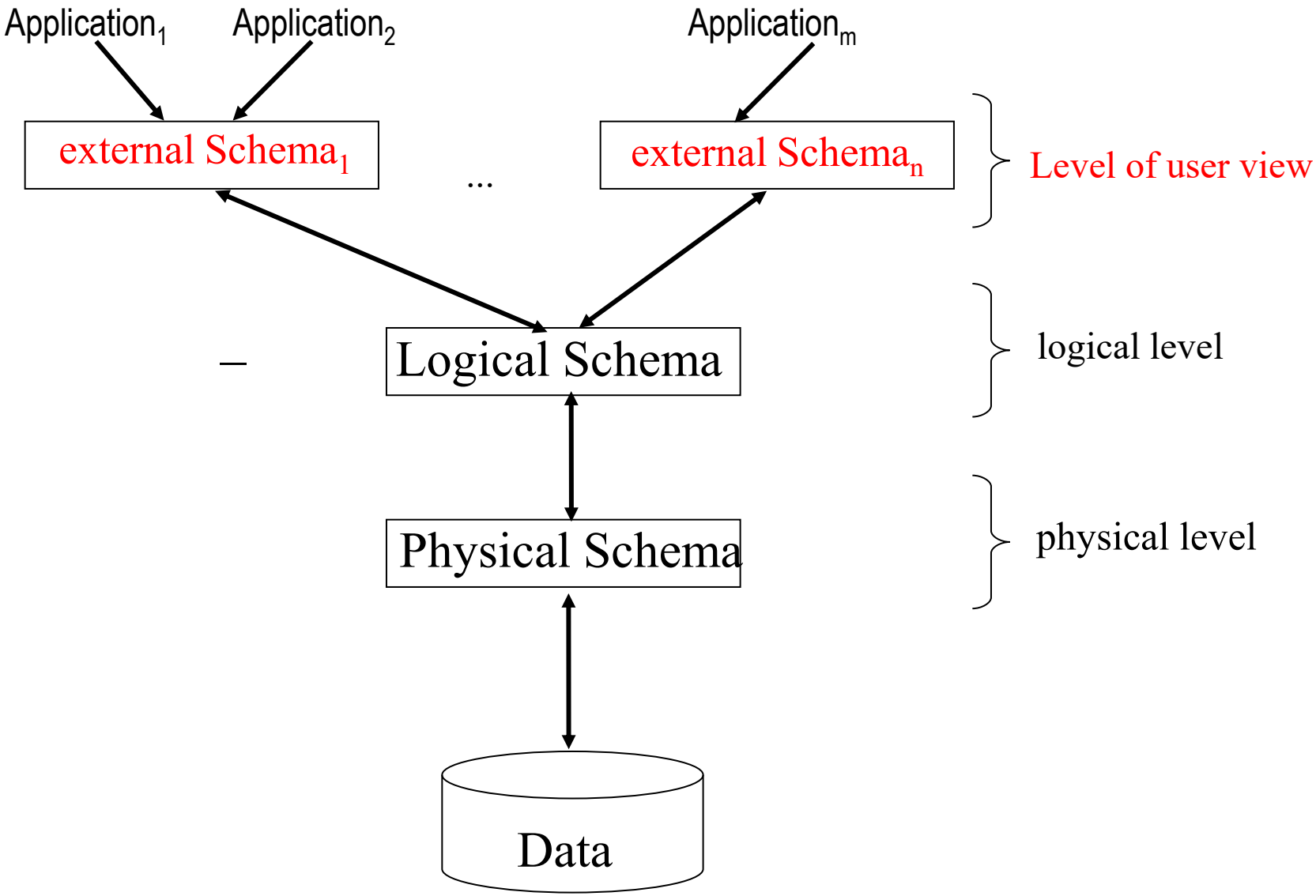


Abstraction levels in DBMS

- ***Level of logical view.*** At this level it is defined how the data base structure should appear for a certain application. This description is often called ***external schema*** (or *user view*), to highlight the fact that it refers to what a user imagines is the database.

While the logical schema is unique, there are typically many external schemas, one for each application (or group of related applications).

Abstraction levels in DBMS



Abstraction levels in DBMS

This architecture guarantees ***data independence***.

- ***Physical Independence***: we can change the physical structures (eg, methods of organization of the files managed by the DBMS or the physical location of the files), without affecting the high-level descriptions of the data and then programs that use the data.
- ***Logical Independence***: It allows us to interact with the external level of the database, independently of the logical level (eg, add a new external schema without changing the logical schema).

Abstraction levels in DBMS

Example.

Consider a database for managing information on the faculty of a university, manage the office of salaries and activities of the library.

Level of user view:

Office salaries: full name, social security number, parameter, salary.

Library: full name, telephone number.

Abstraction levels in DBMS

Logical level

Data on teachers are described by a unique set of records.

The logical schema of a relational database is declared as follows:

```
CREATE TABLE Staff
(    Name          CHAR (30),
     SSN      CHAR (15),
     Salary        INTEGER,
     Parameter     CHAR (6),
     Contact       CHAR (8))
```

Abstraction levels in DBMS

To create the two logical views, we will write:

```
CREATE VIEW  StaffForSalaries AS
SELECT       Name, SSN, Salary, Parameter
     FROM    Staff
CREATE VIEW  StaffForLibrary AS
     SELECT  Name, Contact
     FROM    Staff
```

Abstraction levels in DBMS

Physical level

The DB designer can choose to store the data in a sequential manner, or by using a hashing technique as the key name and surname:

MODIFY Staff TO HASH ON Name

Abstraction levels in DBMS

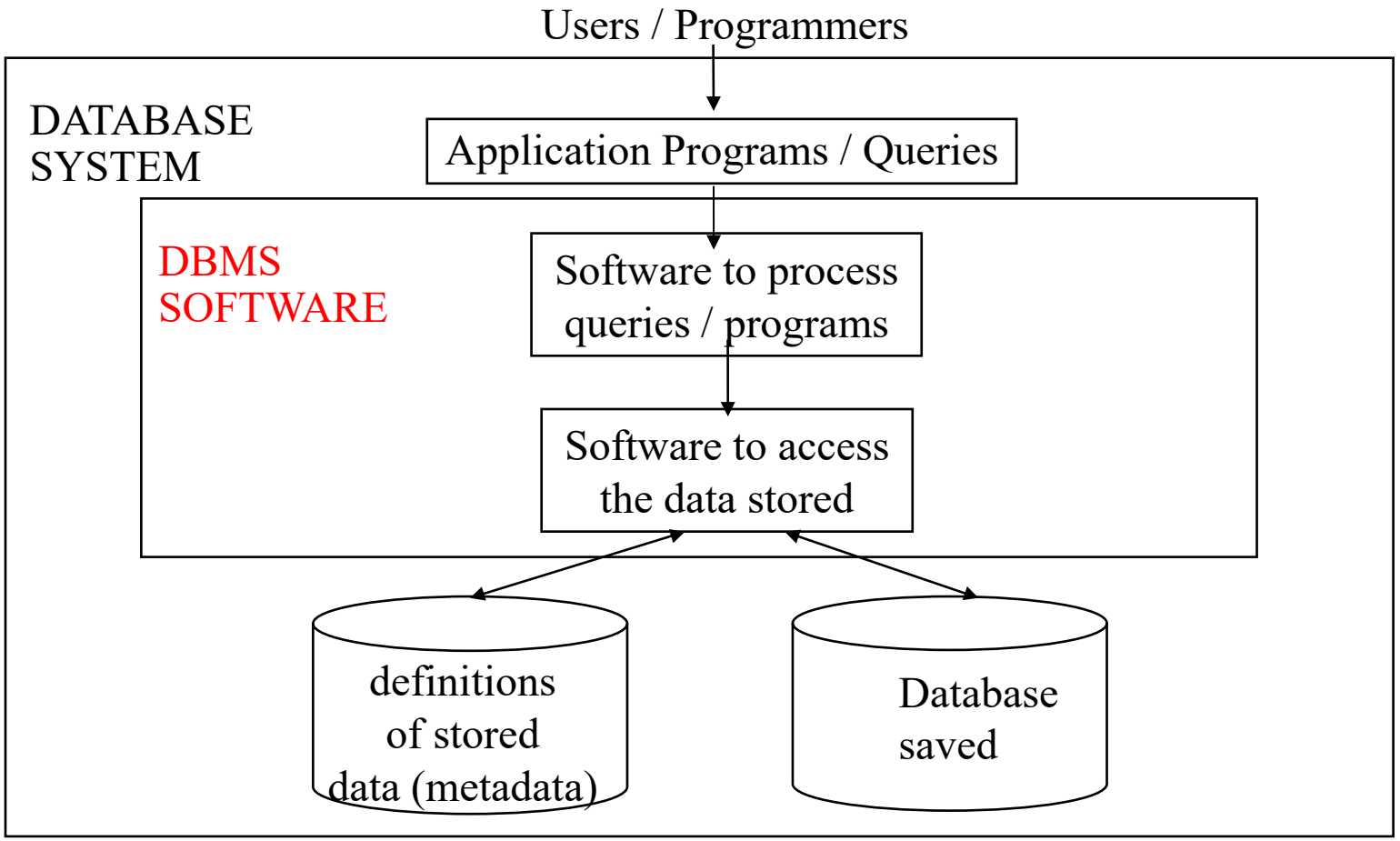
Later, it turns out that the salaries office performs frequently the data recovery operation on the teachers who have a certain parameter. We might think to change the physical schema by adding an index on the parameter:

```
CREATE INDEX ParameterIDX ON Staff(parameter)
```

This guarantees the independence of applications from the physical organization of the data. We avoid to change the programs that make use of staff data.

The component modules of a DBMS

A DBMS is a complex software system.



Addison Wesley Longman, Inc, 2000, Elmasri / Navathe, Fundamentals of Database Systems, Fourth Edition.

The component modules of a DBMS

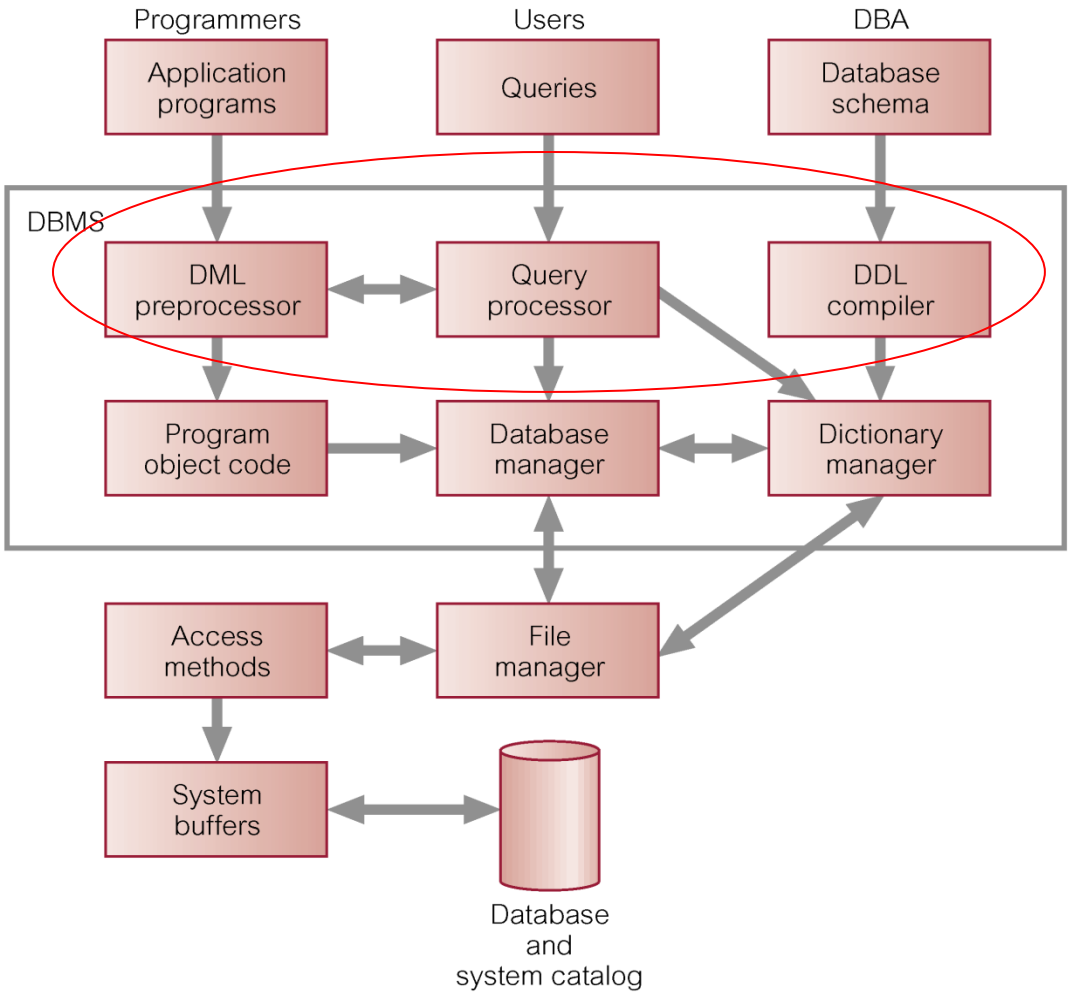
- The software to process queries and programs include:
  - a **DDL compiler** that processes the schema definitions and stores the descriptions of the schemes (the metadata) in the catalog of the DBMS.
  - a **Query compiler** that handles high-level queries that are entered interactively. It scans, analyzes and compiles or interprets a query creating access code to the database and then generates calls to the execution processor for executing the code.

The component modules of a DBMS

- a **precompiler** that extracts DML commands from application programs written in the host language and sends them to ***DML compiler***. This produces object code that will be linked with the rest of the compiled program. The result is an interface for parametric users which will include calls to the running processor the database.

The component modules of a DBMS

A more detailed view of the components



The component modules of a DBMS

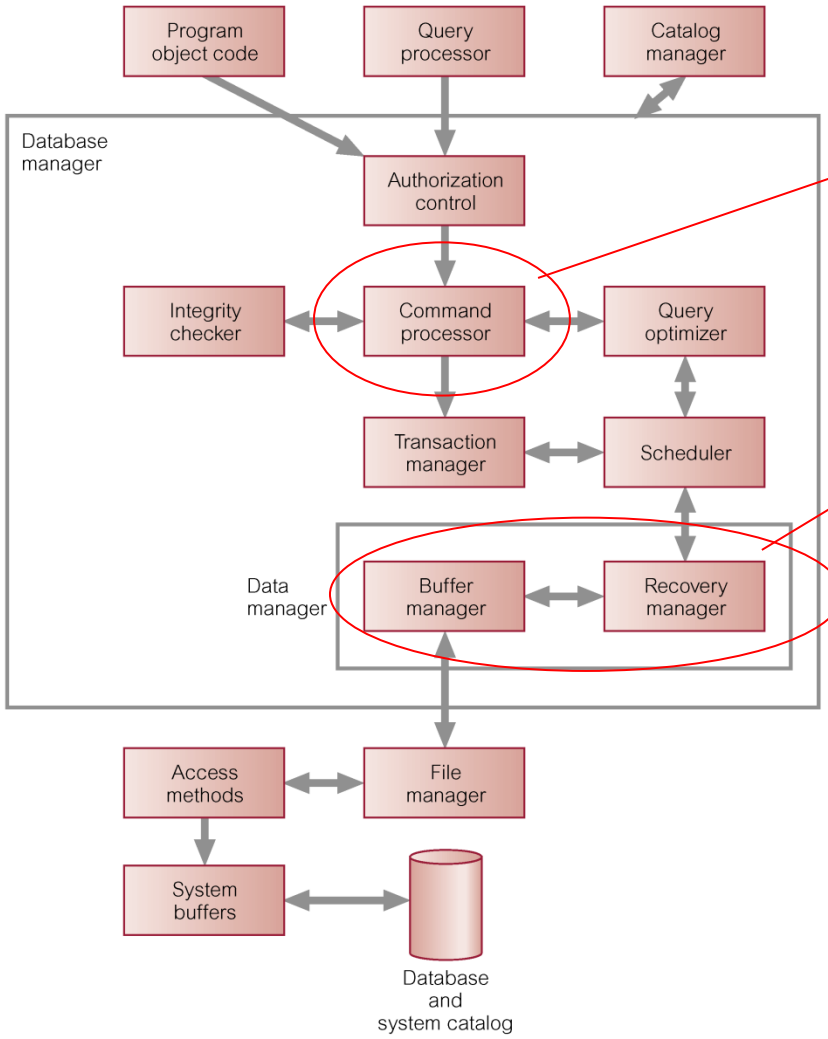
- The database and catalog (metadata) are usually stored on disk.
- The disk access is mainly controlled by the operating system, which schedules the I / O from disk.
- The software to access the stored data (*database manager*) consists essentially of:
  - a **stored data manager**, which controls the access to the information of the DBMS stored on the disk, either it is in the DB or in the catalog. It can make use of the operating system functions to access the disk, but additionally performs buffer management functions.

The component modules of a DBMS

- an ***executing processor of the data base*** that manages access to the DB at run-time. It receives requests for retrieval and update and runs them. The disk access is always controlled by the manager of the stored data. Calls to the processor come from the query compiler, which analyzes and translates the queries.

The component modules of a DBMS

A more detailed view of the "database manager"



Execution Processor of the Database

Manager of stored data

Architecture of a DBMS: Reliability

The DBMS should have mechanisms to protect data from hardware or software failures and unwanted interference from concurrent access to data by multiple users.

Example:

If two users / applications have both access to the personal data of a person, it is possible that while one is changing the address of residence (Street, Number, Zip, City), the other is reading the same address. The risk is that the read address is inconsistent, since it is made in part from the old one and in part from the new.

Architecture of a DBMS: Reliability

To this end, a DBMS guarantees that the interactions with the database are carried out by means of *transactions*.

A ***transaction*** is a sequence of read and write actions on the database and data in temporary memory, ensuring that the DBMS performs the following properties (***ACID*** *properties*):

Architecture of a DBMS: Reliability

Atomicity: A transaction is performed entirety (*committed transaction*) or not performed at all. The transactions that terminate prematurely (*aborted transaction*) are treated as if they had never started.

Consistency preservation: A correct execution of the transaction must bring the DB from one consistent state to another (the integrity constraints must be respected).

Isolation A transaction should not make updates visible to other transactions until it ends normally.

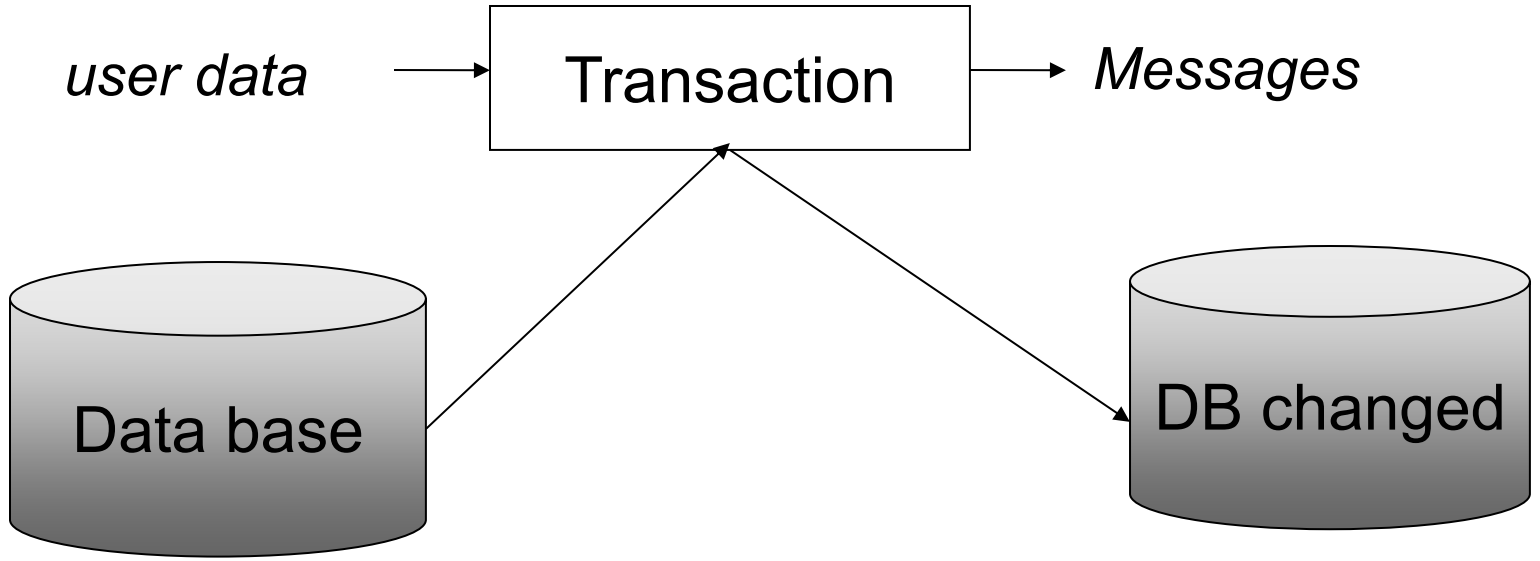
Durability (*persistence*): The changes on the DB of a completed transaction (normally completed) are permanent, i.e. they are not alterable by any malfunctions subsequent to termination.

Architecture of a DBMS: Reliability

A transaction can be expressed as:

- A set of expressions in a DML.
- A part of a sequential program.

In both cases there is some mechanism to indicate to the system the start point and end point of a transaction.



Architecture of a DBMS: Reliability

A ***failure*** is an event that brings the DB in an inconsistent state. We distinguish three types:

- ***Transaction failure.*** Are interrupted transactions that do not involve loss of data neither in temporary storage (buffer) nor in permanent memory. They are due to conditions already expected. Examples are the violation of integrity constraints and attempts to access to protected data.
- ***System Failures.*** The DBMS stops to work and there is loss of information contained in the temporary memory but not in permanent memory. They are due to hardware / software malfunctioning.
- ***Disasters*** (*disk / media failure*): Damage the permanent memory containing the DB.

Architecture of a DBMS: Reliability

The instantaneous interruption of a transaction or of the entire system activates appropriate procedures that allow to return the previous correct state of the data, that is, prior to the manifestation of the malfunction (***recovery procedures***).

To perform these procedures, a DBMS maintains a backup of the Database and keeps track of all changes made on the DB from the moment of the last backup.

Thanks to these ausiliary data, when there is a malfunction of the DBMS can rebuild a correct version of the data using the latest copy and rerunning all the operations that have changed the data of which it has kept track.

Architecture of a DBMS: Reliability

In order to ensure the reliability, a DBMS must also *manage the concurrent access*, so to avoid the loss of changes.

Example: Reading and simultaneous updating of a balance.

A simple way to solve the problem would be to run only isolated transactions, i.e. in such a way that, for every pair of transactions T_i and T_j, all the actions of T_i precede those of T_j (*serial execution*).

The DBMS have much more sophisticated mechanisms to manage concurrency.

Architecture of a DBMS: Security

The DBMS provide mechanisms similar to those found in operating systems to check that the data accessing only authorized persons.

- Identification of authorized users.
- theft protection through encryption.

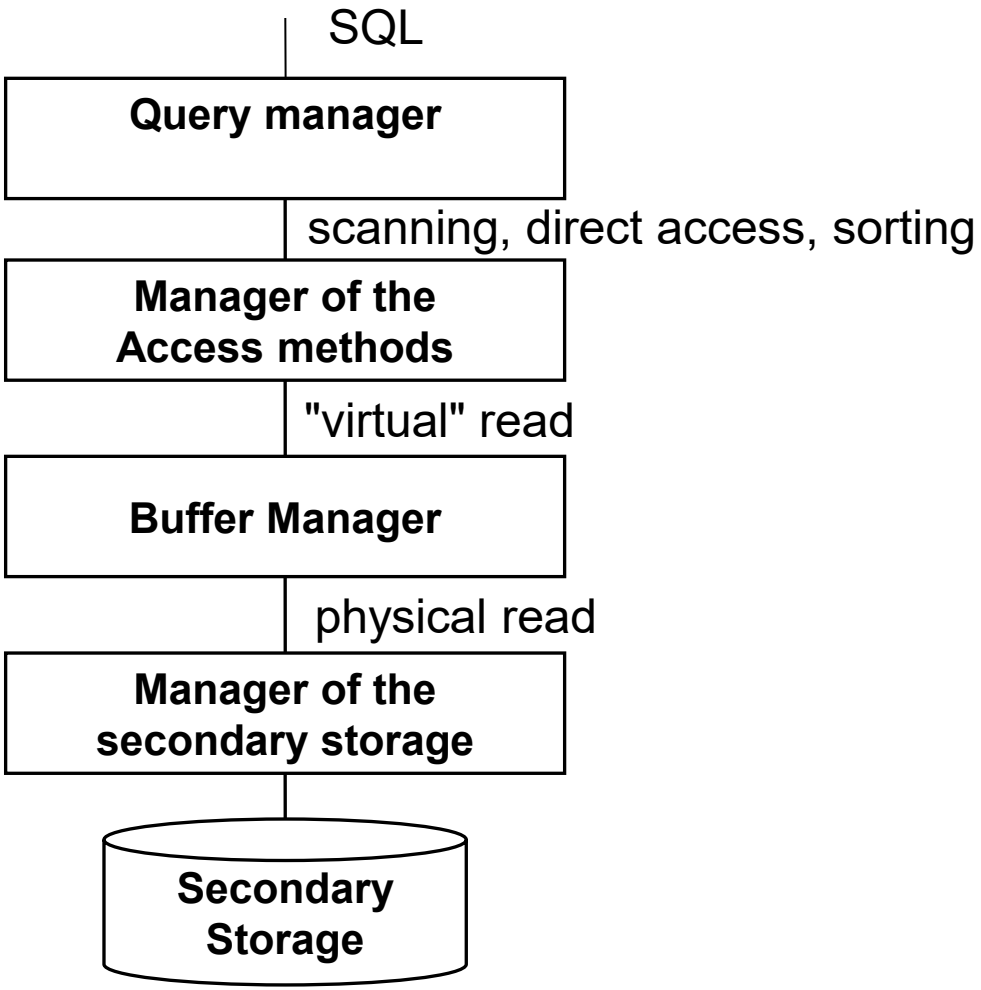
Also they allow us to specify the *restrictions*.

- Restrictions on data readable and editable (GRANT).
- Restrictions on hours of access to the DB.
- Access only to aggregate information (statistics), without the ability to inspect individual data.

Architecture of a DBMS: The DB as a shared resource

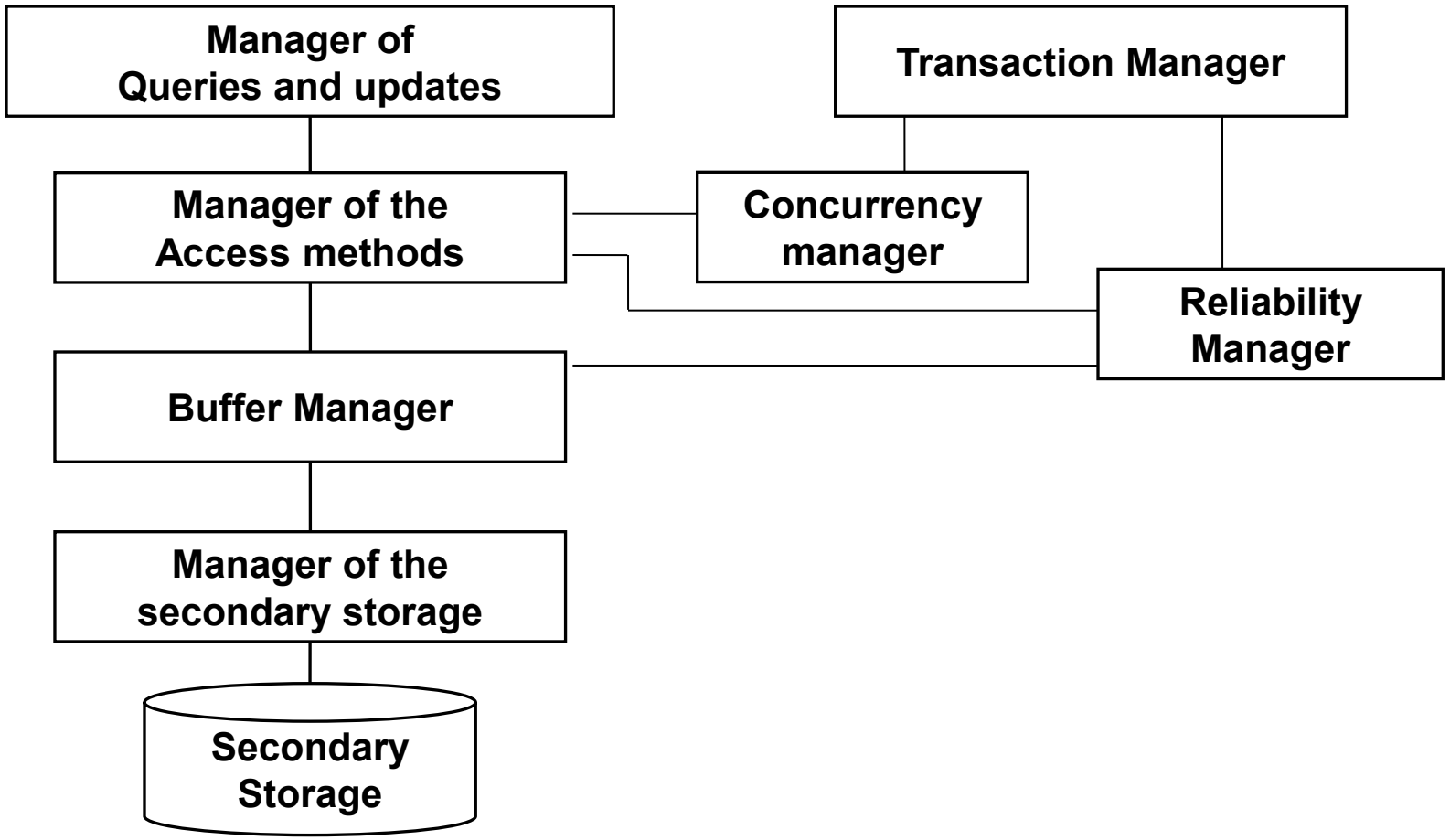
- In everything is complicated by the fact that a database is a resource **integrated**, **shared** between the various applications
- Consequences:
  - several activities on partially-shared data:
    - Mechanisms of authorization
  - multi-user activities on the shared data:
    - control of **concurrency**

Access and Query management



Access and Query Manager

Manager transaction



Technology of databases, topics

- Secondary memory and buffer management
- Organizing physical data
- Management ("optimization") of queries
- Control of reliability
- Concurrency Control
- Distributed architectures

Primary and secondary memory

- Programs can only refer to data in main memory
- The databases must be (substantially) in a secondary memory for two reasons:
  - dimensions
  - persistence
- Data in the secondary storage can be used only if they are first moved into main memory (this explains the terms "primary" and "secondary")

Primary and secondary memory

- The secondary storage devices are organized in blocks of (usually) fixed length (Order of magnitude: a few KB)
- The only operations on the devices are reading and writing of a page, i.e. a block of data (i.e. a string of bytes);
- for convenience we consider block and page synonyms

primary and secondary memory

- Access to secondary storage:
  - time to head positioning (10-50ms)
  - time to latency (5-10ms)
  - time to transfer (1-2ms)
- on average not less than 10 ms
- The cost of an access to secondary memory is four or more orders of magnitude higher than that for operations in main memory
- Therefore, in applications "I / O bound"(i.e. with many secondary memory accesses and relatively few operations) the cost depends exclusively on the number of secondary memory accesses
- In addition, access to "neighbors" blocks are cheaper (contiguity)

Buffer management

- **Buffer**:
- The buffer is an area of the main memory which consists of frames that may contain pages of data from the disk.
- The size of a frame is equivalent to that of the pages.
- Since the contents of the disk is often much greater than that which may contain the main memory (especially the buffer), we can keep in main memory only a small subset of pages.

Buffer management

- Therefore we will have to select pages to maintain in memory and those to remove from the buffer according to certain criteria, if we want to keep the system efficient.
- The buffer manager handles the buffer.

The purpose of the buffer management

- Reduce the number of accesses to secondary storage
  - In the case of reading, if the page is already in the buffer, the DBMS does not have access to secondary memory
  - In the case of writing, the buffer manager may decide to delay the physical write (if that is compatible with the reliability management - we will see later)

The purpose of the buffer management

- The management of the buffer and the difference between primary and secondary memory costs may suggest innovative algorithms.
- Example:
  - Table of 10,000,000 records, 100 bytes each (1GB)
  - 4KB blocks
  - Buffer available 20M
  - How can we do the sorting?
  - Merge-sort "multi-way"

Data managed by the buffer manager

- A directory which, for every page, maintains: (simplified version)
  - the physical file and the block number
  - two state variables:
    - a counter that shows how many programs use the page
    - a bit indicating whether the page is "dirty", that is, if it has been changed

The buffer manager functions

- Functions:
  - It receives read and write requests (of pages)
    - It executes them accessing to secondary memory only when necessary and, instead, using the buffer whenever possible
- The policies are similar to those implemented for memory management by operating systems; Principles:
  - "Data locality": the likelihood of having to re-use the data currently in use is relatively high
  - "Law 80-20" 80% of operations always use the same 20% of the data

The buffer manager interface

- ***fix***: performed by the transaction and regards the request of a page; requires a read from the disk if the page is not in the buffer (increments the counter associated with the page)
- ***setDirty***: informs the buffer manager that the page has been modified
- ***unfix***: indicates that the transaction has completed the use of the page (decrements the counter associated with the page)
- ***force***: Synchronously moves the page in a secondary memory (upon request of relibility manager, when data must not be lost)

Performing *fix*

- Search page in the buffer;
  - if the page is present, returns the address
  - otherwise, look for a free page in the buffer (counter to zero);
    - if found, if the page has been modified, performs a ***write operation.*** After, it reads the page to load, it calculates and returns the address.
    - otherwise, two alternative policies
      - "**steal**": selection of a" victim", occupied page of the buffer; The data of the victim are written to secondary memory; it reads the page of interest from the secondary memory and returns the address
      - "**no-steal**": the operation is put on hold
  - In any case, when accessing to the page, its counter is incremented.

Details

- The buffer manager writes records in two different contexts:
  - **Synchronous** when it is explicitly requested with a force
  - **Asynchronous** when the buffer manager deems appropriate (or necessary). In particular, the buffer manager may decide to anticipate or delay writing operations of pages made free by means of *unfix* that have been modified (p*re-flushing*).
- Note that a frequently used page may remain a long time in memory and can undergo several changes during this time period. The writing will occur only once.

DBMS and file system

- The file system is the operating system component that handles the secondary storage
- The DBMS uses its functionalities, but to a limited extent, to create and delete files and, if necessary, to read and write individual blocks or sequences of contiguous blocks.
- The file organization, both in terms of the distribution of records in the blocks and in terms of the structure within individual blocks, is managed directly by the DBMS which creates its own "abstraction" of the file.

DBMS and file system

- The DBMS manages the file blocks allocated as if they were one large secondary storage space and builds, in such a space, the physical structures with which implements the tables.
- The DBMS sometimes creates large files to store different tables (to the limit, the entire database)
- Sometimes, files are created at different times:
  - it is possible for a file to contain data of several tables and that different tuples of the same table are stored in different files.
- Often, but not always, each block is dedicated to the tuples of a single table

Blocks and tuples

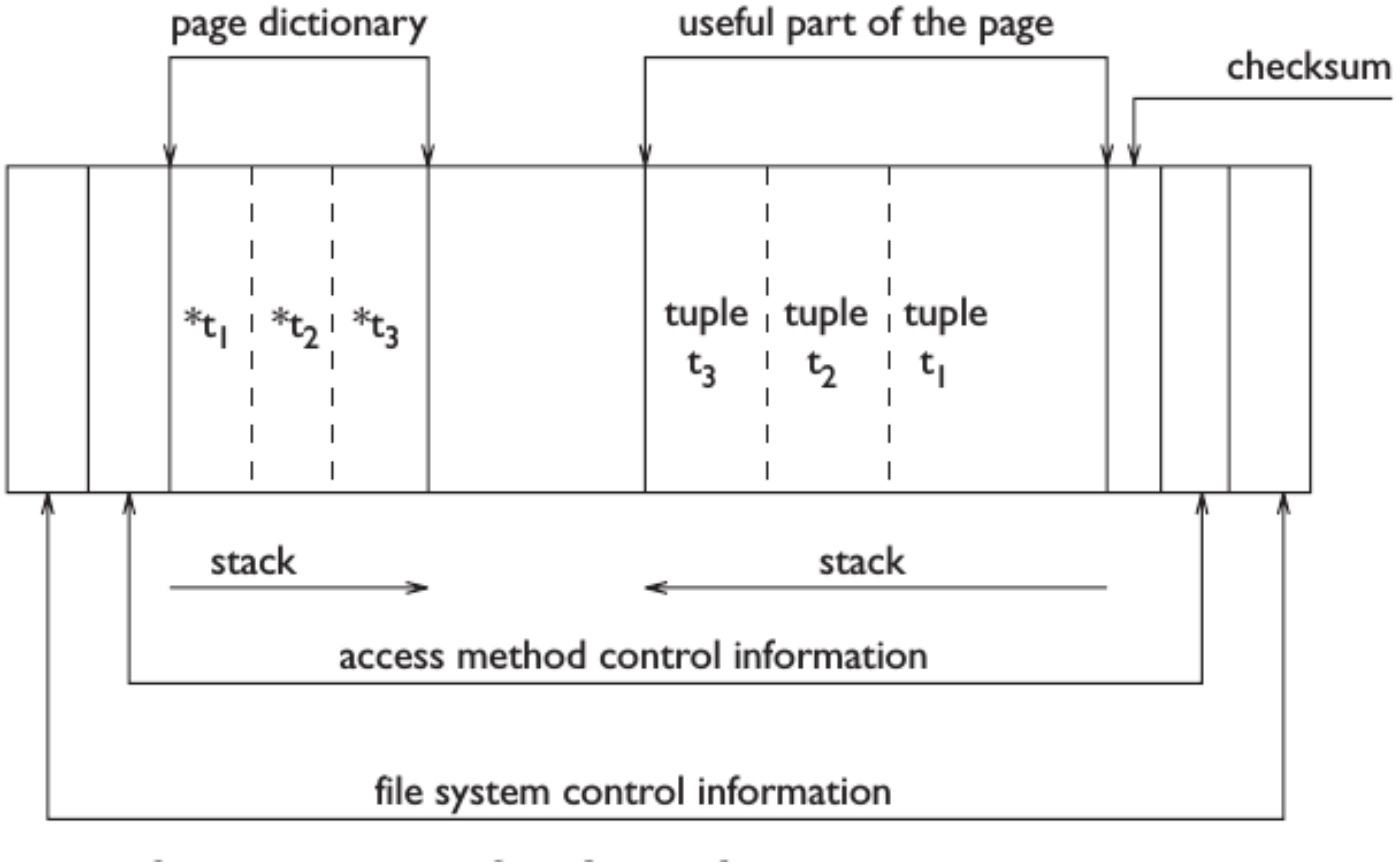
- The blocks ("physical" components of a file), and the tuples ("logical" components) have different dimensions in general:
  - the block size depends on the file system
  - the size of the tuple (simplifying a bit) depends on the application requirements, and can also vary within a file
- The distribution of tuples on the blocks becomes a critical factor to consider.

Blocking Factor

- Given:
  - L_R: Size of a tuple (constant, for simplicity: "tuples of fixed length")
  - L_B: Block size (e.g. 4KB)
- if $L_B >= L_R$, we can have floor(L_B / L_R) tuples in one block (blocking factor).
- In general, L_R can not divide exactly L_B so that each block can have some unused space. The remaining space can be:
  - used (record "spanned"), to store other tuples
  - not used ("unspanned")
- The spanned policy is mandatory when:

$$L_R > L_B$$

Organization of tuples into pages



Organization of tuples into pages

- In this case:
- Each page (which coincides with a block) has an initial part *block header* and a final part *block trailer.* Such parts are used by the operating system.
- Each page (containing the data) has an initial part *page header* and a final part *page trailer* containing control information relative to the physical structure.

Organization of tuples into pages

- In many cases, each page has a page dictionary with pointers to the actual data. Typically, the actual data and dictionaries grow into opposite directions in accordance with a stack structure.
- Each page contains a parity bit to check the validity of information contained.

Organization of tuples into pages

- The situation is complicated in the case of tuples of variable length, which are necessary in the case of tuples that contain VARCHAR, CLOB, BLOB or NULL values.
- In this case, the data dictionary will contain information about the offset of each tuple with respect to the beginning of the actual data.

Organization of tuples into pages

- The primitives offered by the manager of the secondary storage are:
- *Insert and update*: They can often lead to a reorganization of the page.
- *Cancellation*: Often involve a reorganization
- *Access to a tuple*: Takes place by means of a key value (associatively) or according to the offset stored in the dictionary.
- *Access to a field of a tuple*: First it requires identifying the tuple and after the identification of the field.

Organization of tuples into pages

- In DBMSs that manage data by "column" (column-oriented DBMSs), the representation of rows and fields is reversed.

Primary structures for file organization

- The primary structure of a file is the structure that defines the criterion according to which the tuples are arranged within the file itself.
- There are three categories of primary structures:
  - Sequential Structures
  - Structures with calculated Access (*hash*)
  - Tree Structures
- However, the tree structures are primarily used as secondary structure (for indexes).

Sequential Structures

- They are characterized by a substantially consecutive arrangement of the tuples in the mass memory which is based on a specific criterion (eg. order of insertion)
- There are three variants:
  - **Serial sequential structures (unordered)**: sorting is physical but not logical
  - **Sequential structures in form of array**: tuples are placed in predetermined positions
  - **Ordered sequential Structures**: Positions identified through *a* key field (not necessarily *the* primary key)

Serial Structures

- Also known as:
  - "Entry sequenced"
  - *Heap*
  - unordered files
- It is very common in relational databases
- The insertions are made
  - in the queue (with periodic reorganizations) using appropriate pointers to the last elements
  - In place of deleted records

Serial Structures

- Optimal for sequential read / write operations.
- Inefficient for searches of a single data (sequential scanning)
- Easy implementation
- In relational databases, they are used in combination with indexes (in order to overcome the problem of sequential scanning)

Array structures

- Tuples placed in predetermined positions
- Useful only for tuples of fixed size
- Each tuple has a unique number *i* which is used to determine its position
- High efficiency (in time) / low flexibility
- In relational databases, they are very rarely used.

Ordered sequential structures

- Positions identified through *a* key field (not necessarily *the* primary key)
- Optimized random access with the key
- Very high management costs (insertion may require a reorganization)
- Very used with tape devices (and differential file)
- They allow binary search, but only up to a certain point (for example, how to find the point at the "middle of the file"?)

Ordered structures

- The ordered structures are used in DBMS only in association with indices, forming a structure that takes the name of ISAM (*index sequential access method)*.

Structures with calculated access (Hash)

- They allow an associative access to tuples (as in an ordered sequential structure)
- However, the tuples do not need to be physically ordered in the mass memory
- The technique is based on the one used for the hash tables in main memory
- Objective: direct access to a set of tuples on the basis of the value of a field (called **key,** which for simplicity we assume to be an identifier, but not necessarily)

Hash table

- If the possible values of the key are in number comparable to the number of tuples then we could use an array; for example: university with 1,000 students and serial numbers between 1 and 1000 or so, and file with all students
- If the possible values of the key are much more than those actually used, we can not use the array (waste of space); eg:
  - 40 students and serial number of 6 digits (one million possible keys)

Hash table

- If we want to continue to use something like an array, but without wasting space, we can think of turning the key values that can indicate an array:
  - **hash function:**
    - associates to each key value an "address", in a space of comparable size (slightly higher space) compared to what is strictly necessary
    - since the number of possible keys is by far greater than the number of possible addresses ("the key space is larger than the address space"), the function can not be injective, and then there is the possibility of collisions (different keys that correspond at the same address)
    - good hash functions uniformly distribute entities, reducing the chances of a collision (which is reduced by increasing the redundant space)

An example

- 40 records
- hash table with 50 positions:
  - 1 cases with 4 collisions
  - 2 cases with 3 collisions
  - 5 cases with 2 collisions

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

File hashed structure, collisions

- When we use the file to store the tuples, it is necessary to consider some parameters:
- We can define the filling factor *f*: the fraction of the physical space used, on average.
- Let *T* be the number of rows expected for the file, *F* be the blocking factor, then the file may provide a number of blocks *B* equal to the integer immediately above
  - $T / (f * F)$
- In this case, the hash function must return a number between 0 and *B*-1

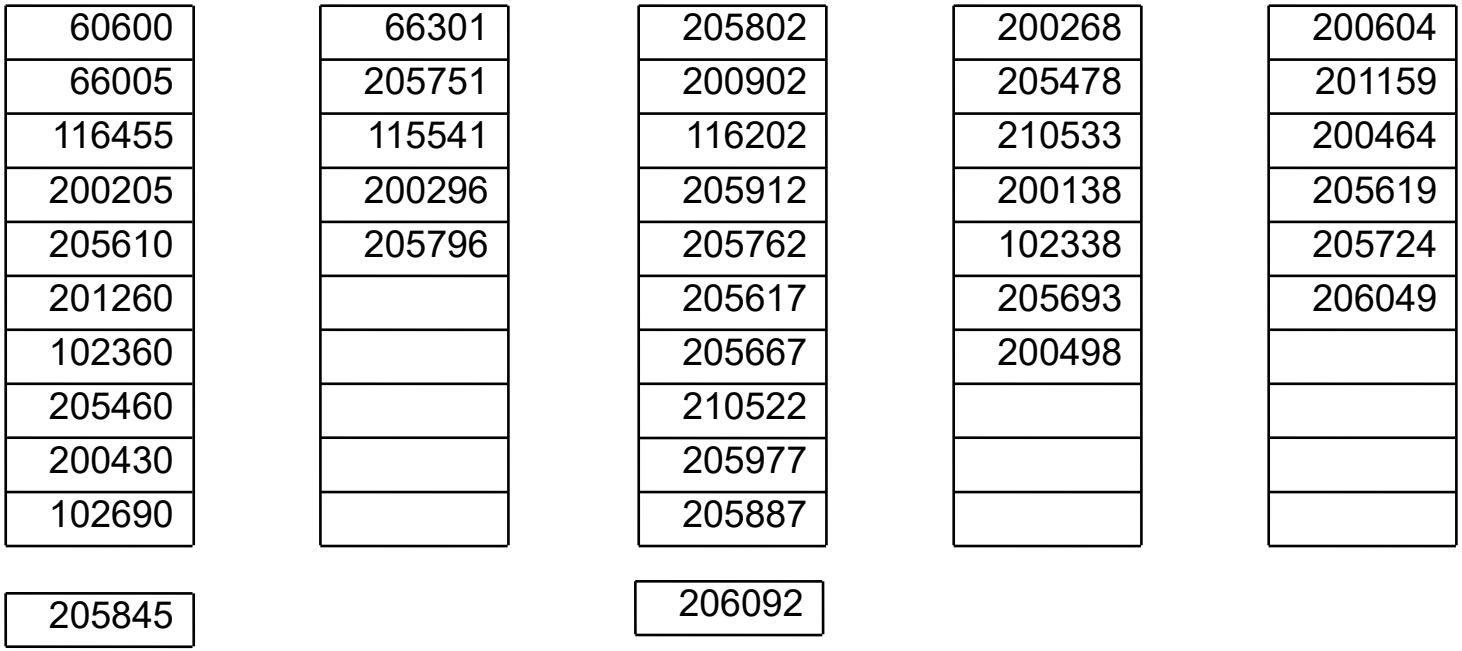
An example

- 40 tuples
- hash table with 50 positions:
  - 1 case with 4 collisions
  - 2 cases with 3 collisions
  - 5 cases with 2 collisions
    - average number of accesses: 1,425
- hash file with blocking factor 10 and filling factor 0.8
  5 blocks with 10 positions each:
  - only two overflows!
    - average number of accesses: 1.05

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

An example: The blocks



Hash table: collisions

- Collisions can be handled by various techniques:
  - successive positions available (on average are accepted *F* collisions)
  - table or overflow chains (using new blocks connected to the one used initially)
  - hash functions "alternative"
- Note:
  - there are collisions (almost) always
  - multiple collisions have probability decreases with increasing multiplicity
  - the average multiplicity of collision is very low

Hash Structure

- Average length of the overflow chains of collisions:

filling factor	blocking factor 1	3	5	10
0.7	1.167	0.286	0.136	0.042
0.8	2	0.554	0.289	0.11
0.9	4.495	1.377	0.777	0.345

- (Gray & Reuter, 1994)

Hash structure, observations

- It is the most efficient organization for direct access based on the key values of equality conditions (timely access): average cost of just above one (the worst case is very expensive but so unlikely that it can be ignored)
- Collisions are usually managed with connected blocks (Overflow)
- It is not efficient for searches based on intervals
- There is no benefit for searches based on other attributes
- The hash file "degenerates" if we reduce the free space: only work with files whose size does not vary much over time

Tree Structures

- The tree structures are in most cases used for the definition of indexes. Some texts even consider the term "index" synonymous of the term "tree"
- Index:
  - auxiliary structure to access (efficiently) to tuples of a file on the basis of the values of a field (or a "sequence" of fields) said key (or, better, pseudokey, because it is not necessarily identifying);

Types of indexes

- primary structures:
  - They contain data
- secondary indexes
  - They facilitate access to data without containing them
- dense indexes:
  - contain a record for each value of the key field
- sparse indexes
  - contain a lower number of records than the number of different values of the key field (eg. pointing to the first tuple of the block)

Secondary Index

- Basic idea: the index of a book: the list of pairs (word, page), sorted alphabetically on words, put at the end of the book and separable from it
- an index *I* of a file *f* is another file with records with two fields: Key and Address (of the records of *f* or to the blocks of *f*), ordered according to the values of the key

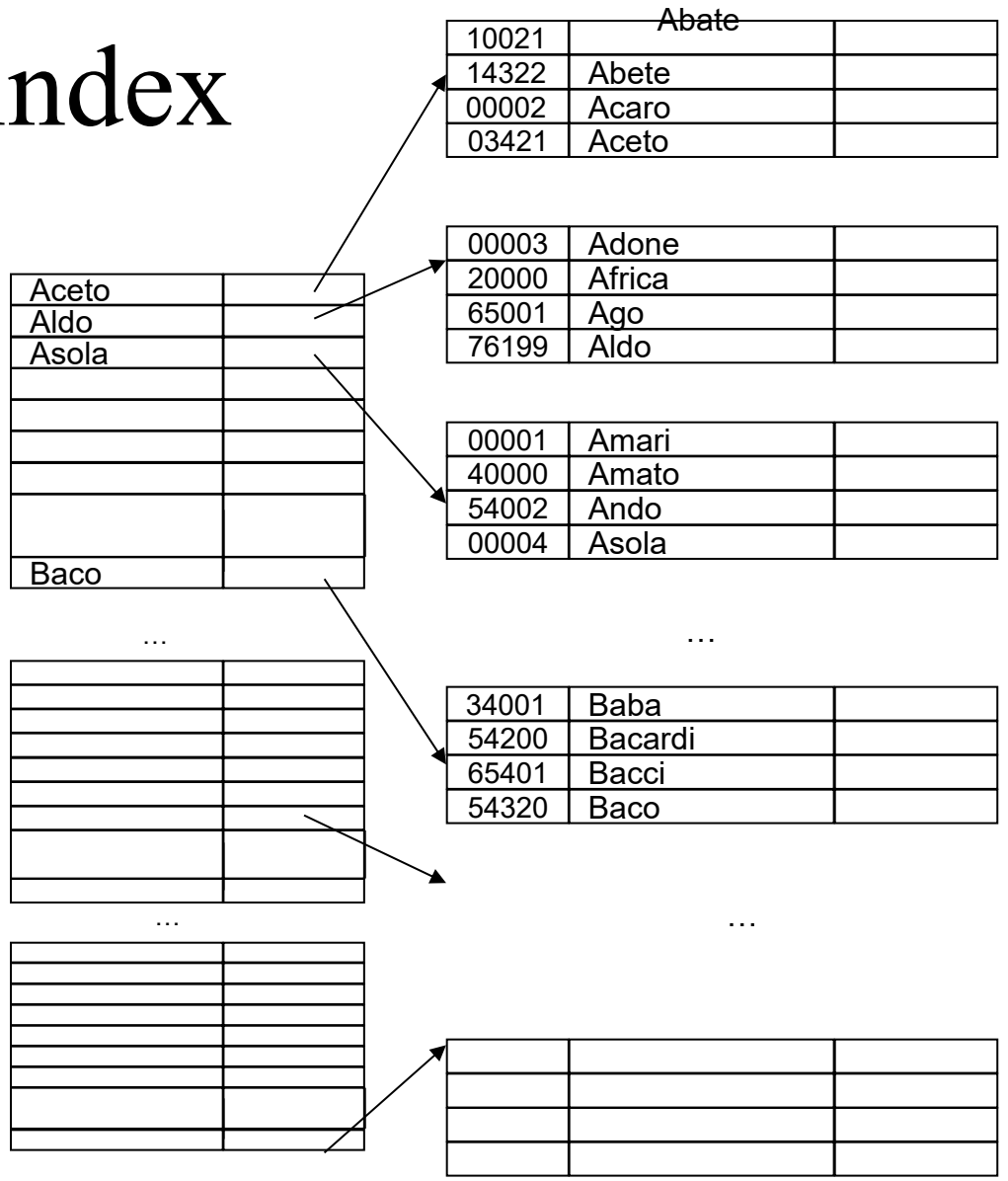
Primary data structure

- It is called “primary” because it not only ensures access on the basis of key, but also includes the physical locations required to store the data (primary structure).

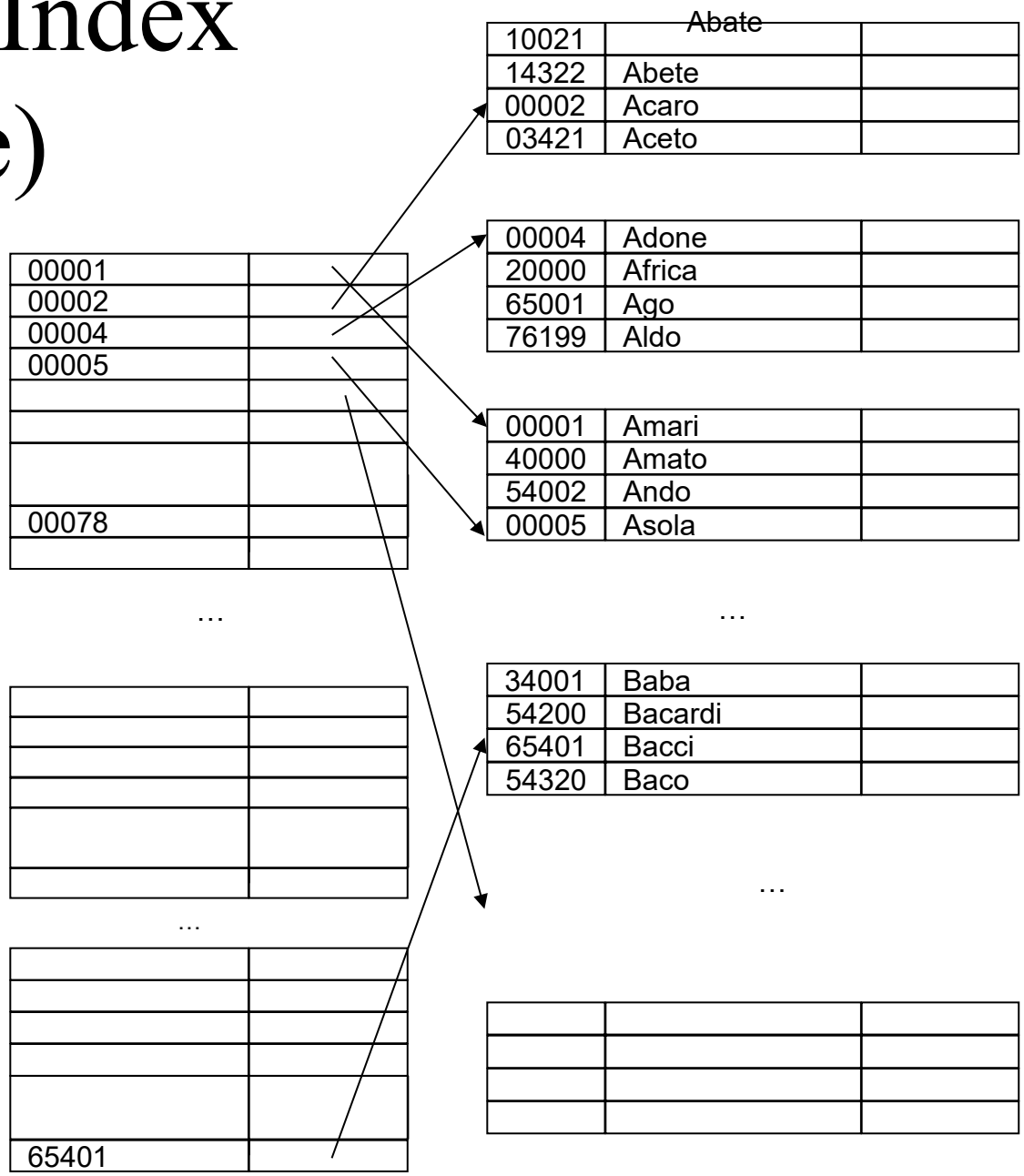
Index Types, comments

- A primary index can be sparse, a secondary must be dense
- For example, again with respect to a book
  - Table of contents (primary)
  - Analytical index (secondary)
- The benefits related to the presence of secondary indexes are significant
- Each file can have at most one primary index and any number of secondary indexes (on different columns). Example:
  - a tour guide may have an index of places, one of the restaurants or hotels
- A hash file may not have a primary index (the storage criterion is based on heap or hash structure)

Primary structure and primary index (Sparse)



Secondary Index (Dense)



Index Size

- L number of records in the file
- B block size
- R length of tuples (fixed)
- K length of the key field
- P length of the addresses (pointers to blocks)

- Number of blocks for the file (approximately): $N_F = L / (B / R)$
- Number of blocks for a dense index: $N_D = L / (B / (K + P))$
- Number of blocks for a sparse index: $N_S = N_F / (B / (K + P))$

Properties of the indexes

- Direct access (on the key) efficient, both for single values and for intervals
- Efficient ordered-sequential scanning
  - –All indexes (in particular the secondary ones) provide a **logical ordering** on the record of the file; with a number of accesses equal to the number of records in the file (there are some benefits due to buffering)
- Changes of the key, insertions, deletions are inefficient (as in sorted files)
  - techniques to alleviate the problems:
    - Overflow files or blocks
    - Marking for eliminations
    - Partial filling
    - Connected blocks (non-contiguous)
    - Periodic reorganizations

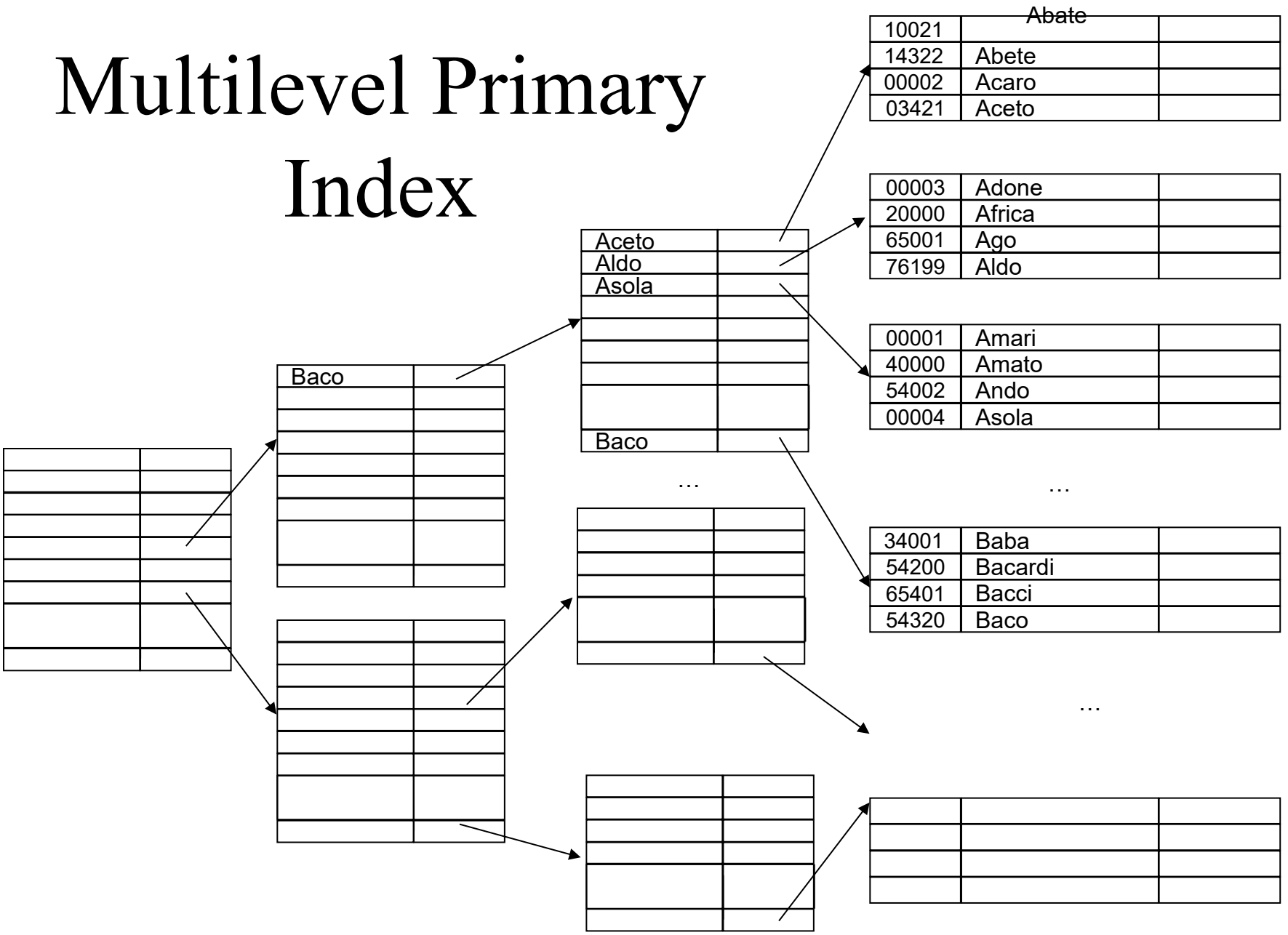
Secondary indexes, comments

- The DBMS can use, as mentioned, pointers to the blocks or pointers to tuples
  - The pointers to the blocks are more compact
  - The pointers to tuples allow
    - to simplify some operations (carried out only on the index, without accessing the file, if not strictly necessary)
    - For example. some aggregation operations
      - »**Count,** all times
      - »avg, min, max, etc. only in special cases

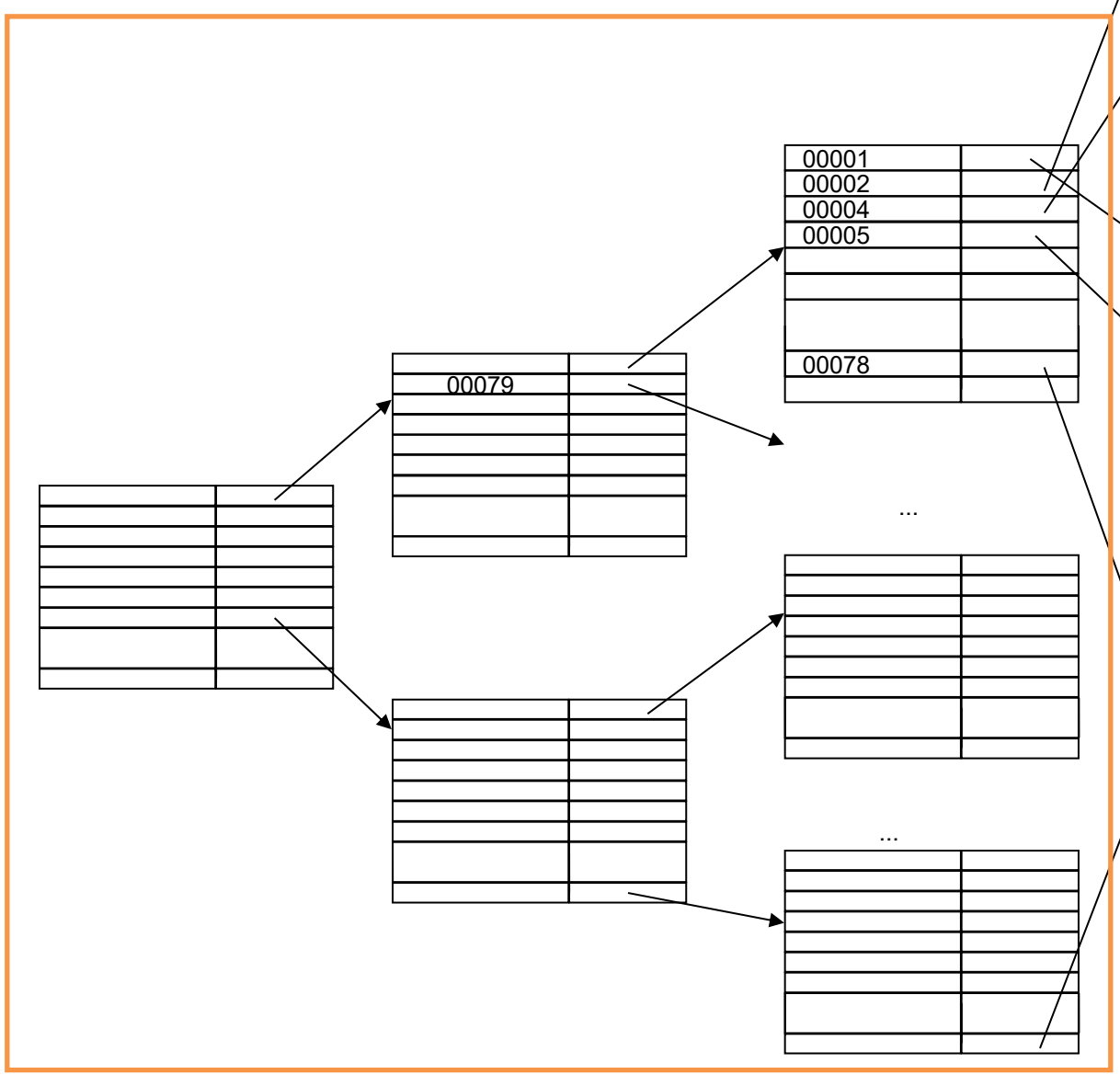
Multilevel Indexes

- Indexes are files themselves, and as so it makes sense to build indexes on indexes, to avoid doing research among different blocks
- There can be multiple levels and one block at the highest level; there should be only few levels, because
  - the index is ordered, so the index on the index can be sparse
  - the index records are small
- N_j, the number of blocks at the level j of the index is (approximately):
  - $N_j = N_{j-1} / (B / (K + P))$

Multilevel Primary Index



Multilevel Secondary Index



10021	Abate	
14322	Abete	
00002	Acaro	
03421	Aceto	

00004	Adone	
20000	Africa	
65001	Ago	
76199	Aldo	

00001	Amari	
40000	Amato	
54002	Ando	
00005	Asola	

...

34001	Baba	
54200	Bacardi	
65401	Bacci	
54320	Baco	

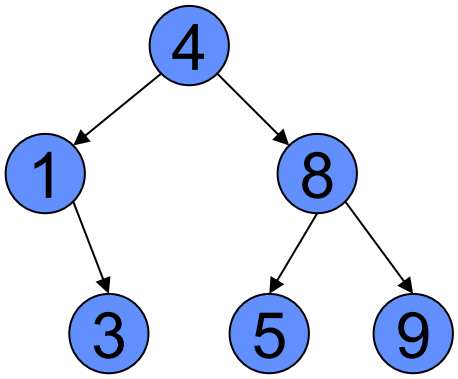
...

Indexes, problems

- All the index structures seen so far are based on ordered structures and are therefore not flexible in the presence of high dynamism
- The indices used by the DBMS are more sophisticated:
  - dynamic multilevel indexes: B-tree, B +-*tree* (Intuitively: search trees balanced)
    - We arrive at the B-tree step-by-step
      - Binary search trees
      - *n*-ry search trees
      - *n*-ary balanced search trees

Binary search tree

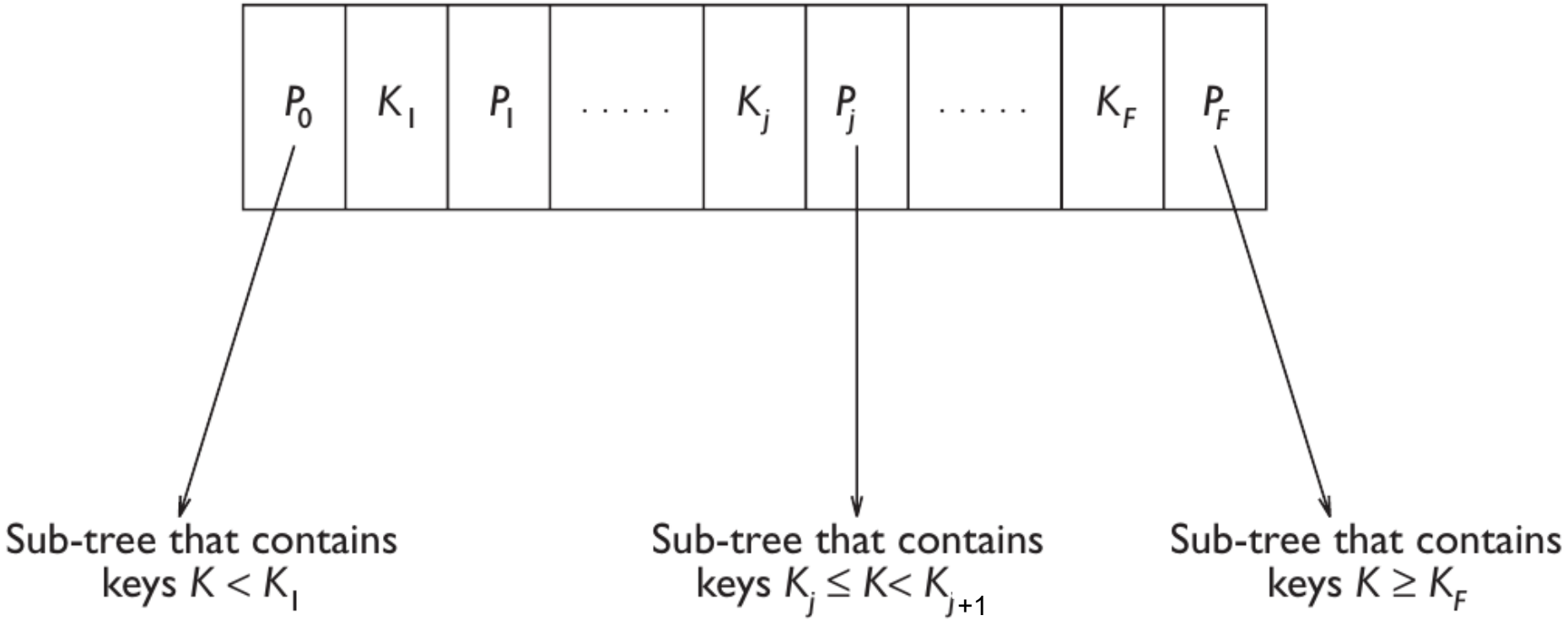
- A labeled binary tree in which each node's left subtree contains only labels smaller than the node's label, and the right subtree contains only labels larger than the node's label.
- Search time (and insertion time) is equal to the depth of the tree:
  - Logarithmic on average (assuming random insertion)



Search tree of order P

- Each node has (up to) P children and (up to) P-1 labels, ordered
- In the *i*-th subtree we have all the labels greater than the (*i*-1)-th and smaller than the *i*-th
- Any search or modification involves a visit of a root-leaf path
- In physical structures, a node may correspond to a block
- The connections are based on pointers and the number of descendants of a node is generally quite high
- A B-tree is a search tree which is kept balanced, thanks to:
  - Partial Filling (average 70%)
  - Reorganizations (local) in the case of unbalance

Organization of the B-tree nodes



The number F + 1 is called *fan-out* of the tree

Search and leaves

- The search uses the keys in internal nodes to reach the leaves.
- The leaves can be organized in two different ways:
  - *Index-sequential*: The leaves contain tuples directly. In this case, an ordered file with associated primary index is created. The position of the single tuple is constrained by the value of its key (insertions and deletions are not as heavy as in array sequential structures or hash structures)
  - *Indirect*: Each leaf node contains pointers to blocks that contain the tuple with the value of the specified key.
    - good for the creation of secondary indexes. Cost of insertions and deletions depends on the organization of the primary structure

Insertions and deletions

- Insertions and deletions
  - They may also result in a modification of the indexes
  - They are preceded by a search until a leaf
- For insertions,
  - if there is space available in the leaf, the tuple is inserted, and the index is not changed
  - otherwise, the node must be divided (*split*), generating two leaves instead of one. In this case, there is the need for an additional pointer in the parent node; if there is no available space in the block containing the parent node, it goes up again, possibly up to the root.

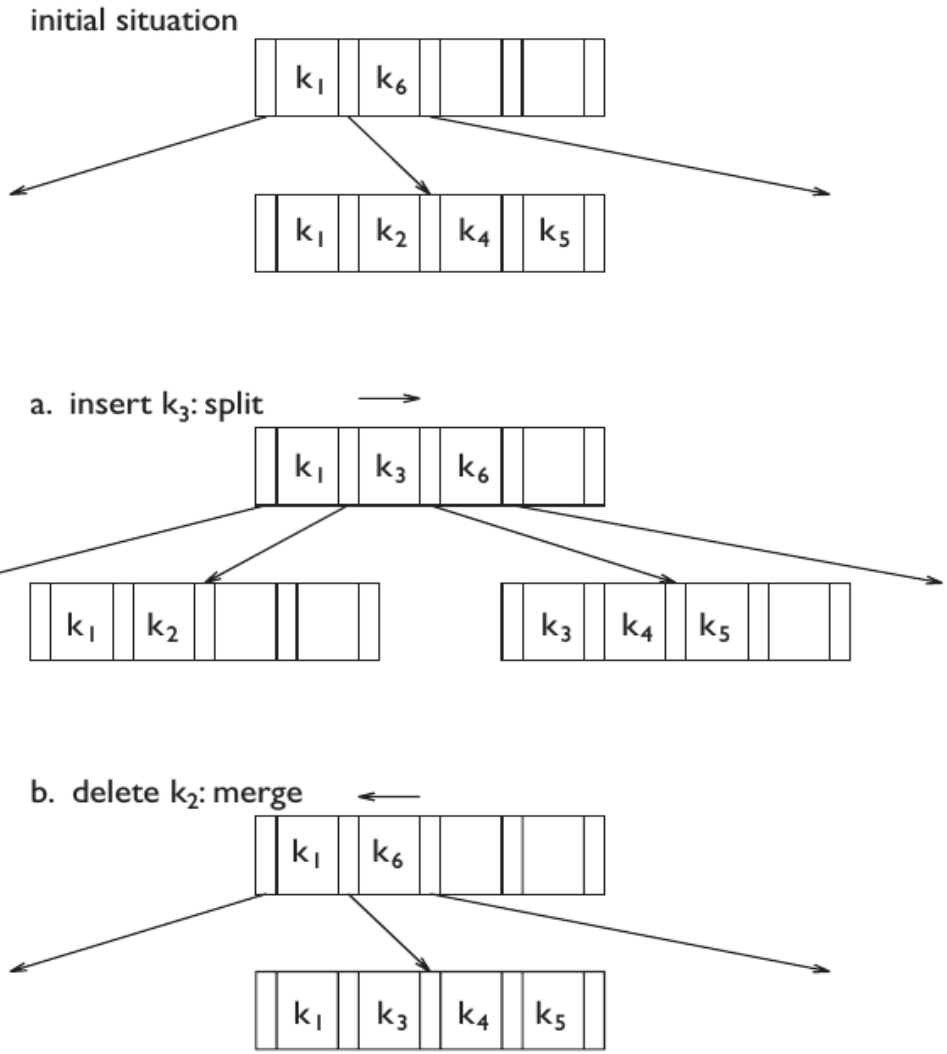
Insertions and deletions

- Cancellations can be made on site by marking the space previously allocated for a tuple as *invalid*.
- Problems:
  - If cancellation concerns one of the key values present in the tree, it is appropriate to retrieve the next value and store it in place of the deleted value.
  - When deletion leads to two contiguous pages that can be collapsed into one, it is necessary to provide a new operation, dual to split, called *Merge.*
    - The merge can obviously propagate recursively.

Entries and cancellations

- The filling factor must always remain greater than 50% and, on the basis of statistical evaluations, the optimal case is an average value equal to 70%

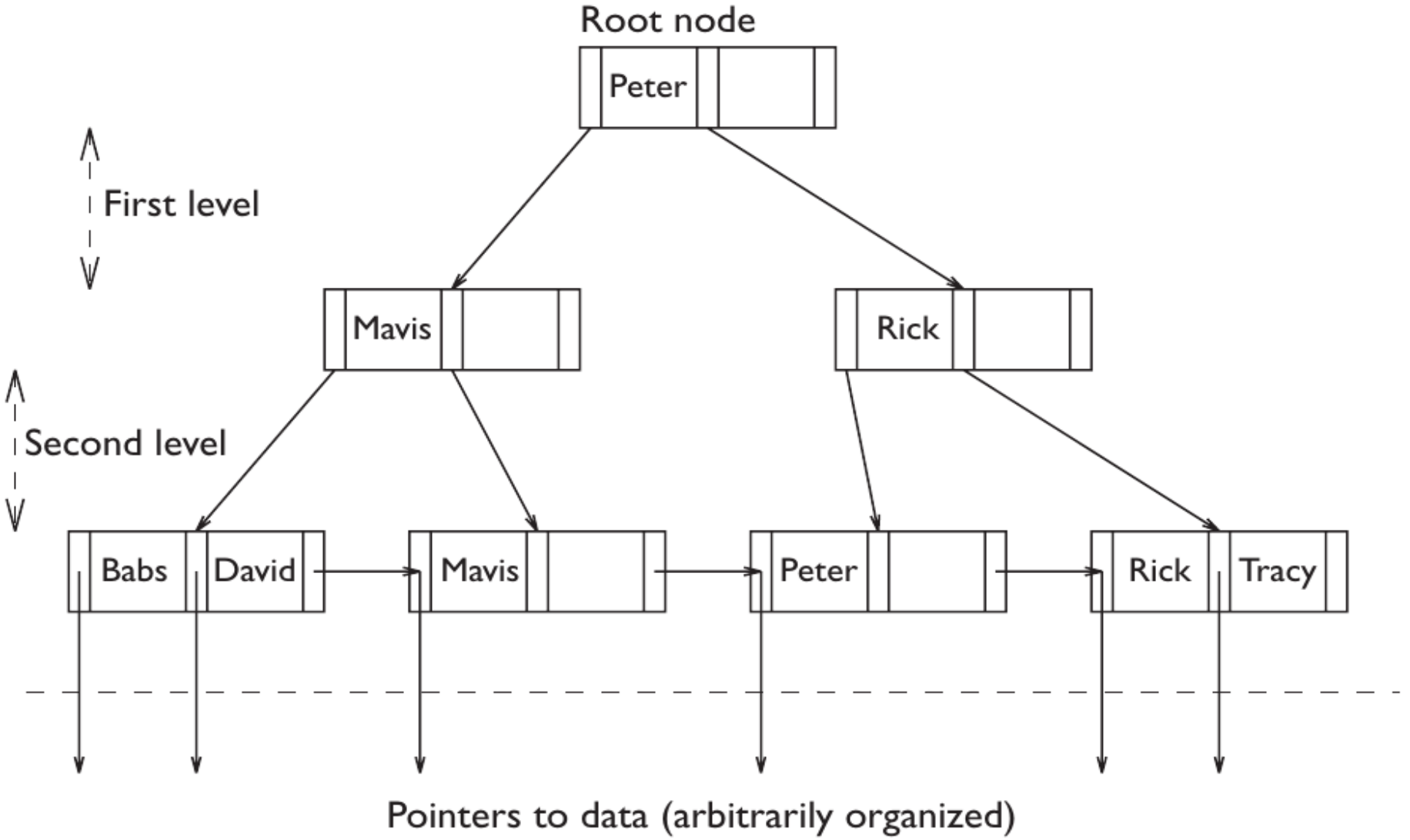
Split and Merge



B tree and B + trees

- B-trees differ from B+trees because the latter are provided with a chain which connects the leaf nodes based on the order imposed by the key

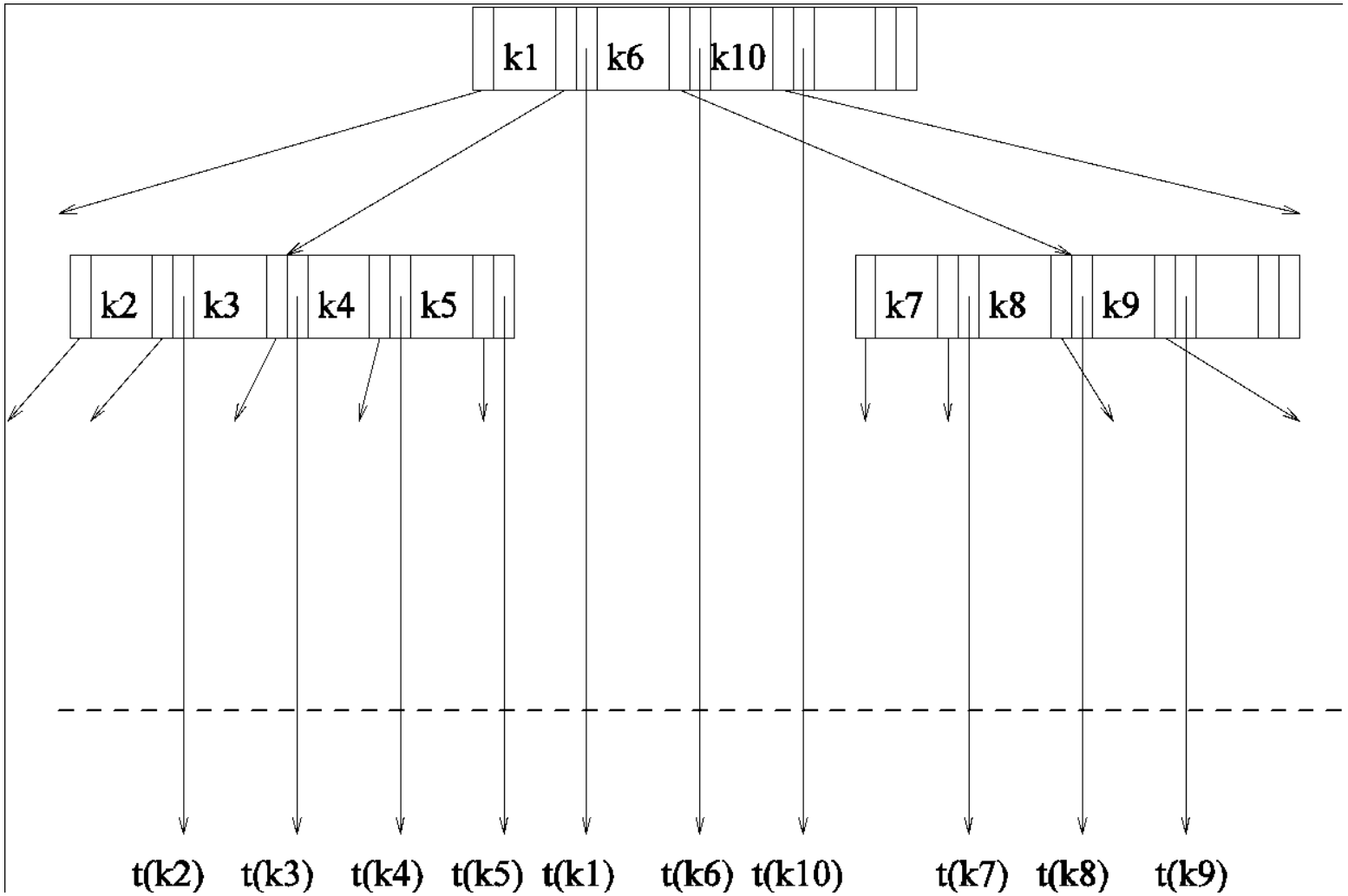
B + tree


Root node
Peter
First level
Mavis
Rick
Second level
Babs
David
Mavis
Peter
Rick
Tracy
Pointers to data (arbitrarily organized)

B tree and B + tree

- B + tree:
  - excellent for searches on intervals and in the case of the leaves organization both of type *index sequential* and *indirect*
  - widely used in DBMSs (larger than B-trees, but lower search time in case of queries on key intervals)
- B tree:
  - An optimization can be expected that intermediate nodes can have direct pointers to the data. In this way it is not necessary to reach a leaf node to access the tuple without the need to waste more space (pointers to the tuples are not repeated)

B-tree


k1
k6
k10
k2
k3
k4
k5
k7
k8
k9
t(k2)
t(k3)
t(k4)
t(k5)
t(k1)
t(k6)
t(k10)
t(k7)
t(k8)
t(k9)

B tree and B + tree

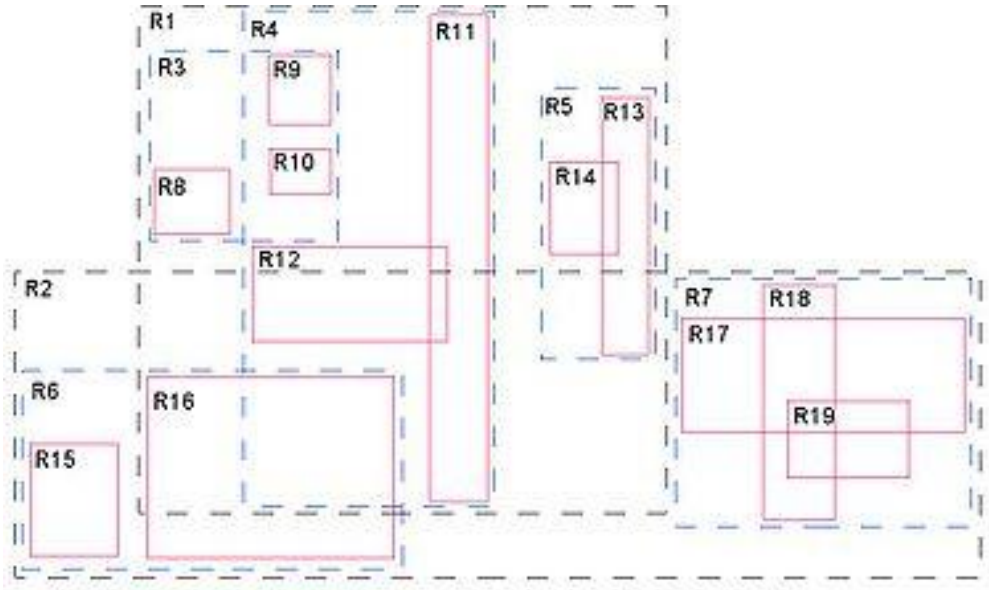
- Often the highest tree nodes reside in the buffer. In addition, they are often not changed: most transactions can exploit them simultaneously.
- A possible reduction of space can be achieved by compression of one of the keys
  - For example, keeping only the key code at the highest levels and only the suffixes at lower levels.

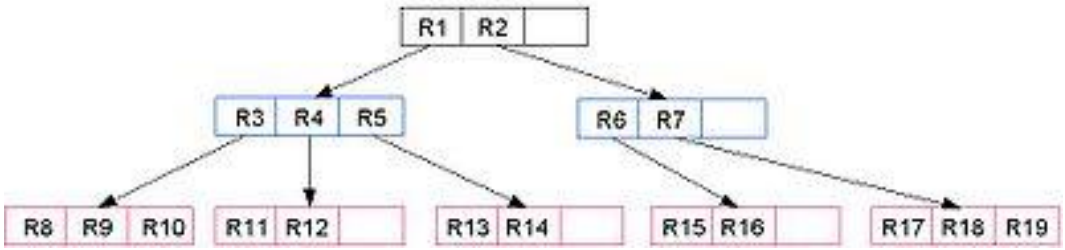
Other indexes (spatial)

- A r-tree is a tree data structure similar to the B-tree, but it is used to index multidimensional spaces, for example the spatial coordinates (X, Y) for geographic data. The data structure divides the space into MBR (*minimum bounding rectangles*) hierarchically nested and, when possible, overlapping.
- Each node of the r-tree has a variable number of entries (up to a predetermined maximum). Any entry that is not a leaf node contains two entities: one identifying the child node, the other the MBR that contains all the entries of the child node.
- The insertion algorithm and deletion of entries by MBR ensures that "spatially close" elements are positioned in the same place (leaf node): a new element will go into the leaf node that requires the least number of extensions of the size of the MBR).
- The search algorithms use the MBR to decide whether or not to search the child node of the current node.
- By the r-tree we can efficiently respond to statements such as: "Find all museums within 2 km from my current position."

About indexes

- An example of r-tree.





The physical structures in relational DBMSs

- The DBMS on the market differ significantly for the details of the physical structures that make available (especially with regard to quantitative parameters)
- But...
  - Almost all systems provide unordered sequential structure as basic structure on which it is possible to define secondary indexes.
  - Almost all systems create an index for the primary key for efficient verification of compliance with constraints. Many systems indicate as *primary index* the index on the primary key. (Not to be confused with the index according to which the file is sorted)

The physical structures in relational DBMSs

- Many systems provide for the possibility of contiguously store the tuples in a table with the same values of a certain field. Some systems call this technique '*cluster*'.
- Some systems provide the ability to define cluster involving multiple relationships.
  - Easy join: tuples of the two relations on which we perform the join are stored together: the cost of the join is linear
- All systems allow the creation of indexes and almost all systems provide realizations based on B+tree (although the documentation often talks about B-tree)

The physical structures in relational DBMSs

- Some systems also provide other types of indexes (bitmap - used in DW)
- Some systems provide *hash indexes*.

Definition of indexes in SQL

- Not standard (in SQL3 and earlier versions):
  - There is no agreement in the standards committee
  - Indexes are considered closely related to the DBMS implementation
- Present in similar form in the various DBMS
  - `create [unique] index` *IndexName* `on` *TableName*(*AttributeList*)
  - `drop index` *IndexName*
- In creation, the order in which the attributes appear is important because it determines the ordering of tuples
- `unique` indicates that the attribute can not be duplicated

Definition of indexes in SQL

- `drop index` *IndexName*

- The elimination of the indices is useful when the application does not need to query the data using the index: in these cases it becomes costly to maintain the index (in terms of space occupied).

Persons

ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234

Persons

ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

Persons

ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

Persons

ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

Persons

I1

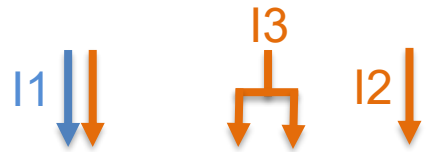
ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

SELECT * FROM Persons WHERE Name="Mario" and Surname="Rossi"

Persons



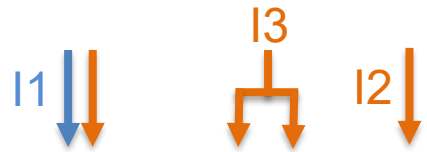
ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

SELECT * FROM Persons WHERE Name="Mario" and Surname="Rossi"

Persons



ID	Name	Surname	Age

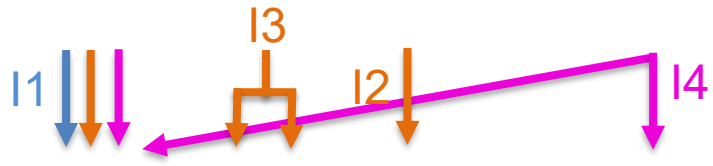
SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

SELECT * FROM Persons WHERE Name="Mario" and Surname="Rossi"

SELECT * FROM Persons ORDER BY Name, Age

Persons



ID	Name	Surname	Age

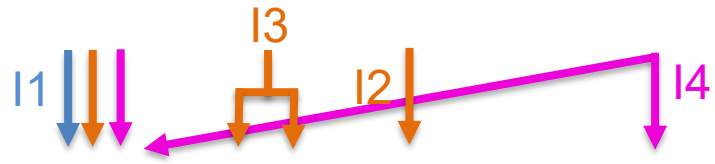
SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

SELECT * FROM Persons WHERE Name="Mario" and Surname="Rossi"

SELECT * FROM Persons ORDER BY Name, Age

Persons



ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

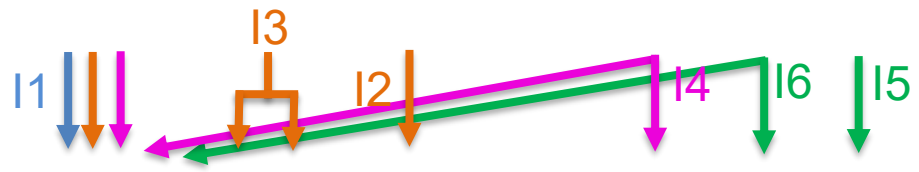
SELECT * FROM Persons WHERE Name="Mario"

SELECT * FROM Persons WHERE Name="Mario" and Surname="Rossi"

SELECT * FROM Persons ORDER BY Name, Age

SELECT * FROM Persons ORDER BY Age, Name

Persons



ID	Name	Surname	Age

SELECT * FROM Persons WHERE ID = 1234 ← Optimized by default

SELECT * FROM Persons WHERE Name="Mario"

SELECT * FROM Persons WHERE Name="Mario" and Surname="Rossi"

SELECT * FROM Persons ORDER BY Name, Age

SELECT * FROM Persons ORDER BY Age, Name

The physical structures in some commercial DBMSs

- Oracle:
  - primary structure
    - Heap files
    - "Hash cluster" (i.e. hash structure)
    - cluster (also with multiple tables) also ordered (with dense B-tree)
  - Secondary indices of various types (B-tree, bit-map, hash functions)
- DB2:
  - Primary: heap or ordered with dense B-tree
  - index on the primary key (automatically)
  - secondary B-tree dense indexes

The physical structures in some commercial DBMSs

- SQL Server:
  - Primary: heap or ordered with B-tree sparse index
  - secondary B-tree dense indexes (called clustered)
- PostgreSQL
  - Primary: ordered with B-tree sparse index
  - Secondary: B-tree dense indexes

Oracle: cluster definition

For the Oracle documentation:

- Clusters are groups of one or more tables physically stored together because they share common columns and are often used together. Because related rows are physically stored together, disk access time improves.
- Like indexes, clusters do not affect application design. Whether a table is part of a cluster is transparent to users and to applications. Data stored in a clustered table is accessed by SQL in the same way as data stored in a nonclustered table.

Oracle: cluster definition

```
CREATE CLUSTER personnel
    (department NUMBER(4))
SIZE 512
STORAGE (initial 100K next 50K);
```

`SIZE` is used to specify the amount of space in bytes reserved to store all rows with the same cluster key value or the same hash value. This space determines the maximum number of cluster or hash values stored in a data block. If SIZE is not a divisor of the data block size, then Oracle Database uses the next largest divisor. If SIZE is larger than the data block size, then the database uses the operating system block size, reserving at least one data block for each cluster or hash value.

Oracle: cluster definition

On the cluster it is possible to define the indexes

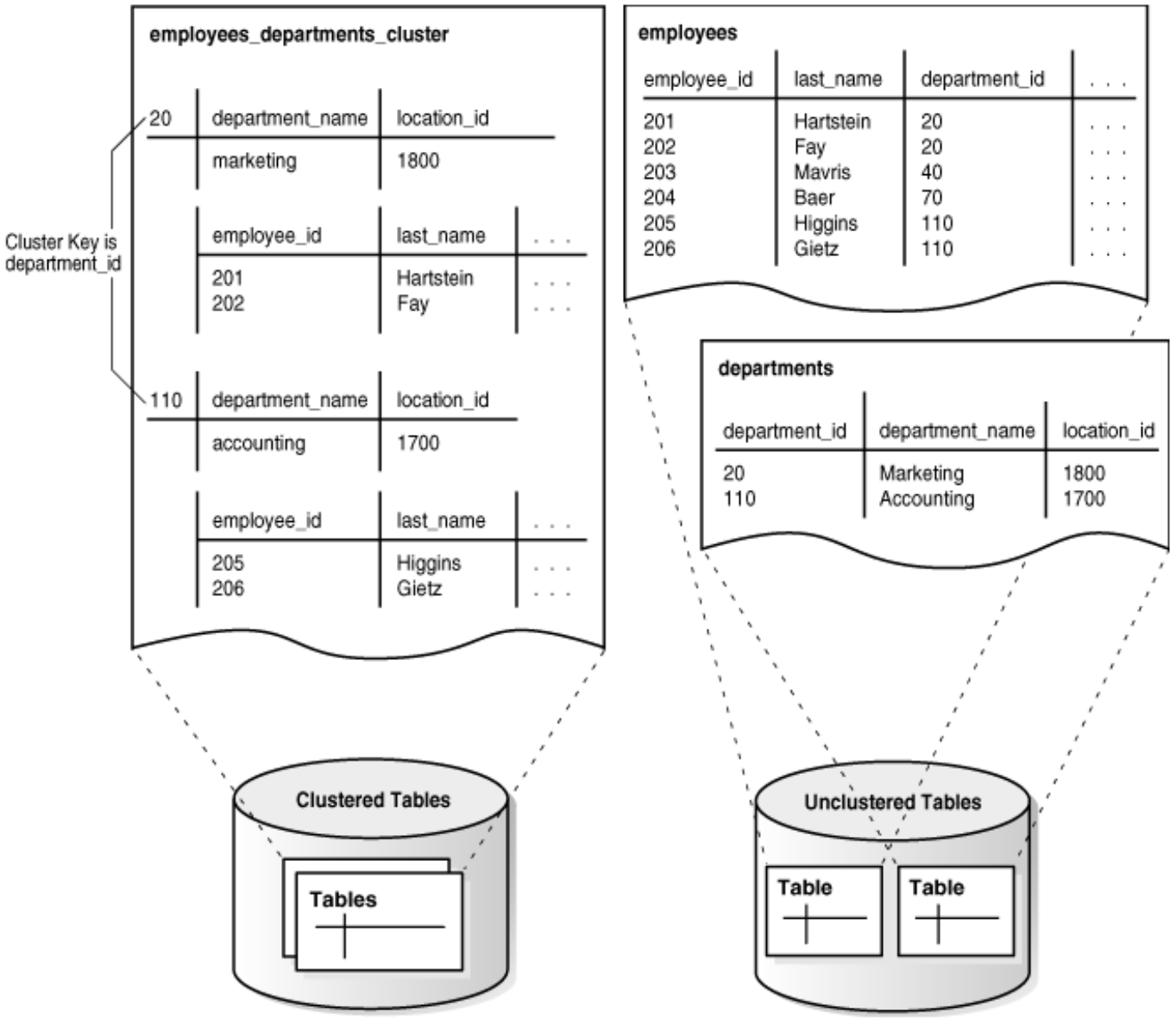
```
CREATE INDEX idx_personnel ON CLUSTER
personnel;
CREATE TABLE dept_10
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE
department_id = 10;
```

The columns listed in clause `CLUSTER` are the columns `employees` that will be stored in the cluster.

Oracle: cluster definition

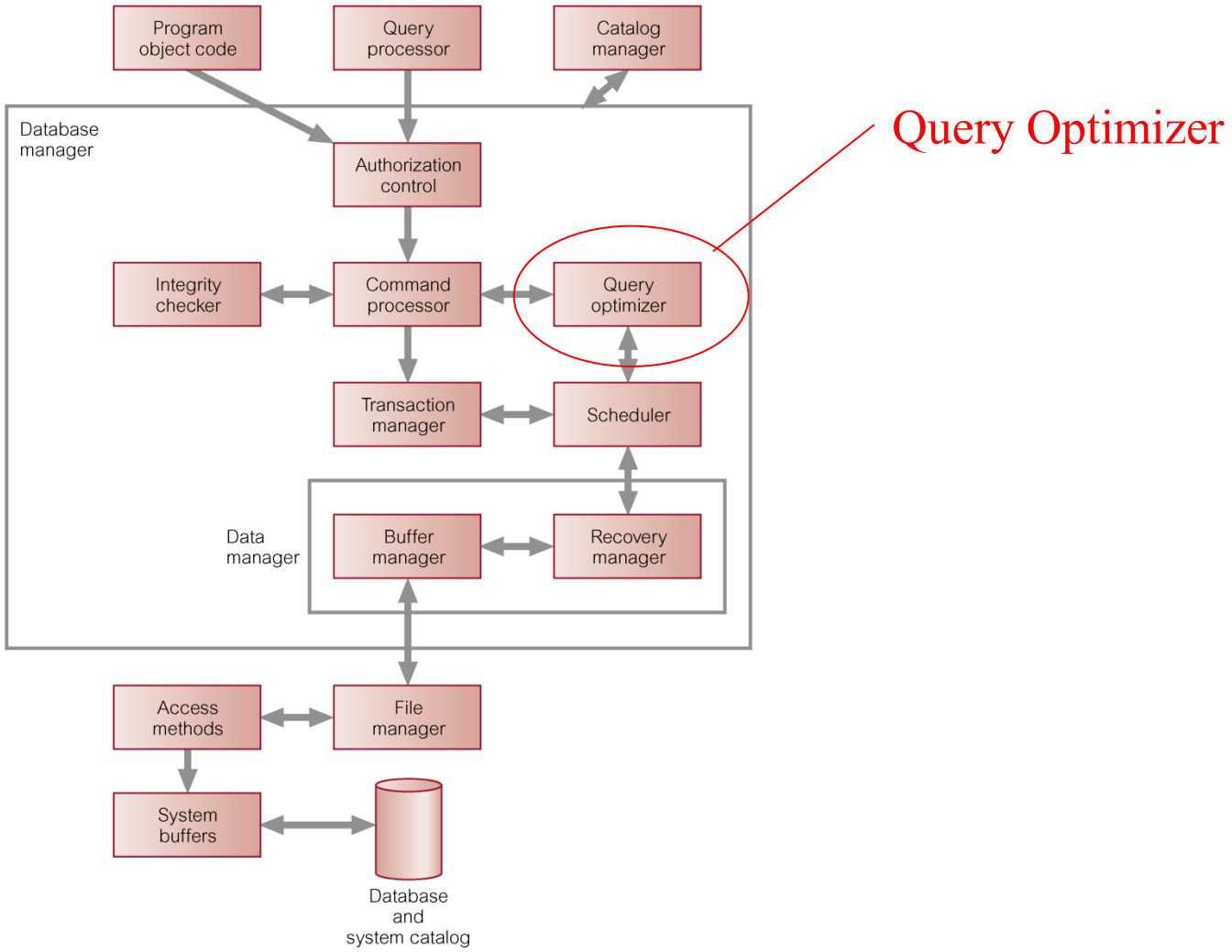
```
CREATE TABLE dept_20
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE
department_id = 20;

CREATE CLUSTER address
   (postal_code NUMBER, country_id CHAR(2))
   HASHKEYS 20
   HASH IS MOD(postal_code + country_id,
101);
```



https://docs.oracle.com/database/121/CNCPT/tablecls.htm#CNCPT609
https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5001.htm

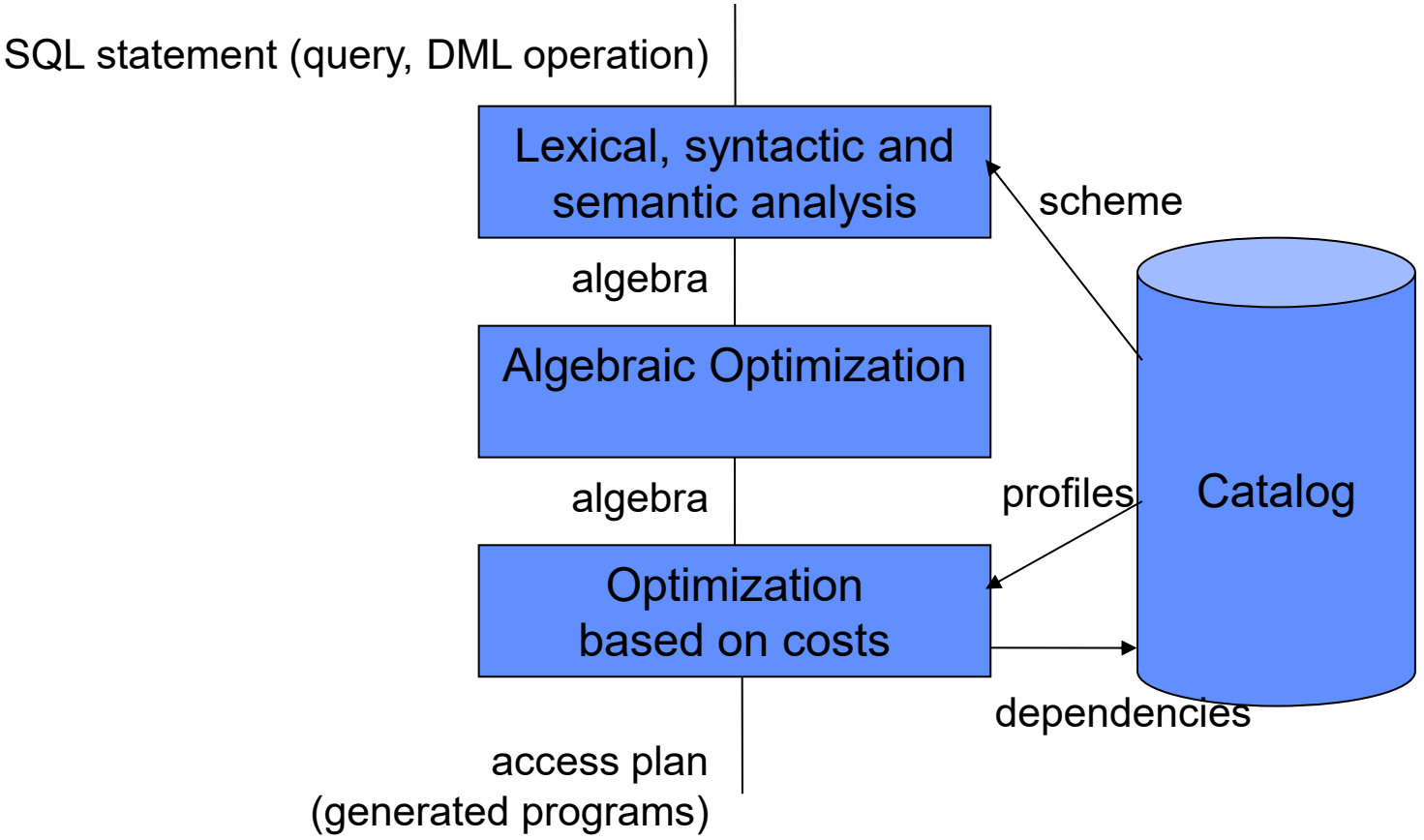
Query execution and optimization



Query execution and optimization

- More important in the existing systems than in the old ones (hierarchical and network):
  - queries are expressed at high levels (remember the concept of ***data independence***):
    - tuple sets
    - little procedurality
  - the optimizer chooses the implementation strategy (usually between different alternatives), starting from the SQL statement

Query execution and optimization



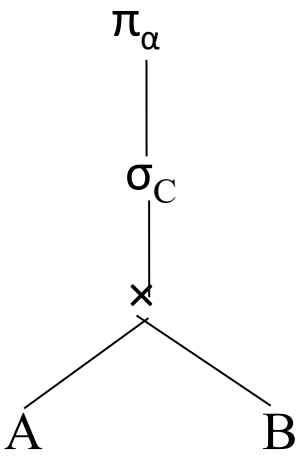
Algebraic Optimization

An expression in relational algebra can be transformed into another *equivalent,* taking advantage of some of the operators properties. These transformations are useful because they can lead to reduce by orders of magnitude the cost of running the expressions (***algebraic optimization***).

Algebraic Optimization

Consider the representation of an algebraic expression as a *tree,* whose leaves are the tables and internal nodes are the algebra operators. The children of an internal node *N* are operands of the operator associated with the node *N*.

Select α from A, B where C



Algebraic Optimization

- How to optimize?

Anticipating the application of projection and selection operators with respect to the product, so as to reduce the size of intermediate results. Graphically, it means moving the projection and selection towards the leaves.

In fact, the selection operator produces a temporary table with a lower number of tuples compared to the table to which it is applied.

The projection reduces the size of the tuples of the operand and eliminates duplicated tuples from the result.

Algebraic Optimization (not asked at the exam)

Let E be an expression of relational algebra, then the following properties are valid:

1. Grouping of selections

$$\sigma_{C_X}(\sigma_{C_Y}(E)) = \sigma_{C_X \wedge C_Y}(E)$$

C_X is a condition on the set of attributes X.

2. Commutativity of selection and projection

$$\pi_Y(\sigma_{C_X}(E)) = \sigma_{C_X}(\pi_Y(E)), \quad \text{if} \quad X \subseteq Y$$

$$\pi_Y(\sigma_{C_X}(E)) = \pi_Y(\sigma_{C_X}(\pi_{XY}(E))), \quad \text{if} \quad X \not\subseteq Y$$

Algebraic Optimization (not asked at the exam)

3. Anticipation of selection with respect to the product

$$\sigma_{C_X}(E_1 \times E_2) = \sigma_{C_X}(E_1) \times E_2 \qquad \text{se } X \text{ sono attributi di } E_1$$

$$\sigma_{C_X \wedge C_Y}(E_1 \times E_2) = \sigma_{C_X}(E_1) \times \sigma_{C_Y}(E_2) \qquad \text{se } X \text{ sono attr. di } E_1 \text{ e } Y \text{ attr. di } E_2$$

Algebraic Optimization (not asked at the exam)

4. Grouping of projections

$\pi_Z(\pi_Y(E)) = \pi_Z(E)$

where π_Z is the projection operator on the set of attributes Z, and $Z \subseteq Y$ in well-formed expressions.

5. Elimination of unnecessary projections

$\pi_Z(E) = E$, if Z represent the attributes of E

6. Anticipation of the projection with respect to the product (pushing down projections):

$\pi_{XY}(E1 \times E2) = \pi_X(E1) \times \pi_Y(E2)$

where X are attributes of E_1 and Y are attributes of E_2.

Relational operators: algebraic properties (not asked at the exam)

We can also demonstrate the following properties:

- ∀ $\sigma_{\phi \wedge \psi}(R) = \sigma_{\phi}(R) \cap \sigma_{\psi}(R) = \sigma_{\phi}(R) \bowtie \sigma_{\psi}(R)$;
- ∀ $\sigma_{\phi \vee \psi}(R) = \sigma_{\phi}(R) \cup \sigma_{\psi}(R)$;
- ∀ $\sigma_{\phi \wedge \neg \psi}(R) = \sigma_{\phi}(R) - \sigma_{\psi}(R)$;
- ∀ $\sigma_{\phi 1}(\sigma_{\phi 2}(R)) = \sigma_{\phi 1 \wedge \phi 2}(R)$;
- ∀ $\sigma_{\phi}(r \cup s) = \sigma_{\phi}(R) \cup \sigma_{\phi}(S)$ (*distributivity of the selection with respect to the union*);
- ∀ $\sigma_{\phi}(R - s) = \sigma_{\phi}(R) - \sigma_{\phi}(S)$ (*distributivity of the selection with respect to the difference*);

Relational operators: algebraic properties (not asked at the exam)

- ∀ $\pi_Y(\pi_X(R)) = \pi_Y(R)$ if $Y \subseteq X$;
- ∀ $\pi_X(r \cup s) = \pi_X(R) \cup \pi_X(S)$ (*distributivity of the projection with respect to the union*);
- ∀ $\pi_X(\sigma_\phi(R)) = \sigma_\phi(\pi_X(R))$, if ϕ uses only attributes *X*;
- ∀ $\sigma_\phi(r \times s) = \sigma_\phi(R) \times s$ if ϕ uses only attributes *r*;
- $r \triangleright\triangleleft (s_1 \cup s_2) = (r \triangleright\triangleleft s_1) \cup (r \triangleright\triangleleft s_2)$
- (*distributivity of the join wrt to the union*).

Algebraic Optimization

A possible algebraic optimization algorithm then proceeds with the following steps:

- Anticipation of the execution of the selections on the projections (*push*) using the rule 2 from right to left (in all other cases the rules will be used in the other direction); given that the rule is used in this direction, always applies the condition $X \subseteq Y$.
- Group the selections using the rule 1.
- Anticipation of the execution of the selections on the product using the rule 3.

Algebraic Optimization

- Repeat the previous three steps until possible.
- Eliminate unnecessary projections using the rule 5.
- Group the projections using the rule 4.
- Anticipate the execution of the projections with respect to the product repeatedly using the rule 2, but only if E is a product, and Rule 6.

In general, therefore, the result of this algorithm is an expression in which the selection and projection are performed as soon as possible, and the selection is anticipated wrt the projection.

Algebraic Optimization

Example:

Students (Name, RegNumber, Province, DoB)

Exams (Course, StudentRegNumber, Date, Mark)

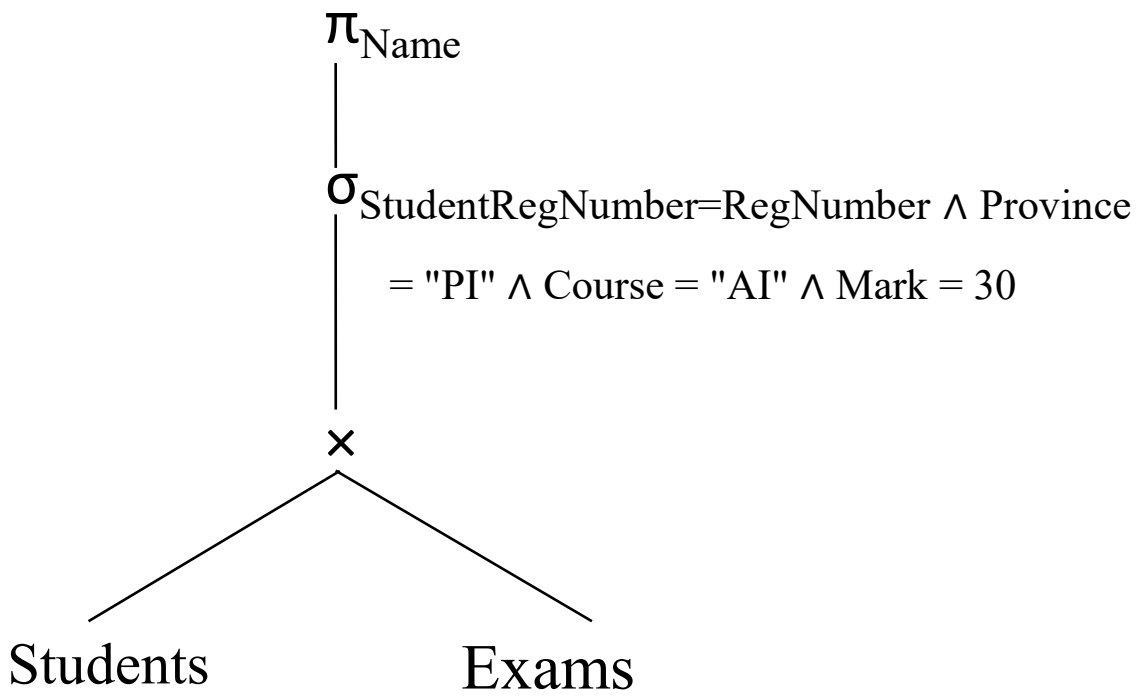
SELECT Name FROM Students, Exams
WHERE (StudentRegNumber=RegNumber AND Province = "PI" AND Course = "AI" AND Mark = 30)

$\pi_{Name}(\sigma_C(Students \times Exams))$,

where C = (StudentRegNumber=RegNumber $\wedge$ Province = "PI" $\wedge$ Course = "AI" $\wedge$ Mark = 30)

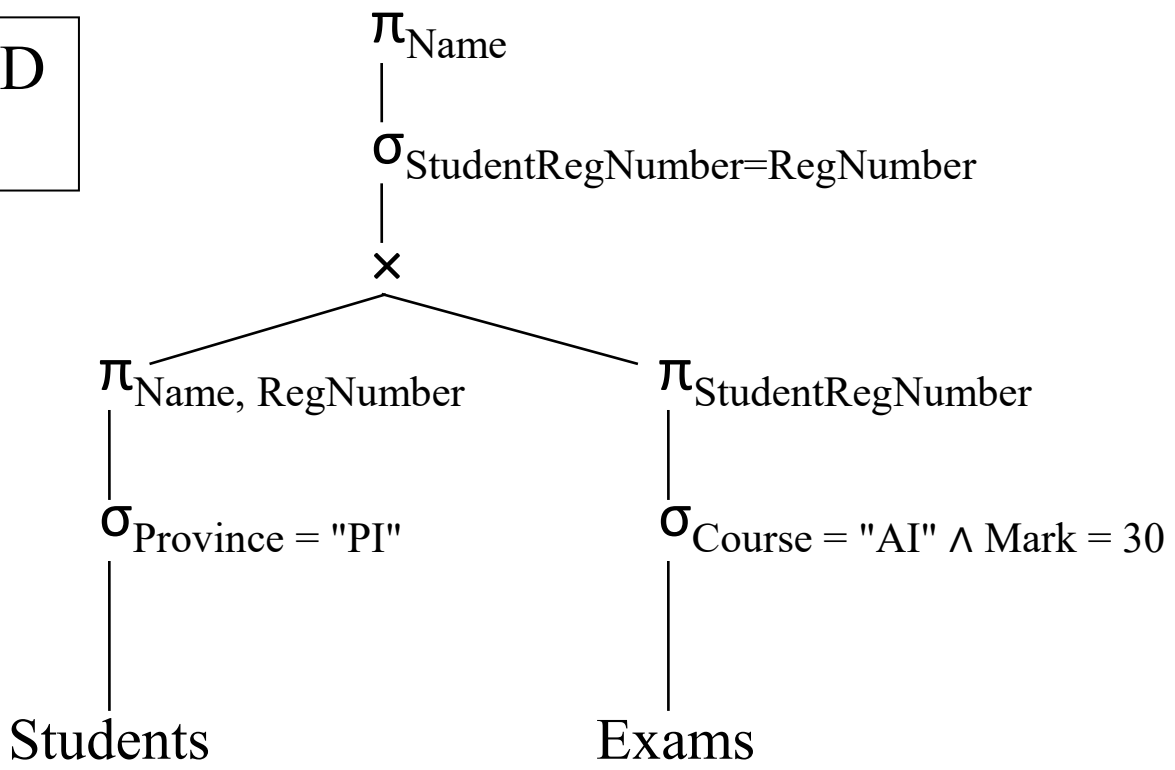
Algebraic Optimization

INITIAL TREE



Algebraic Optimization

TRANSFORMED TREE



Algebraic Optimization

Exercise.

1. Consider the relational schema

   S (M, N, P, A)

   E(C, S, V)

   Given the expression of relational algebra:

   $\sigma_{P = 'c' \wedge S = 's'}(\pi_{N, P, S}(\sigma_{M = C}(S \times E)))$

   Provide a query tree representation and show the tree after algebraic optimization.

Algebraic Optimization

2. Given the expression of relational algebra:

   $\pi_{N, S}(\sigma_{A = 'a' \wedge S = 's'}(\sigma_{M = C}(\pi_{N, M, A}(S) \times \pi_{C, S}(E))))$

   Provide a query tree representation and show the tree after algebraic optimization.

"Profiles"of relations

- The relations profiles are data structures able to maintain quantitative information:
  - *card (T)* cardinality of each relation *T*
  - *size (T)* size of the tuples in *T*
  - *size (A_j, T)* size of an attribute A_j of *T*
  - *val (A_j, T)* number of distinct values of A_j in *T*
  - *min (A_j, T)*, *max (A_j, T)* minimum and maximum value of each attribute
- They are stored in the "catalog" and updated with commands such as `update statistics`
- Used in the final stage of optimization, to estimate the size of the intermediate results

"Profiles "of relations

- **Profiles of the selections**
- The profile of a selection $T' = \sigma_{Ai = v}(T)$
- is obtained by the formulas:
- 1. $card\ (T') = (1 / val(A_i, T)) * card(T)$
- 2. $size\ (T') = size\ (T)$
- 3. $val\ (A_i, T') = 1$
- 4. $val\ (A_j, T') = card\ (T') * val\ (A_j, T) / card\ (T),\ i <> j$
- 5. $max\ (A_i, T') = min\ (A_i, T') = v$
- 6. $max\ (A_j, T') = max\ (A_j, T);\ min\ (A_j, T') = min\ (A_j, T)$

"Profiles "of relations

- **Profiles of projections**
- The profile of a selection $T' = \pi_L(T)$ where $L = A_1, A_2 \ldots, A_n$
- 1. $card(T') = min(card(T), PROD_{i=1..n}\ val(A_i, T))$
- 2. $size(T') = SUM_{i=1..n}\ size(A_i, T)$
- 3. $val(A_i, T') = val(A_i, T)$; $max(A_j, T') = max(A_j, T)$; $min(A_j, T') = min(A_j, T)$

"Profiles "of relations

- **Profiles of joins**
- The profile of a natural join $T'' = T' \triangleright\triangleleft_{A=B} T$ with *val (A, T ') = val (B, T) and A* and *B* defined on the same domain
- 1. *card (T '') = (1 / val(A, T')) * card (T ') * card (T)*
- 2. *size (T '') = size (T ') + size (T)*
- 3. *val* (A_i, *T ''), min* (A_j, *T), max* (A_j, *T)* assume the same values in the original tables

"Profiles "of relations

- Note that this analysis is entirely empirical and is based on the assumption of uniform distribution of data in tables.
- This is due to the impossibility of predicting the effect of an operation.
- In any case, from the viewpoint of the optimizer, it does not have the need for precise values, but is sufficient to reach a good degree of approximation.

Internal Query Representation

The internal representation which is given by the optimizer to the query takes into account:

- the physical structure used to implement the tables,
- the available indexes (defined on them)

The first simple transformation allows the DBMS to transform the leaf nodes in nodes that take into account the physical structures

Running Operations

- The DBMS implements relational algebra operators (or rather, combinations thereof) by means of fairly low-level operations, which, however, can implement various operators "in one shot"
- basic operators:
  - scan
  - sorting
  - direct access
  - join

Scanning (*scan*)

- A scanning operation operates by carrying out various operations in contextual manner:
  - Projection (without elimination of duplicates)
  - Selection based on a predicate
  - Insertions, deletions, and updates

Sorting

- Orderings can be carried out by:
  - Responding to specific requests of the programs (ORDER BY)
  - Proper implementation of the projections (duplicate removal) (SELECT DISTINCT)
  - Grouping (GROUP BY)
  - JOIN (not always, but is optimized)
- The algorithms used are classic algorithms adapted to work on buffers and minimize the exchange of information between the buffer and secondary storage
- In the presence of a secondary index on the sort field, sorting may be accomplished by a scan of the index leaves (B + tree)

Direct access

- It can only be executed if there are defined physical structures that support:
  - indices
  - hash structures

Direct access based on index

- Effective for queries (on the "key" of the index)
  - "Punctual" ($A_i = v$)
  - on intervals ($v_1 \leq A_i \leq v_2$)
- For conjunctive predicates
  - you choose the most selective for direct access and then verify on others after reading (and therefore in main memory)
- For disjunctive predicates:
  - Indexes on all attributes are necessary, but we should use them if very selective (could degenerate in the scan). We have to pay attention to removing duplicates

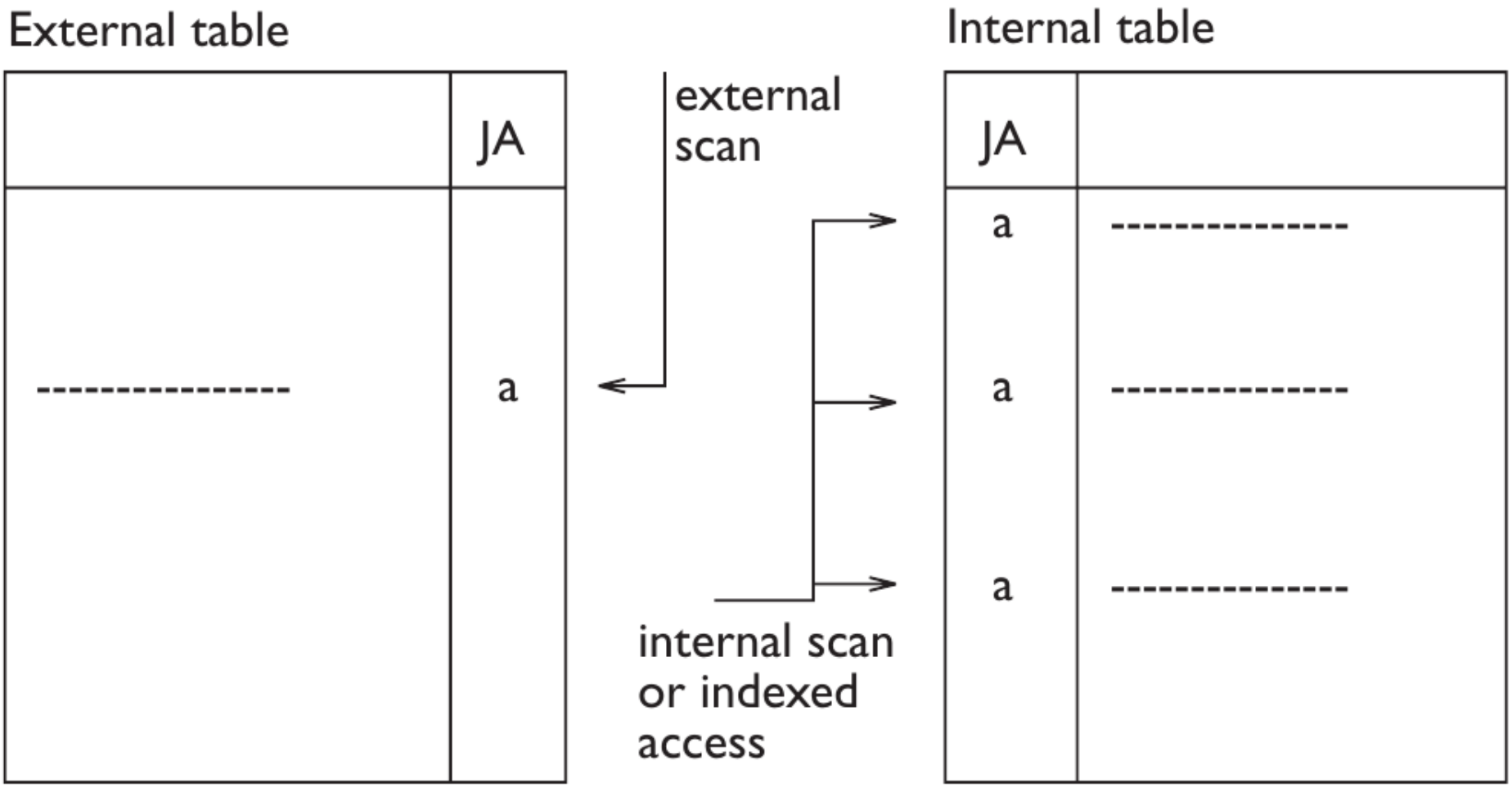
Direct access based on hash

- Effective for queries (on the "key" of the index)
  - "Punctual" ($A_i = v$) - more efficient compared to the indices
  - NOT for interval ($v_1 \leq A_i \leq v_2$)
- For conjunctive and disjunctive predicates, we can repeat the same discussion made for indexes

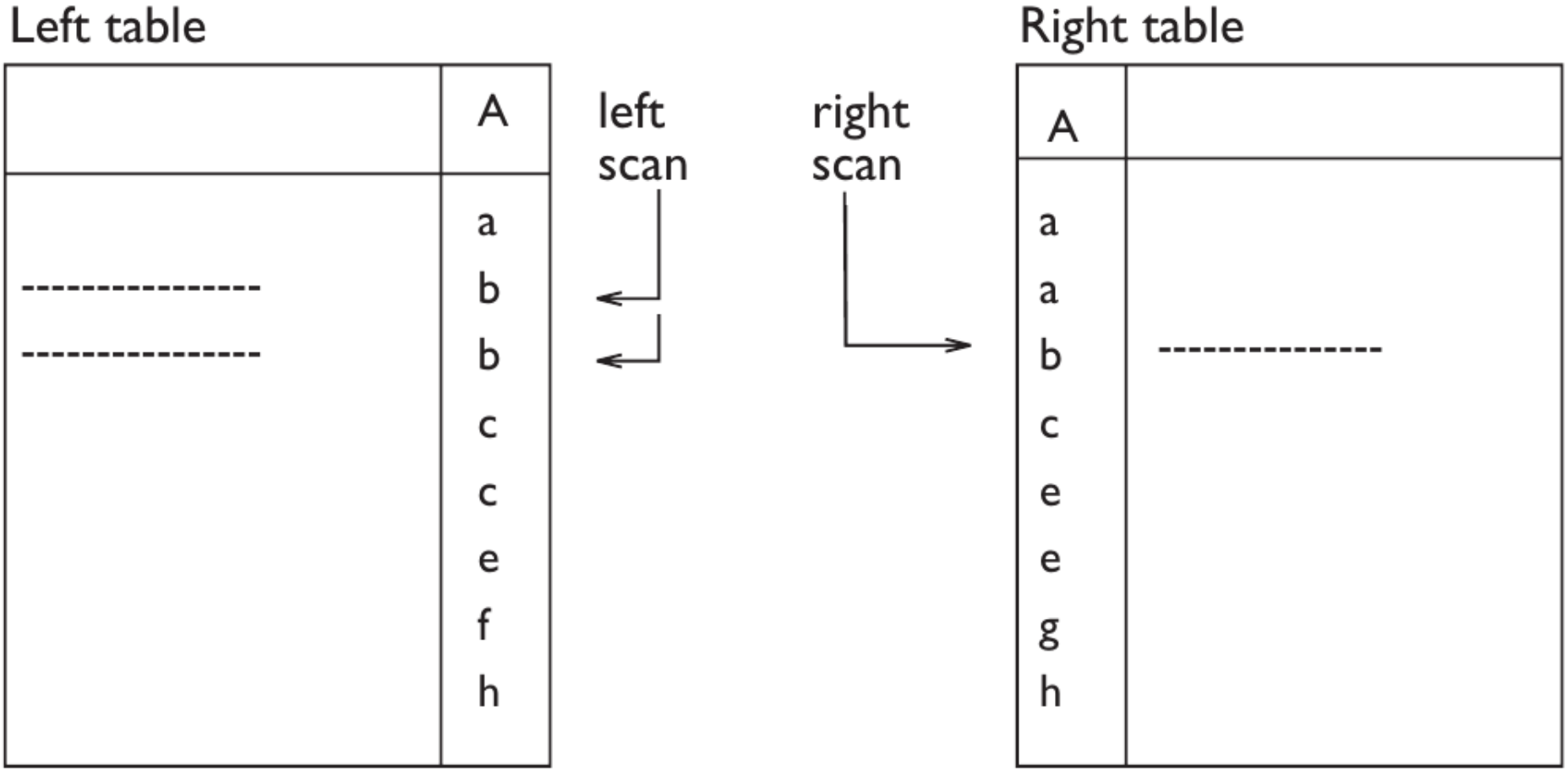
Join

- Join is the most expensive operation and can degenerate in the Cartesian product
- The three most known techniques (algorithms) for the realization of the Join are:
  - *Nested-loop*,
  - *Merge-scan an*
  - *Hash-based*

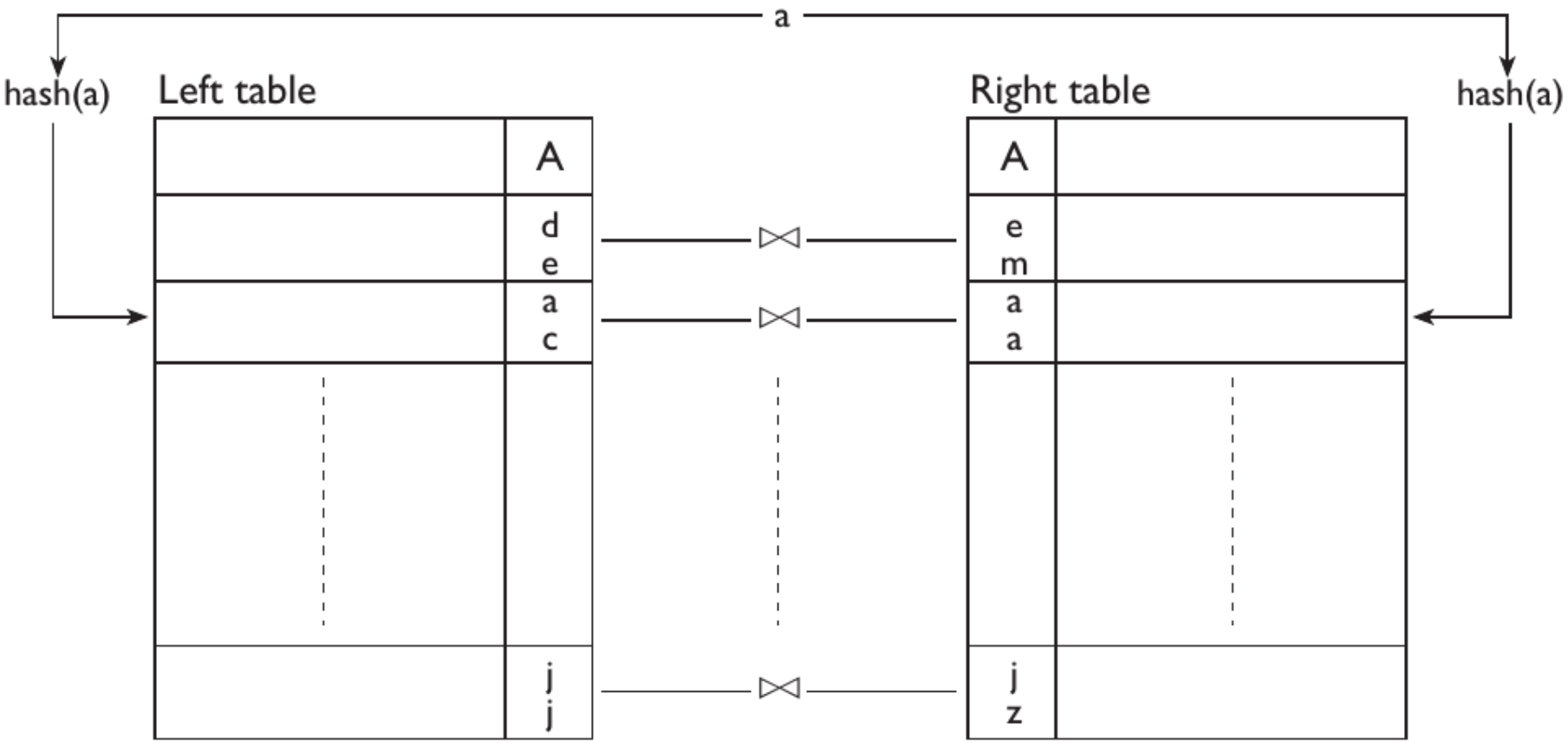
Nested-loop


External table
JA
a
Internal table
JA
a
a
a
external scan
internal scan or indexed access

Merge-scan


Left table
A
a
b
b
c
c
e
f
h
left
scan
right
scan
Right table
A
a
a
b
c
e
e
g
h

Hash join



Join of the individual collision portions (buckets)

Join

- Each of the three techniques has a cost that depends on the initial situation in which it applies:
- presence of indexes
- ability to load an operand (the join) fully buffered

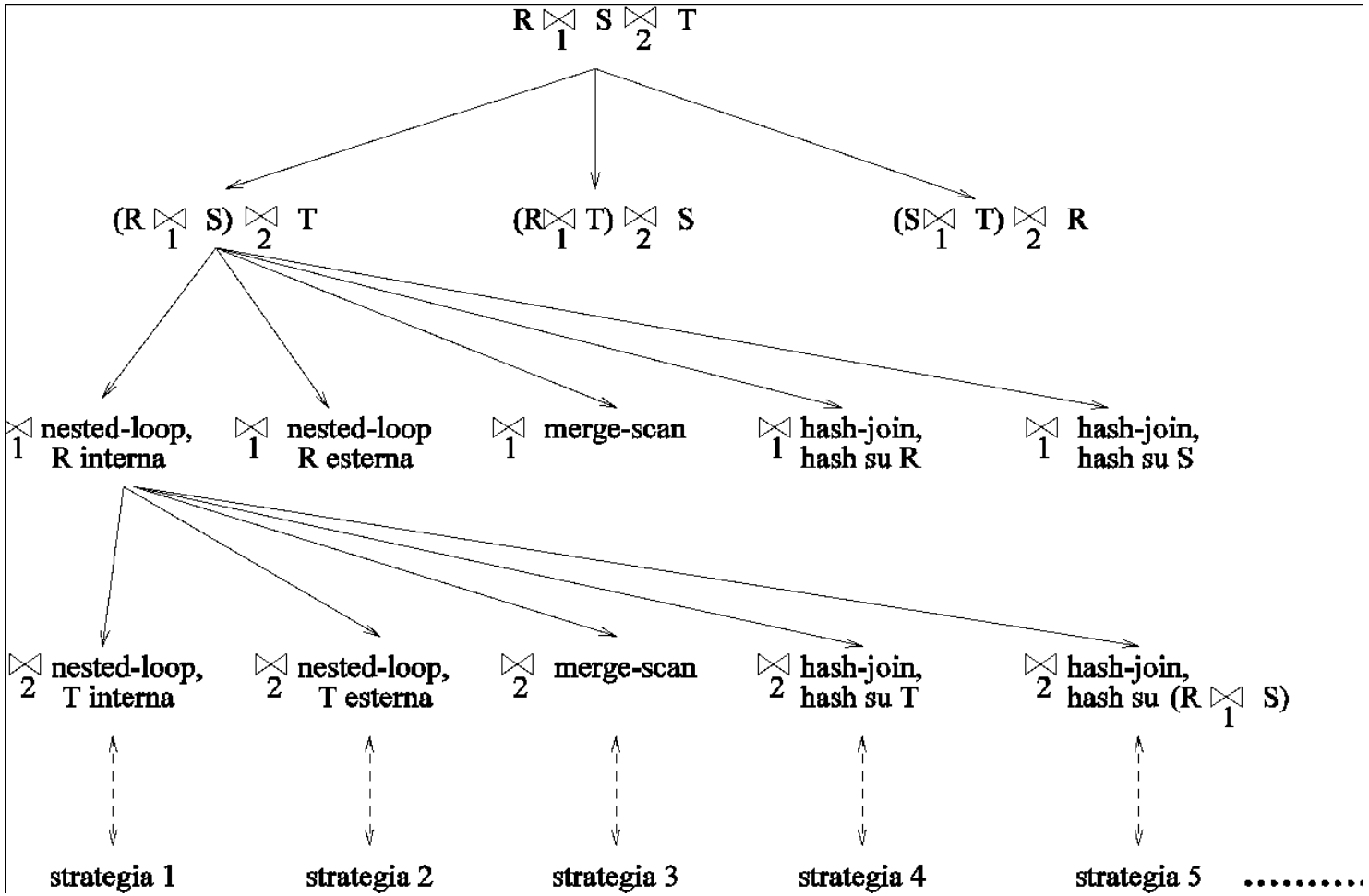
Cost-based Optimization

- A complex problem, with choices regarding:
  - operations to be performed (eg .: scanning or direct access?)
  - Order of operations (eg. join three tables; order?)
  - Details of the method (eg .: which join method?)
  - Sorting? is it necessary? If so, when it should be run?
- Parallel and distributed architectures open further degrees of freedom

Cost-based Optimization

- It builds a *tree of alternatives* in which each node defines a particular decision.
- The size of this tree increase exponentially based on the number of alternatives
- Each leaf corresponds to a specific *execution plan* described by the choices represented in the path between the root and the leaf.
- The optimizer usually looks for a "good" solution, not necessarily the "best"

A tree of the alternatives



* hash join is not symmetrical

Cost-based optimization

- The optimization is based on the identification of the "less expensive" solution
  - *Branch and bound* to delete entire subtrees
- The optimizer is usually a "good" solution, not necessarily the "best"
  - The cost of the cost-based optimization must be significantly lower than the operation
- The optimal strategy could require the construction of *ad hoc* temporary structures (eg. indexes) and the optimization cost must take into account such temporary structures.

Cost-based optimization

- The intermediate results are often allocated in buffer structures and consumed immediately after their production by exploiting the *pipelining* of operations.
- When this is not possible, it is necessary to store intermediate results in the mass memory causing a dilation of execution times.

Physical Design

- It is the final stage of the design process of a data base
- input
  - the logical schema and the application workload information
  - the specific DBMS
- output
  - physical schema, which includes the definition of the tables with the relative physical structures and the definition of many parameters, often related to the specific DBMS

Physical Design in the (object) relational model

- The common feature of (object) relational DBMS is the availability of the indexes:
  - –physical design often coincides with the choice of indexes (in addition to parameters, which are strictly dependent on the DBMS)
- The keys (primary) of relations are usually involved in selections and joins: Many systems require (or suggest) to define indexes on primary keys
  - –Other indices (or hash structures) are defined with reference to other selections or "important" joins.
- Multirelational clusters

Physical Design in the (object) relational model

- Other indexes are defined on attributes which are ordered.
- Try to avoid hash and indexes and structures when data are constantly modified
  - Update times of structures
  - Data reorganization times
  - Space for storing auxiliary structures

Physical Design in the (object) relational model: a systematic approach

- (Based on the application loading)
- Suppose we have a set of operations $op_1, op_2, \ldots, op_n$, Each with relative frequency $f_1, f_2, \ldots, f_n$.
- For each operation op_i we can define a cost c_i (Run-time), which may vary depending on the possible choices of physical structures.
- Minimizing the function
  - $\Sigma_{i = 1..n} (c_i * f_i)$

Physical Design in the (object) relational model: a systematic approach

- In fact, such a systematic approach may be difficult to manage in a complete way:
  - Many choices
  - The system may not behave as we foresee

- Regarding the second point, it must be remembered that the choices are made by an optimizer that might ignore indexes or other physical structures defined by the user.

Physical Design in the (object) relational model

- If performance is unsatisfactory, "tune" the system by adding or deleting indexes
- It is useful to verify if and how the indexes are used with the SQL command `show plan` or, in some DBMS, `explain`

Physical Design in the (object) relational model: a recommended approach

Given:

- The reference DBMS
- One logical scheme
- A set of operations *O*

Define: primary structures and indexes to optimize *O*

Recommended Steps

1- Identification of the schema of the operation, for each operation to be optimized

2- Choose the most convenient path to follow and estimate the number of rows involved in that path

3- The candidate attributes for indexing are

  - –Those involved in join operations
  - –Those involved in selections
  - –Those involved in order by and group by operations

4- Evaluation of alternatives for each attribute involved. If a qualitative assessment does not dissolve the doubts, go to an analytical assessment.

Exercise

Given: EMPLOYEE(ID, Surname, Name, Birthdate) with |EMPLOYEE|=N such that each tuple is of size L byte. Let K be the size of the key.

Let us suppose to have a DBMS which supports heap + hash and B+trees. Let also suppose that blocks are of size B and pointers have size P.

- The set of operations is:
- 1. Search by ID number (frequency=f1)
- 2. Search by surname (frequency=f2); On average, this query returns 4 tuples.

Design the physical structures for

- N =5,000,000, L = 125, K = 5, B = 1,000, P = 4,
- f1 = 100/h, f2 = 1000/h

Exercise: 1 solution

Hash on the ID and a B+tree on the surname.

- Let us suppose that the surname attribute is 20 chars long.
- F for the B+tree is 1.000/(20+4) = 41.6 ≈ 41.
  The depth of the tree for N = 5,000,000 is approximately 5 levels ($\log_{41}$ 5.000.000).

By analyzing the accesses for the operations, we have:

- 1. Operation 1 has a cost 1 (hash).
- 2. Operation 2 has a cost of 5 accesses to find the leaf, plus 4 accesses to obtain (in the primary structure) the 4 records.
- f1*1+f2*9=100*1+1000*9 = 9100 accesses per hour.

Exercise: Solution 2

A B+tree on the IDs and hash on the surnames.

- F for the B+tree is 1.000/(5+4) = 111,1 ≈ 111.
  The depth of the tree is for N = 5.000.000 is 4 levels. (≈ $\log_{111}$ 5.000.000)

This means that operation 1 has a cost of 100 * 5 accesses, that is 500 accesses per hour.

- For operation 2, we know that a block can contain 6.4 tuples (1000/125=8, with a filling factor of 80%), so to reduce conflicts, and having an average reading cost closer to 1 (*).
  This means that: the cost of operation 2 is f2*(average cost * number of returned records) = 1000 * (1*4) = 4.000 accesses.
  However, since all the 4 returned records will probably be in the same block, the actual cost will be close to 1000 (cluster organization).

The second solution has a total cost of 1500 accesses per hour.

We prefer the second solution

(*) 1.16 as for Gray & Reuter table

References

P. Atzeni, S. Ceri, P. Fraternali, S. & R. Paraboschi Torlone
Databases: Architectures and lines of evolution. Second edition.
McGraw-Hill Books Italy, 2007
Chapter 1

For more information:

Navathe, Elmasri
Fundamentals Of Database Systems -, fourth / fifth edition
Pearson Education, 2004