Contents

- Data organizations for secondary storage
  - storage structures
  - indexing structures
- Query processing
- Transactions
  - Concurrency control
  - Recovery
- Authorization and access control
- Distributed database systems and multidatabase systems

Components of a DBMS

- A DBMS must provide data management functions that are
  - efficient
  - reliable
  - concurrent
  - secure (protected)
- Each of the above requirements is supported by specific components of the DBMS that together comprise the system architecture
Components of a DBMS

Efficiency:
- **File system**: it manages the allocation of disk space and the data structures used to represent the information stored on secondary storage.
- **Buffer manager**: it is responsible for transferring information between secondary storage and main memory.
- **Query parser**: it translates the DDL and DML commands into an internal format (parse tree).
- **Optimizer**: it generates optimal execution plans for queries.

Reliability:
- **Recovery manager**: it assures that the DB is in a consistent state upon system failures and application program errors.

Concurrency:
- **Concurrency controller**: it assures that interactions among concurrent application programs do not result in an inconsistent database state.

Integrity:
- **Integrity manager**: it assures that the integrity constraints are verified.

Security:
- **Authorization manager**: it checks that accesses to the database are executed according to the authorizations.

A DBMS uses several data structures that include:
- the files storing the user data (that is, the files storing the DB).
- the files storing the system data (that is, the files storing the DBMS catalogs).
- indexes (like B-trees and hash tables).
- data statistics (for example the cardinality of each relation) that are used to determine the optimal query execution strategy.

So far we have seen high-level DBMS models, that is, we have talked about the logical level. Such level is the correct level for the DB users (end-users and applications). However an important factor is represented by the DBMS performance. The DBMS performance depends from the data structure efficiency and on the efficiency of the DBMS when operating on them.
Efficiency

- Various data structures can be used to implement a data model
- The choice of the structures to use depends on the type of accesses that are executed on the data
- A DBMS has its own strategies for implementing the data model
- However the (expert) user may influence the choices made by the system (physical level) through the use of some specialized commands (like commands for the creation of indexes and clusters)

Storage devices

- The data managed by a DBMS must be physically stored on a storage device
- Main memory and other faster smaller memories
  - Very fast access to data
  - Small storage capacity
  - Volatile – the content is lost if the system crashes

Storage devices

- Secondary storage
  - Magnetic disks, optical disks
  - Very large capacities, low costs, slow access
  - Need to transfer data in main memory in order to process them
  - Non volatile
- Databases are in most cases stored on secondary storage (magnetic disks)
  - Too large with respect to the storage capacities of main memory devices
  - Increased guarantees for data persistence
  - Very low costs

Disks

- The information is stored on the disk surface on concentric circles, each with a different diameter, called tracks; tracks are subdivided in sectors
- For disks with multiple platters, the set of tracks with the same diameter form a cylinder
- Data stored on the same cylinder can be retrieved much faster with respect to the data stored at different cylinders
- The hardware mechanism for reading and writing is the head with is connected to a mechanical arm
For multiple platter disks, there is a head for each platter. The read-write heads of all the tracks are mounted on a single assembly called a disk arm, and move together.

- Multiple disk arms are moved as a unit by the actuator.
- Each arm has two heads, to read disks above and below it.

Performance measures of disks

access time, data transfer rate, reliability

- **Access time**
  - the time from when a read or write request is issued to when data transfer begins. To access data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called seek time, and it increases with the distance the arm must move. Typical seek time range from 2 to 30 milliseconds.

- **Average seek time** is the average of the seek time, measured over a sequence of (uniformly distributed) random requests, and it is about one third of the worst-case seek time.

- **Once the seek has occurred, the time spent waiting for the sector to be accesses to appear under the head is called rotational latency time.** Average rotational latency time is about half of the time for a full rotation of the disk. (Typical rotational speeds of disks ranges from 60 to 120 rotations per second).
Performance measures of disks
access time, data transfer rate, reliability

- **data transfer rate**, the rate at which data can be retrieved from or stored to the disk. Current disk systems support transfer rate from 1 to 5 megabytes per second.

- **reliability**, measured by the mean time to failure. The typical mean time to failure of disks today ranges from 30,000 to 800,000 hours (about 3.4 to 91 years).

Disks

- Data are transferred between the disk and the main memory in units called blocks.
  - A block is a contiguous sequence of bytes from a single track of one platter.
  - Block sizes typically range from 512 to 4096 bytes.
  - The lower levels of the file system manager convert block addresses into the hardware-level cylinder, surface, and sector number.
  - The block transfer time is the time taken by the head for transferring a block in the buffer, once the head is located at the beginning of the block.
  - Such time is much shorter than the time required to position the head at the beginning of the block (seek time).

Optimization of Disk-Block Access

- Access to data on disk is several orders of magnitude slower than access to data in main memory. Optimization techniques besides buffering of blocks in main memory:
  - **Scheduling**: If several blocks from a cylinder need to be transferred, we may save time by requesting them in the order in which they pass under the heads. A commonly used disk-arm scheduling algorithm is the elevator algorithm.
  - **File organization**: Organize blocks on disk in a way that corresponds closely to the manner that we expect data to be accessed. For example, store related information on the same track, or physically close tracks, or adjacent cylinders in order to minimize seek time.
  - **Nonvolatile write buffers**: Use nonvolatile RAM (such as battery-back-up RAM) to speed up disk writes drastically (first write to nonvolatile RAM buffer and inform OS that writes completed).
  - **Log disk**: Another approach to reducing write latency is to use a log disk, a disk devoted to writing a sequential log. All access to the log disk is sequential, essentially eliminating seek time, and several consecutive blocks can be written at once, making writes to log disk several times faster than random writes.

File Organization

- Data are generally stored in records.
  - A record consists of a set of related values.
  - Each value is represented by one or more bytes and corresponds to a field of the record.

- A **record type** consists of the names of the fields in the record together with the corresponding value types.

- The number of bytes required for storing a value of a given type is fixed for each system.
File Organization

A record of the following record type
   Name of the Record Type
type Employee = record
Field Names Field Types
   Emp#: integer;
   Name: array[1..20] of char;
   Job: array[1..10] of char;
   HiringDate: date;
   Salary: integer;
   Bonus: integer;
   Dept#: integer;
can be stored in 50 bytes
(4*4=16 for the integers + 4 for the date + 10 + 20 for the strings)

Files with fixed length records

Record deletion:
- Insertions are more frequent than deletions
  - The records are not moved; the space occupied by
    the deleted record is left unoccupied and used for
    next insertion
- It is necessary to allocate some auxiliary structures (file
  header) to efficiently determine where to insert a new
  record (the file header contains the pointer to the first
  deleted record; all deleted records are linked)

Files with variable length records

Example
- Employee record type – suppose that
  - The attribute job is multivalued
  - The bonus attribute is optional and thus may be
    missing
- Different representations are possible
Files with variable length records

- **Byte string representation**
  - A special end-of-record symbol is added to denote the end of each record
  - Each record is stored as a contiguous sequence of bytes
  - Special separator characters, that do not appear as value in any field, are used to denote the end of the variable length fields

---

Files with variable length records

- **Fixed-length reserved-space representation**
  - A file with variable length records is represented through a file with fixed length records
  - For each record, the maximum length is allocated
  - Problem: poor use of space => slow accesses (the records will tend to spread over several blocks)

---

Files with variable length records

- **Byte string – main disadvantages:**
  - It is not easy to re-use the space occupied by a deleted record; there are techniques to manage insertions and deletions, but they tend to generate fragmentation
  - If the length of a record increases, the record must be moved
    - This can be expensive is the record is referenced by another record

---

Files with variable length records

**Example of byte string representation**

<table>
<thead>
<tr>
<th>ID</th>
<th>Full name</th>
<th>Position</th>
<th>Date</th>
<th>Salary</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>7308</td>
<td>Red</td>
<td>Engineer</td>
<td>Dec-17-2004</td>
<td>1600</td>
<td>X</td>
</tr>
<tr>
<td>7499</td>
<td>Andrew</td>
<td>Technician</td>
<td>Feb-20-2003</td>
<td>800</td>
<td>X</td>
</tr>
<tr>
<td>7654</td>
<td>Martin</td>
<td>Secretary</td>
<td>Aug-28-2005</td>
<td>800</td>
<td>X</td>
</tr>
<tr>
<td>7782</td>
<td>Black</td>
<td>Engineer</td>
<td>Jun-1-2004</td>
<td>2450</td>
<td>X</td>
</tr>
</tbody>
</table>
Files with variable length records

Fixed-length reserved-space representation

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Position</th>
<th>Date</th>
<th>Salary</th>
<th>#</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>Red</td>
<td>engineer manager</td>
<td>Dec-17-2004</td>
<td>1600</td>
<td>500</td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>Andrew</td>
<td>technician</td>
<td>Feb-20-2003</td>
<td>800</td>
<td>#</td>
<td>40</td>
</tr>
<tr>
<td>7654</td>
<td>Martin</td>
<td>secretary interpreter</td>
<td>Sept-28-2005</td>
<td>800</td>
<td>#</td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>Black</td>
<td>engineer</td>
<td>Jan-1-2004</td>
<td>2450</td>
<td>200</td>
<td>30</td>
</tr>
</tbody>
</table>

Files with variable length records

Fixed-length representation

- A variable length record is represented by a list of fixed length records, linked through pointers
- In order to avoid repeating the fields that have a single value, two types of block are used in a file:
  - anchor block, storing the first record of each list
  - overflow block, storing the subsequent records of each list

Organization of records in blocks

- A file can be seen as a collection of records
- Because the data are transferred between secondary storage and MM in blocks, it is important to assign the records to blocks so that a same block contains records that are related
- If records that are often used together are stored in the same block, the number of disk accesses is reduced
  - ex.: in the fixed-length organization for the variable length records, it can be useful to store all records, representing the same variable-length record, in the same block
  - If several updates are executed, a block ends up storing records from different lists
Organization of records in blocks

- It is not always possible to organize the records in blocks so that each block is completely occupied by records.
  - Very often each block has some storage which is not used.
- It is possible to store one part of a record in a block and the other part in another block.
- **Spanned organization**: each record may be stored in several blocks.
  - This cannot be avoided if the record size is greater than the block size.
- **Unspanned organization**: each record is stored in a single block.
  - It is the preferred organization for files with fixed-length records.

Techniques for the allocation of file blocks on disk:

- **Contiguous allocation**: the file blocks are allocated on contiguous disk blocks.
  - Reading the entire file is very efficient.
  - Expanding the file is difficult.
- **Linked allocation**: each file block contains a pointer to the next file block.
  - Reading the entire file is not very efficient.
  - Expanding the file is difficult.
- **Use of buckets**: (that is, of groups of blocks) for groups of records that are related (for example, all employees of the same department).

Organization of records in blocks

- It is possible to have buckets occupying several blocks: the blocks of the same bucket are linked (the block header stores the pointer to the next block).
- If the size of a bucket increases, new blocks are allocated.
  - The free blocks are linked so that they can be used when new insertions are executed in the same bucket.
- It is better not to reuse the free blocks of a bucket to store records from another bucket.
  - The blocks of the same bucket are stored in the same cylinder.

Mapping of relations onto files

- For DBMS of small size, a solution often adopted is to store each relation in a separate file.
- For large-scale DBMS such basic strategy must be extended (the DBMS must allocate the records in the blocks so to minimize the I/O operations).
  - A frequent strategy is to allocate for the DBMS a large file storing all relations.
  - Such file is managed by the DBMS.
Clustering

- Consider the following query:
  SELECT Emp#, Name, Office
  FROM Employees, Departments
  WHERE Employees.Dept# = Departments.Dept#
- An efficient storage strategy is based on clustering (that is, grouping) the tuples of the two tables that have the same value for the join attribute
- Clustering may make the execution of other queries inefficient
  - ex. SELECT * FROM Departments

Buffer management

- The main goal of the storage strategies is to minimize the number of disk accesses
- Another strategy is to keep the largest number of blocks in MM
- A buffer can be used to keep in MM several file blocks
- A buffer is organized in pages, that have the same size of the blocks
- Performing operations on pages that are already in the buffer greatly improves performance

Clustering

<table>
<thead>
<tr>
<th>10</th>
<th>Building Construction</th>
<th>1100</th>
<th>D1</th>
<th>7977</th>
</tr>
</thead>
<tbody>
<tr>
<td>7372</td>
<td>New Engineer</td>
<td>03/06/81</td>
<td>2450.00</td>
<td>500.00</td>
</tr>
<tr>
<td>7879</td>
<td>Dave Engineer</td>
<td>17/11/81</td>
<td>2600.00</td>
<td>500.00</td>
</tr>
<tr>
<td>7534</td>
<td>Mill Engineer</td>
<td>23/01/82</td>
<td>1300.00</td>
<td>150.00</td>
</tr>
<tr>
<td>7875</td>
<td>Demand Manager</td>
<td>10/12/80</td>
<td>10000.00</td>
<td>?</td>
</tr>
<tr>
<td>20</td>
<td>Research</td>
<td>10/12/80</td>
<td>1600.00</td>
<td>500.00</td>
</tr>
<tr>
<td>7370</td>
<td>Radi Engineer</td>
<td>17/12/80</td>
<td>1600.00</td>
<td>500.00</td>
</tr>
<tr>
<td>7536</td>
<td>Pink Manager</td>
<td>32/04/81</td>
<td>2975.00</td>
<td>?</td>
</tr>
<tr>
<td>7756</td>
<td>Scott Secretary</td>
<td>09/11/81</td>
<td>1000.00</td>
<td>?</td>
</tr>
<tr>
<td>7476</td>
<td>Adams Engineer</td>
<td>31/07/81</td>
<td>1100.00</td>
<td>500.00</td>
</tr>
<tr>
<td>7902</td>
<td>Ford Secretary</td>
<td>03/12/81</td>
<td>1000.00</td>
<td>?</td>
</tr>
<tr>
<td>30</td>
<td>Road Maintenance</td>
<td>31/01/81</td>
<td>1500.00</td>
<td>?</td>
</tr>
<tr>
<td>7900</td>
<td>Gianni Engineer</td>
<td>03/12/81</td>
<td>1950.00</td>
<td>?</td>
</tr>
</tbody>
</table>

Buffer management

- The buffer manager of a DBMS uses buffer management policies that are more sophisticated than those used in OS:
  - LRU policies are not always the most suitable for DBMS
  - For reasons related to recovery management, in some cases a buffer page cannot be transferred on disk (in such case the block is said to be pinned); in other cases, the block has to be forced to disk
  - A DBMS is able to predict better than the OS future references to blocks
Buffer management: Example

- Consider the join operation Employees \( \times \) Departments (assume that the two relations be in different files)
- relation Employees
  - Once that a tuple of this relation has been used, it is not any longer
  - As soon as all the tuples of a block have been processed, the block is not any longer needed
    - toss-immediate strategy

Departments relation
- The block most recently accessed will be accessed only after all the other blocks will be processed
- The best strategy for the Departments file is to remove the last used block
  - most recently used strategy - MRU
- It is however necessary pin the block being currently processed until all tuples in the block have been analyzed; once all tuples have been analyzed, the block can be unpinned

Auxiliary access structures

- Very often queries only access a small subset of the data
- To efficiently process queries, it can be useful to allocate some additional data structures (called auxiliary access structures) able to directly find the records that verify a given query (without requiring a scan of all the data)

Each technique must be evaluated with respect to:
- Access time
- Insertion time
- Deletion time
- Storage size
- Very often it is preferable to increase storage occupancy if this improves performance
- We use the term search key to denote an attribute or attribute set used for the search in the data structure (different from the notion of primary key of the relational data model)
Auxiliary access structures

A search can be performed based on:
- **primary key**: the value of the key identifies a unique record
  - Ex. The taxpayer with SSN 567-34-9087
- **secondary key**: the value of the key can identify multiple records
  - Ex. The taxpayers living in West Lafayette
- **value range** (both for the primary key and secondary key)
  - Ex. The taxpayers with an income between 60K and 90K
- **combination of the previous conditions**
  - Ex. The taxpayers living in West Lafayette with an income between 60K and 90K

To perform efficient searches, one could maintain the file ordered according to the values of a search key.

However, searchers based on other search keys are inefficient.

**Primary organizations**: The impose an allocation criteria for the data (for example, hash functions)

**Secondary organizations**: Use of indexes (separated from the data file) usually organized according to a tree

In general, several access paths to the data are present.
Indexes

- **basic idea**: to associate with a file, a “table” containing entries of the form: 
  \((k_i, r_i)\) where:
  - \(k_i\) is a value of the search key on which the index is built
  - \(r_i\) is a reference to the record (or records) having \(k_i\) as value of the search key
  - The reference can actually be the address (logical or physical) of a record or of a block

Indexes: example

- **Data file**
  - keys: \(c5\), \(c2\), \(c11\), \(c7\), \(c4\)
  - Addresses: 0, 8, 16, 32, 48

- **Index**
  - key \(k_i\)
  - address \(r_i\)
    - \(c2\): 8
    - \(c4\): 48
    - \(c5\): 0
    - \(c7\): 32
    - \(c11\): 16

Indexes

- The various techniques differ with respect to the approach they use for organizing the set of pairs \((k_i, r_i)\)
- Advantages in the use of an index:
  - They key is only a (small) part of the information contained in the records; therefore requires less storage than the data file
  - An index can however have very large sizes that introduce management problems like those for files

- Example:
  - The index for a file containing 50k records, with search keys of size equal to 20 bytes, and pointers of size equal to 4 bytes, requires about 1.2Mb
    - Size of the each index entry: 20 + 4 bytes
    - Max number of entries: 50k
    - Total: \(24 \times 50K = 1.2\) Mb
Types of index

- The index organizations can be classified with respect to various dimensions:
  - Uniqueness of the values of the search key
  - Ordering of the records in the data file
  - Number of the index entries
  - Number of levels

Uniqueness of key values

- **Index on primary key:**
  - Index on an attribute which is primary key (or key) of the relation; each entry of the index corresponds to a single record in the file

- **Index on secondary key:**
  - Index on an attribute which is not a primary key (or key) of the relation; each entry of the index may correspond to multiple records in the file

Ordering of records in the data file

- **Clustered index** (also referred to as primary index):
  - An index allocated on the attribute according to which the data file is ordered

- **Unclustered index** (also referred to as secondary index):
  - An index allocated on an attribute according to which the data file is not ordered

Number of pairs in the index

- **Dense index**
  - It is an index such that the number of entries \((k_i, r_i)\) is equal to number of values of the search key

- **Sparse index**
  - It is an index such that the number of entries \((k_i, r_i)\) is lower than the number of values of the search key
Number of levels

- **Single level index**
  - An index organized according to a single level
- **Multiple level index**
  - An index organized according to multiple levels

Indexes sparse and dense

- **Dense index:** the index contains an entry for each value of the search key
  - search: the index is searched to retrieve the record with the key value equal to the searched value
  - The data are retrieved from the file
- **Sparse index:** the entries of the index are created only for some of the key values.
  - search: the index is searched until a record is found that has a key value which equal or lower than the searched value
  - The file is sequentially scanned until the searched record is found; then the data are retrieved

Dense index

<table>
<thead>
<tr>
<th>Name</th>
<th>Position</th>
<th>Date</th>
<th>Salary 1</th>
<th>Salary 2</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782 Neri</td>
<td>Engineer</td>
<td>01/06/81</td>
<td>2450.00</td>
<td>200.00</td>
<td>10</td>
</tr>
<tr>
<td>7934 Mill</td>
<td>Engineer</td>
<td>17/12/80</td>
<td>1600.00</td>
<td>300.00</td>
<td>20</td>
</tr>
<tr>
<td>7877 Green</td>
<td>Manager</td>
<td>01/12/81</td>
<td>950.00</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>7566 Pink</td>
<td>Manager</td>
<td>01/12/81</td>
<td>950.00</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>7688 Scott</td>
<td>Secretary</td>
<td>09/11/81</td>
<td>800.00</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>7499 Andrews</td>
<td>Technician</td>
<td>20/02/81</td>
<td>800.00</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>7521 White</td>
<td>Technician</td>
<td>20/02/81</td>
<td>800.00</td>
<td>100.00</td>
<td>30</td>
</tr>
</tbody>
</table>

Sparse index

<table>
<thead>
<tr>
<th>Name</th>
<th>Position</th>
<th>Date</th>
<th>Salary 1</th>
<th>Salary 2</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782 Neri</td>
<td>Engineer</td>
<td>01/06/81</td>
<td>2450.00</td>
<td>200.00</td>
<td>10</td>
</tr>
<tr>
<td>7934 Mill</td>
<td>Engineer</td>
<td>17/12/80</td>
<td>1600.00</td>
<td>300.00</td>
<td>20</td>
</tr>
<tr>
<td>7877 Green</td>
<td>Manager</td>
<td>01/12/81</td>
<td>950.00</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>7566 Pink</td>
<td>Manager</td>
<td>01/12/81</td>
<td>950.00</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>7688 Scott</td>
<td>Secretary</td>
<td>09/11/81</td>
<td>800.00</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>7499 Andrews</td>
<td>Technician</td>
<td>20/02/81</td>
<td>800.00</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>7521 White</td>
<td>Technician</td>
<td>20/02/81</td>
<td>800.00</td>
<td>100.00</td>
<td>30</td>
</tr>
</tbody>
</table>
Indexes sparse and dense

- A dense index supports a more efficient search, but it has higher update costs.
- A sparse index is less efficient but has lower update costs.
- Because very often, the strategy is to minimize the number of block transferred, a solution often adopted is to insert an entry in the index for each block.

Multilevel indexes

- A index, even if it sparse, may have very large dimensions.
- Example:
  - A file of 100000 records, with 10 records for each block, requires an index with 10000 entries.
  - If each block contains 100 entries of the index, the index will require 100 blocks.

Multilevel indexes

- If the index is small, it can be kept in main memory.
- Very often however, it is necessary to keep the index on disk; in such case, an index scan may require a large amount of block.
  - If a binary search is used, the index in our example would require transferring 7 blocks.
- Solution:
  - The index is handled as a file and a sparse index is allocated on the index itself.
  - In such case, the index is called a two-level sparse index.

Multilevel indexes

- If the external index is kept in MM, only a single block must be accessed.
Clustered indexes

- If several indexes are to be allocated on a given file, the index the key of which is used to order the file, is called primary index, or clustered index.
- All other indexes are called secondary indexes, or unclustered indexes.
- The secondary indexes are dense; the file is not ordered with respect to the keys of these indexes.
- The use of multiple secondary indexes makes query execution more efficient, but it increases the update costs.

Techniques

- The data structures for secondary storage differ with respect to those for main memory because the main requirement is to minimize the number of block transfers (that is the main factor in the search cost); possible organizations:
  - Based on trees (B-tree and its variations)
  - Based on hash tables

B-trees

- The B-trees are organizations based on the balanced tree data structure.
- Fundamental requirements for secondary storage indexes:
  - balance: the index must be balanced with respect to the blocks and not with respect to the single nodes (the number of accessed blocks determines the I/O cost of the search operation).
  - minimal storage: it is important to establish a lower bound to the utilization of blocks, so as to avoid the under-utilization of storage.
  - update efficiency: the update operations must have a limited cost.

B-trees

- The B-tree guarantees a storage utilization of at least 50% (at least half of each allocated page actually stores index entries).
- In a B-tree, the cost of search is, even in the worst case, logarithmic in the index cardinality (that is, the number of search keys).
- In a B-tree, each node at $m$ children, where $m$ is the only parameter depending on the specific characteristics of storage, that is, the block dimension.
A B-tree of order \( m \) \((m \geq 3)\) is a balanced tree that verifies the following properties:
- each node contains at most \( m - 1 \) elements
- each node contains at least \( \lceil m/2 \rceil - 1 \) elements,
- The root may contain a single element

Each non leaf node containing \( j \) elements has \( j + 1 \) children

Each node has a structure of the form:

\[
p_0(k_1, r_1)p_1(k_2, r_2)p_2 \ldots p_{j-1}(k_j, r_j)p_j
\]

where \( j \) is the number of the elements in the node

In each node

\[
p_0(k_1, r_1)p_1(k_2, r_2)p_2 \ldots p_{j-1}(k_j, r_j)p_j
\]

\( k_1, \ldots, k_j \) are ordered key values, that is, \( k_1 < k_2 < \ldots < k_j \)

There are \( j + 1 \) references to children nodes \( p_0, \ldots, p_j \) and \( j \) references to the data file \( r_1, \ldots, r_j \)

Let \( K(p_i) \) \((i = 1, \ldots, j)\) denote the set of key values stored in the subtree with root \( p_i \), for each non-leaf node we have that:
- \( \forall y \in K(p_0), y < k_1 \)
- \( \forall y \in K(p_i), k_i < y < k_{i+1}, i = 1, \ldots, j - 1 \)
- \( \forall y \in K(p_j), y > k_j \)

Format of a node of a B-tree:

- Pointer to a tree node
- Pointer to the data file

Example of B-tree with \( m=5 \):

- Pointer to a tree node
**B-trees**

- **Height** = number of nodes to be traversed in a path from the root to a leaf node
- the B-trees allow one to estimate with good approximation with average height of a tree based on the number of key values
  - Therefore it is easy to estimate the search costs
- \( b_{\text{min}} \) = minimum number of nodes that a B-tree of height \( h \) may store
- \( b_{\text{max}} \) = maximum number of nodes that a B-tree of height \( h \) may store

**Minimum number of nodes in a tree with height \( h \)**

\[
\begin{align*}
\text{Minimum number of nodes} & = 1 + 2 + 2\left\lfloor \frac{m}{2} \right\rfloor + 2\left\lfloor \frac{m}{2} \right\rfloor + 2\left\lfloor \frac{m}{2} \right\rfloor + \ldots + 2\left\lfloor \frac{m}{2} \right\rfloor \\
& = 1 + 2\left\lfloor \frac{m}{2} \right\rfloor^{h-1} - 1 \\
& = \frac{m}{2} - 1
\end{align*}
\]

we recall that

\[
\sum_{x=0}^{k-1} x^k = \frac{x^k - 1}{x - 1}
\]
B-trees

- Maximum number of nodes
  \[ b_{\text{max}} = 1 + m + m^2 + \ldots + m^{h-1} = \frac{m^h - 1}{m - 1} \]

- \( N_{\text{min}} \) = minimum number of key values in a tree of height \( h \)

- \( N_{\text{max}} \) = maximum number of key values in a tree of height \( h \)

B-trees

- Minimum number of key values: the number of nodes is \( b_{\text{min}} \) and thus each node contains \( \lceil m/2 \rceil \) - 1 key values and the root contains a single key value
  \[ N_{\text{min}} = 1 + (\lceil m/2 \rceil - 1)(b_{\text{min}} - 1) = 1 + (\lceil m/2 \rceil - 1)2^{\lceil m/2 \rceil - 1} - 1 = 2^{\lceil m/2 \rceil - 1} - 1 \]

B-trees

- Maximum number of key values: the number of nodes is \( b_{\text{max}} \) and thus each node, included the root, contains \( m - 1 \) keys
  \[ N_{\text{max}} = (m - 1)b_{\text{max}} = (m - 1)\frac{m^h - 1}{m - 1} = m^h - 1 \]

B-trees

- let \( N \) denote the number of key values in a B-tree, we have that:
  \[ 2^{\lceil m/2 \rceil - 1} - 1 \leq N \leq m^h - 1 \]
  \[ \log_a (N + 1) \leq h \leq 1 + \log_{[m/2]} \frac{N + 1}{2} \]
B-trees

- Height of a B-tree in terms of the number of nodes and the order; assume that key values are 10 bytes long and the pointers are 4 bytes long.

<table>
<thead>
<tr>
<th>Page size</th>
<th>N = 1000</th>
<th>N = 10000</th>
<th>N = 100000</th>
<th>N = 1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>36</td>
<td>1.9</td>
<td>3.2</td>
<td>3.9</td>
</tr>
<tr>
<td>1024</td>
<td>73</td>
<td>1.5</td>
<td>2.7</td>
<td>3.1</td>
</tr>
<tr>
<td>2048</td>
<td>146</td>
<td>1.4</td>
<td>2.4</td>
<td>3.0</td>
</tr>
<tr>
<td>4096</td>
<td>292</td>
<td>1.2</td>
<td>2.2</td>
<td>2.6</td>
</tr>
</tbody>
</table>

B-trees: search operation

- Once the root is transferred to main memory, a search is performed on the key values in the root, until the presence or absence from the node of the key value is determined.
  - If the key value is not found, the search is continued in the only subtree of the current node that may contain the key value.
  - If the node is a leaf node, then the key value is not present in the tree.
- The cost of the search is given by the number of nodes that are read, that is:
  \[ C_{\text{search}}^{\text{min}} = 1 \]
  \[ C_{\text{search}}^{\text{max}} = h \]

B-trees: insertion

- Main idea for insertion and deletion operations:
  - The updates always start from the leaf nodes and the tree grows or shrinks from the bottom of the tree.
- Example: insertion
  - No new children nodes of the leaf nodes are created; rather an new leaf node, that the same level of the existing ones, is created and a key value (separator) is propagated to the next level (toward the root).
  - The nodes at the next level are not necessarily full and then can store information propagated from the leaf nodes.
  - The propagation of the update effects until the root may result in an increase of the height of the tree.
B-trees: insertion

- The insertion requires first of all a search operation to verify whether the key value is already present in the tree.
- The insertion is always performed on a leaf node – two cases can arise:
  - If the leaf node is not full, the key value is inserted and the updated leaf node is re-written.
  - If the leaf node is full, a splitting process starts that can propagate to next level and, in the worst case, can reach the root.

B-trees: insertion - splitting

- Let P be a full node in which a key value must be inserted.
- Ordered sequence of m entries that would be created with the new key value:
  \[ p_0 \rightarrow k_0 \rightarrow p_1 \rightarrow k_1 \rightarrow p_2 \rightarrow \ldots \rightarrow k_m \rightarrow p_{m+1} \]
  with \( g = \left\lceil m/2 \right\rceil - 1 \)
- The keys are partitioned as follows:
  - Node P stores the elements:
    \[ p_0 \rightarrow k_0 \rightarrow p_1 \rightarrow k_1 \rightarrow p_2 \rightarrow \ldots \rightarrow k_g \]
  - A new node P' stores the elements:
    \[ p_{g+1} \rightarrow k_{g+1} \rightarrow p_{g+2} \rightarrow \ldots \rightarrow k_m \rightarrow p_{m+1} \]

B-trees: insertion - splitting

- In the node Q, father of P, the following entry is inserted:
  \[ k_{g+1} \rightarrow p' \]
  where \( p' \) is the pointer to the new node P'.
- If also Q is full, the splitting process must be repeated.
- If the root must be split, the new root becomes:
  \[ p_{g+1} \rightarrow p_0 \rightarrow p_1 \rightarrow \ldots \rightarrow p_m \]
  where \( p \) is the pointer to node P (the old root node).
B-trees: insertion - costs

- Best case (no splitting):
  - $h$ nodes are read and a leaf node is re-written, thus
  $$C_{\text{insertion}}^{\text{min}} = h + 1$$
- Worst case (the splitting propagates up to the root):
  - $h$ nodes are read and $2h+1$ are re-written, thus
  $$C_{\text{insertion}}^{\text{max}} = 3h + 1$$

B-trees: deletion

- Also in this case the process starts from a deletion from a leaf node
- First of all a search is performed to determine whether the node storing the element to be deleted
  - If the element is not stored in a leaf node, then it is replaced with an element such that its key value is the smallest value in the subtree pointed by the pointer to the right of the element to be deleted

B-trees: deletion

- Deletion of an element from a leaf node – there are two cases:
  - If the leaf node is not in underflow (that is, it has at least $\lceil m/2 \rceil - 1$ elements after the deletion), the key is deleted and the updated leaf node is re-written
  - If the leaf node is in underflow, a concatenation process or redistribution process is started

B-trees: deletion

- The concatenation of two adjacent nodes $P$ and $P'$ is possible if the two nodes contain on the overall less than $m - 1$ keys
- A node with less than $\lceil m/2 \rceil - 1$ elements, that is, in underflow, is combined with an adjacent node with at most $\lfloor m/2 \rfloor$ keys
- the concatenation process is the opposite of the splitting process
B-trees: deletion
concatenation

- Initial situation:
  - node $P$ in underflow with elements:
    $$p_0k_1p_1k_2p_2 \ldots k_ep_e$$ ($e = \lceil m/2 \rceil - 2$)
  - node $P'$ adjacent to the right of $P$ with elements:
    $$p'_0k_{e+1}p_{e+1} \ldots$$
  - node $Q$, father of $P$ and $P'$, with elements
    $$\ldots k_{t-1}p_{t-1}k_tpt_{t+1} \ldots$$
    where $p_{t-1}$ is a pointer to $P$ and $p_t$ is a pointer to $P'$

B-trees: deletion
redistribution

- The deletion of the key value $k_t$ from the father node may in turn trigger a concatenation (or a redistribution)
- the concatenation may propagate to the root, resulting in the deletion of the root node and thus in a decrease of the tree height

B-trees: deletion
concatenation

- the concatenation of two adjacent nodes results in the following situation:
  - node $P$ with elements:
    $$p_0k_1p_1k_2p_2 \ldots k_ep_e$$
  - node $P'$ with elements:
    $$p'_0k_{e+1}p_{e+1} \ldots$$
  - node $Q$ with elements:
    $$\ldots k_{t-1}p_{t-1}k_{t+1}p_{t+1} \ldots$$
    where $p_{t-1}$ is a pointer to $P$

B-trees: deletion
redistribution

- If two adjacent nodes cannot be concatenated, then the elements of the two nodes can be redistributed
- The redistribution operation involves also the father node because one of its elements must be modified; however the number of its elements is not modified and therefore there is no propagation to the next level
B-trees: deletion redistribution

- Initial situation:
  - node $P'$ in underflow with elements:
    $p'_0k'_1p'_2 \ldots k'_jp'_j$ ($j = \lceil m/2 \rceil - 2$)
  - node $P$ adjacent to $P'$ on the right with elements:
    $p_0k_1p_2 \ldots k_p$ $p_e$
  - node $Q$, father of $P$ and $P'$, with elements:
    $\ldots k_{t-1}p_{t-1}k_{t}p_{t}k_{t+1}p_{t+1} \ldots$

- Where $p_{t-1}$ is a pointer to $P'$ and $p_t$ a pointer to $P$.

To redistribute the elements in the two nodes:
- Consider the list of key values $k'_1 \ldots k'_j k_t k_1 \ldots k_m$.
- The first $\lfloor (t+j)/2 \rfloor$ key values are left in $P'$.
- In the father node the key $k_t$ is replaced by the key value of position $\lfloor (t+j)/2 \rfloor + 1$.
- The remaining $\lceil (t+j)/2 \rceil$ key values are inserted in $P$.

(a) Concatenation [deletion of 8]
(b) Redistribution [deletion of 3]

- Best case (the deletion is performed in a leaf node and no concatenation nor redistribution are needed):
  - $h$ nodes are read and a leaf node is re-written, thus $C_{\text{deletion}}^{\text{min}} = h + 1$
- Worst case (all nodes along the search path must be concatenated, with the exception of the first two; therefore for the child of the root a redistribution operation must be performed and the root has to be modified):
  - $2h - 1$ nodes are read and $h+1$ nodes are re-written, thus $C_{\text{deletion}}^{\text{max}} = 3h$
B-trees: deletion details of the worst case

- R: C(search) + C(excluding the root) + C(root)
- W: C(concatenation) + C(redistribution) + C(root)

hp1: each read operation requires a I/O operation
- R: h + 3(h-1) + 1 = 4h - 2
- W: (h-2) + 2 + 1 = h+1

hp2: if I assume to concatenate always to right or left:
- R: h + 2(h-1) + 1 = 3h - 1
- W: (h-2) + 2 + 1 = h+1

hp3: the accessed nodes remain in main memory + hp2
- R: h + (h-1) + 0 = 2h -1
- W: (h-2) + 2 + 1 = h+1

B-tree: redistribution in data insertion

- The structure of a B-tree depends from the order according to which the data are inserted
- If the key values were inserted according to their relative order, we would have a B-tree with all leaf nodes filled at a half, possibly except for the last node
- To improve storage utilization:
  - The redistribution during the insertion is used: instead of splitting a full node, a redistribution is executed with an adjacent node, until such node is full
  - Such an approach generates trees with nodes storing more keys, but it increases the insertion costs

B+-trees

- A B-tree is very efficient with respect to search and modification operations that involve a single record
  - For example, let m = 100 and N = 1000000; in such case the search of a key requires at most 4 disk accesses
- A B-tree however is not particularly suited for sequential operations nor for range searches
  - The retrieval of the next key value may requires accessing a large number of nodes
- To address such problem a variation to the B-tree structure has been proposed, known as B+-tree

B+-trees

- Main idea: in a B-tree, the key values have two functions:
  - Separators: to determine the path to follow during the search
  - Key values: to allow accessing the information associated with them (that is, the pointers to the data)
- In a B+-tree such functions are kept distinct:
  - The leaf nodes contain all the key values (and the associated information)
  - The internal nodes (organized as a B-tree) store some separators which have the only function of determining the path to follow when searching for a key value
B+-trees

- In addition the leaf nodes are linked in a list, in order to efficiently support range searches or sequential searches (there is also a pointer to the first element of such list to support fast accesses to the minimum key value)
- partial duplication of the keys
  - The index entries (keys and data references) are only stored in the leaf nodes
  - A search for a given key value must always determine a leaf node
- The subtree on the left side of a separator contains key values that are lower than the separator; the subtree on the right side of a separator contains key values which are greater or equal than the separator
- In the case of alphanumeric keys, one can reduce the space requirements by using separators that have reduced lengths

A B+-tree of order $m \geq 3$ is a balanced tree that verifies the following properties:
- each node contains at most $m - 1$ elements
- each node, except the root, contains at least $\lceil m/2 \rceil - 1$ elements; the root may contain a single element
- Each non leaf node containing $j$ elements has $j + 1$ children
- Each leaf node has the following structure
  
  $(k_1, r_1)(k_2, r_2) \ldots (k_j, r_j)$

  where:
  - $j$ is the number of the elements of the node
  - $k_1, \ldots, k_j$ are ordered key values, that is, $k_1 < k_2 < \ldots < k_j$
  - $r_1, \ldots, r_j$ are references to the data file
- Each leaf node has a pointer to the next leaf node and to the previous leaf node
Each non leaf node has the following structure:

\[ p_0k_1p_1k_2p_2 \ldots p_{j-1}k_jp_j \]

where:
- \( j \) is the number of the elements of the node
- \( k_1, \ldots, k_j \) are ordered key values, that is, \( k_1 < k_2 < \ldots < k_j \)
- \( p_0, \ldots, p_j \) are \( j+1 \) references to the children nodes

For each non leaf node, let \( K(p_i) \) (\( i = 0, \ldots, j \)) be the set of key values stored in the subtree with root \( p_i \), we have:
- \( \forall y \in K(p_0), y < k_1 \)
- \( \forall y \in K(p_i), k_i \leq y < k_{i+1}, i = 1, \ldots, j - 1 \)
- \( \forall y \in K(p_j), y \geq k_j \)
- each \( k_i \), for \( i = 1, \ldots, j \), is the minimum element of \( K(p_i) \)

Example of B+-tree with order 5
**B+-trees vs B-trees**

- assumption: nodes have the same size in both organization
- The search of a single key value is in general more expensive in a B+-tree (we have always to reach a leaf node to fetch the pointer to the data file)
- for operations requiring an ordering of the retrieved records according to the search key values or for range queries, the B+-tree is to be preferred (the links among the leaf nodes eliminates the need of accessing the intermediate nodes)
- the B-tree requires less storage (the key values are stored only once)

**B- and B+-trees: terminology**

- **Order of a B-tree**
  - For us: maximum number of children of a node
  - variation: minimum number of keys that a node may store
    (used in the original definition)
  - Some textbooks call B-trees what we have called B+-trees
  - Other textbooks call B*-tree a variation of the B-tree where the storage utilization for nodes must be at least 66% instead of 50%

---

**Hash organizations**

- The hash organizations are mainly used primary data organizations
- The use of indexes has the disadvantage that a scan of an additional data structure has to be executed to retrieve the data; this is because the association (key, address) is explicitly maintained
- A hash organization use a hash function H that transforms each key value into an address
- To perform a search, given a key value k, one has simply to compute H(k)

**Hash organizations**

- Each address generated by the hash function determines a logical page, or bucket
- the number of elements that can be stored in the same bucket determines the capacity c of the buckets
- If a key is assigned to a bucket that already contains c keys, we have an **overflow**
- The presence of overflows may require the use of a separate storage area, called **overflow area**
- The storage area comprising all buckets that can be addressed by the hash function **primary area**
A hash function is **perfect** if for a given set of keys does not generate overflows.

A perfect function can always be defined by allocating a primary storage area able to store all possible records.

Given a finite alphabet containing $V$ symbols, $V^L$ is the cardinality of the set of keys of length $L$ that can be defined on such alphabet.

Let $N$ be the number of records to be stored, $N/V^L$ is called **density of the active keys**.

In general, the density of the active keys is low.

A hash function generates $M$ addresses $(0, \ldots, M - 1)$ where $M$ is the number of the buckets in the primary area.

**Static organization**: the value of $M$ is constant (dimensioning the primary area is a crucial part of the design of the organization).

**Dynamic organization**: the primary area grows or shrinks in order to adapt to the effective data volume (several hash functions are used).

**Load factor**: (the ratio between the number of active keys and the maximum number of records that can be stored)

$$d = \frac{N}{MC}$$

The design of a data organization based on the use of hash functions requires to specify:

- The key transformation function
- The strategy for managing the overflows
- The load factor
- The page size

A transformation function is an application $H$ from the set of possible key values to the set $0, \ldots, M - 1$ of the possible addresses that verifies the following properties:

- **Uniform distribution** of the key values in the address space (each address must be generated with the same probability)
- **Random distribution** of the key values (possible correlations among the key values must not result in correlations among the generated addresses)

Such properties depend from the considered set of keys and therefore there is not optimal transformation function.
The performance of a hash function varies depending on the set of key values on which the function operates.

We will consider hash functions defined for key values that are integer numbers.

If the key values are alphanumeric strings, one has to associate with each key a unique integer number before applying the transformation.

**Division method**: the numeric key is divided by a number \( P \) and the address is obtained by taking the integer remainder:

\[ H(k) = k \mod P \]

where \( \mod \) denotes the integer remainder.

In order to choose \( P \) the following guidelines can be followed:

- \( P \) is the first prime number which is lower or equal to \( M \), or \( P \) is non prime, lower or equal to \( M \), without any prime factor lower than 20.
- If \( P < M \) then we should have \( M := P \).

Experiments carried out on files storing data with different characteristics show that the division method is the one that adapts best.

Other possible hash functions are:

- **mid square**: the key value is multiplied by itself, a number of digits equal to \( M - 1 \) is extracted from the central part of the result of the product, and the obtained number is normalized to \( M \).
- **shifting**: the key values are split in a number of parts, each consisting of a number of digits equal to \( M - 1 \); such parts are added and the resulting number is normalized to \( M \).

**Goal**: to minimize the number of accesses to the buckets in order to retrieve the record.

The methods can be classified into:

- **Concatenation methods**: based on the use of pointers; the overflow records can be stored in a separate area or in the primary area itself.
- **Open addressing methods**: based on the use of a scan function to determine other buckets in the primary area where to store the overflow records.
Hash organizations

Handling overflows

- Primary area concatenation methods
  - coalesced chaining:
    - An overflow from the primary page $i$ is stored in the first non full primary page $i+h$ and a reference is activated from $i$ to $i+h$
    - The records for which $H(k) = i$ or $H(k) = i+h$ are stored in page $i+h$ until it becomes full
    - When page $i+h$ generates an overflow, we proceed in the same way
    - The list thus links pages that contain overflows from both page $i$ and page $i+h$

- separate chaining:
  - The records that collide (that is, to which the same address is assigned) are linked by a list
  - Whereas in the case of coalesce hashing, the lists link the pages, here the lists link the records
  - When the transformation function generates for a record a page which is fully and which stores some overflow records, one of such records is stored in another page
  - Such method improves the performance but complicates managing the overflows

- Concatenation methods with separate area
  - The overflows are stored in area different from the primary area
  - The separated area is usually organized in several pages
  - each page can be used for
    - overflows from the same page of the primary area (the pages storing overflows from the same page of the primary area are linked)
    - overflows from different pages of the primary area (the overflow records from the same page of the primary area are linked)
  - The capacity of the pages of the separate area must not necessarily be the same of the pages of the primary area
Open addressing methods
- They probe the primary area, starting from the page of address $H(k)$, to find a page which is not full
- Simplest method: linear probing
  $H_i(k) = (H_0(k) + si) \mod M$
  - The initial address $H_0(k)$ is incremented by constant quantity $s$, called step
  - If $s$ does not factors in common with $M$, the first $M$ values of $H_i(k)$ are all the possible addresses of the pages in the primary area

Open addressing methods
- problem: primary agglomeration
- Instead of being uniformly distributed over the entire primary area, the records tend to cluster in certain pages
- If $H_0(k_1)$ coincides with $H_0(k_2)$, the overflow probability of such page increases and next applications of the probe function will return the same pages for both keys

Open addressing methods
- example: function $H_i(k) = (H_0(k) + 3i) \mod 31$
- The keys 1234 and 245 generate respectively the sequences $(25, 28, 0, 3, 6, 9, 12, \ldots)$ and $(28, 0, 3, 6, 9, 12, \ldots)$
- An overflow from page 25 increases the probability of overflow from page 28, then of page 0, and so on
Hash organizations

Handling overflows

• Open addressing methods
  - In order to avoid the primary agglomeration, the
    probe step must be made variable by using a non
    linear probing function
  - quadratic probe
    \[ H_i(k) = (H_0(k) + s_1*i + s_2*i^2) \mod M \]
  - problem: agglomeration secondary, due to keys
    that are associated with the same initial address

• Problem: agglomeration secondary, due to keys
  that are associated with the same initial address

Load factor

• Given an estimate of the number \( N \) of records to be
  stored and given the capacity \( c \) of the buckets, the
  choice of the load factor \( d \) determines the number \( M \)
  of bucket of the primary area
• Low values of \( d \) reduce the overflow but increase the
  storage required for the file
• However in order to reduce the number of overflows,
  that impacts both the search and update costs, the
  load factor should not be too high

• Typical values are between 0.75 and 0.85
• In order to prevent problems concerning the hash
  function, it is necessary to carefully choose \( M \)
• Typically one can proceed as follows:
  - Given \( N \), number of records to be stored, and \( c \), the capacity of
    the buckets, a load factor \( d \) is selected
  - \( M \) is then determined as \( M = \frac{N}{d} \)
  - The overflow percentage is estimated; if it is too high, \( d \) must
    be reduced
Hash organizations
Load factor
- The total number of overflow \( N_t \) is obtained as the difference between \( N \), that is, the total number of records, and the number of records in pages without overflow and with overflow, that is:
\[
N_t = N - \left( \sum_{c=0}^{\infty} P(x)Mx + cP_M \right)
\]
where \( P_x = 1 - \sum P(x) \) is the probability that an address be generated more than \( c \) times.
Hash organizations

**Load factor**

Percentage of overflow records for different values of capacity and load factor:

<table>
<thead>
<tr>
<th>c / d</th>
<th>0.5</th>
<th>0.7</th>
<th>0.9</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.31</td>
<td>29.08</td>
<td>34.00</td>
<td>39.79</td>
</tr>
<tr>
<td>5</td>
<td>2.48</td>
<td>7.11</td>
<td>13.75</td>
<td>17.55</td>
</tr>
<tr>
<td>10</td>
<td>0.44</td>
<td>2.00</td>
<td>6.59</td>
<td>12.51</td>
</tr>
<tr>
<td>50</td>
<td>0.05</td>
<td>2.04</td>
<td>5.63</td>
<td></td>
</tr>
</tbody>
</table>

**Page capacity**

- For increasing values of c, and for constant values of d, the percentage of overflow records decreases, under the assumption of an ideal hash function and with a separate area for overflow.
- Suppose that we have to store 750 records in 1000 pages with c = 1 or in 500 pages with c = 2; in both cases the load factor is d = 0.75; however in the first case the overflow percentage is 29.6%, whereas in the second case the percentage is 18.7%.
- If we keep d = 0.75 and set c = 10, the performance further improves.
- Therefore for organizations on secondary storage the page capacity should be greater than 10.

Dynamic hash organizations

- In the dynamic hash techniques the memory allocation dynamically changes depending on the file dimension, without requiring the reorganization of the file.
- The possible approaches can be classified in two categories:
  - The approaches based on the use of auxiliary data structures (virtual hashing, extensible hashing, dynamic hashing)
  - The approaches that only work on the primary area (linear hashing, spiral hashing)
- In both cases, the key transformation function must be properly modified when the organization evolves.

Because the size of the physical pages (blocks) depend from the OS, logical pages (buckets) are used:

- The data file is divided in bucket (one or more blocks) and a bucket directory records a pointer to each bucket.
- The key transformation generates a bucket number.
- From the bucket directory the first block of the bucket is accessed to retrieve the record.
- If the first block does not store the record, the next block of the bucket is retrieved and so on.
Dynamic hash organizations: virtual hashing

- Idea: to double the primary area when an overflow arises in a bucket and to redistribute the records between the saturated bucket and its corresponding bucket in the newly allocated area, referred as its buddy, by making use of a new hash function.

- If then another bucket in the original primary area becomes full and its buddy has not yet been used, its records are redistributed between itself and its buddy.

- Because at a given time only some buddies are in use, it is necessary to use an auxiliary data structure to determine the hash function to use.

Initially, a small number (for example 7) M of contiguous pages of capacity c is allocated for the data area.

- A binary vector B is introduced having a number of elements equal to the number of pages of the data area.

- When a page of the data area is used, the corresponding element of B is set to 1.

- A transformation function $H_0$ is used such that when applied to a key generates an address in the range 0 and $M - 1$.

If when inserting a record with key value k in the page of address m, an overflow is generated:
- The data area is doubled.
- The vector B is doubled and all entries between $M + 1$ and $2M - 1$ are set to 0.
- $H_0$ is replaced by $H_1$ that generates addresses in the range 0 and $2M - 1$.
- $H_1$ is applied to k and to all the keys of the records in the page m; the records are thus redistributed between the page m and the page $M + m$.
- The entry of B corresponding to the page $M + m$ is set to 1.

The method requires using a series of transformation functions $H_0, H_1, \ldots, H_r$.

- $H_r$ returns an address in the range $(0, 2^r M - 1)$ (the subscript denotes the number of increments of the data area).

- The functions H must verify the following property:
  - For each $j = 0, \ldots, r$:
    - $H_{j+1}(k) = H_j(k) \text{ or } H_j(k) + 2^j M$
    - $H_r(k) = k \mod 2^r M$

- A function commonly used is $H_r(k) = k \mod 2^r M$.
Dynamic hash organizations: virtual hashing - search

INPUT: k: key to be searched
r: number of increments of the data area
OUTPUT: address corresponding to the key, if the key is present, -1 otherwise

METHOD:
function search(r:int; k:int): int;
If r < 0 Then return -1; /* the key is not present */
Else If B(Hr(k)) = 1 Then return Hr(k)
Else return search(r - 1, k)
endif
endif

Virtual hashing: example

Initial situation
M = 7
c = 3

The key 3820 has to be inserted
- H0(3820) = 5
- because B(5) = 1, the new key must be stored in the page with address 5

The page is full and thus the data area must be incremented
the records of the page with address 5 are redistributed between this page and a new page with address 12, through the use of function H1(k) = k mod 14
The vector B is expanded and its values updated; from this moment on the function H1 is used

Virtual hashing: example

Situation after the insertion of 3820
Virtual hashing: example

- Suppose that the key 3343 is now to be inserted
  - $H_1(3343) = 11$, but because $B(11) = 0$, that is the page 11 has not been yet used, $H_0$ is used
  - $H_0(3343) = 4$, the page 4 is however full and thus
    - The page 11 is activated by setting to 1 the corresponding bit in the vector $B$
    - All records from page 4 are redistributed, that is the records with key 7830, 1075, 6647 and the new record with key 3343, through the use of function $H_1$, so that they are redistributed between page 4 and page 11

Virtual hashing: notes

- Suppose that the key $k_1$ must be inserted in a page $m = H_0(k_1)$ that is full
- If by applying the function $H_j$ to the page $m$ the records are again all contained in page $m$ (thus generating an overflow again), the data area is extended again and the function $H_j$ is used
- We proceed in this way until no more overflows are generated
- This process certainly terminates
  - It terminates with $H_j$ with $2^jM$ greater than all the key values to be redistributed
- It may however happen that there is no more available storage; in such case the overflow cannot be handled

Extensible hashing

- The extensible hashing avoids doubling the data storage by using an auxiliary data structure, referred to as directory
- The directory is a set of $2p$ entries, with addresses from 0 to $2p-1$; $p \geq 0$ is called directory depth
- The data area is expanded by adding a new page each time a record is to be inserted in a page already full
Extensible hashing

- **Main idea**: use of a hash function that, given a key value $k_i$, does not return a bucket address; instead it returns a binary string of proper length (for example, 32 bits) called **pseudo-key**
- The hash function associates with each key a pseudo-key of which the first $p$ bits are considered to directly access one of the $2^p$ entries of the directory, then the bucket address included in the entry is fetched
- each bucket has a local depth $p' \leq p$ (recorded in the bucket) indicating the actual number of bits used to allocate the keys in the bucket

An entry in the directory contains the reference to a page of the data area containing all records with pseudo-keys having the same prefix of length $p'$

To search for a record given its key value:
- The first $p$ bits are extracted from its pseudo-key
- The entry of the index corresponding the string of $p$ bits is accessed
- From the entry the address of the page containing the searched record is fetched

To insert a record:
- Initially there is only one bucket and $p = p' = 0$
- When a record must be inserted in a page which full and has depth $p'$, if $p = p'$ the directory is expanded (by doubling the number of entries) and $p$ is incremented by 1; the values of the pointers are copied in the new corresponding entries

- Both in the case $p = p'$ and in the case $p' < p$, a new bucket is allocated and the keys are redistributed between the two buckets (the one full and the newly allocated one) by making use of the $p'+1$-th bit of the pseudo-keys
- For both buckets, the depth is set to $p'+1$ and the directory is updated by inserting a reference to the new bucket in the proper entry
Extensible hashing: example

Possible values of the pseudo-key

- Red: 0010 1001 0101 1111 1010 0100 1100 1011 0011
- White: 1010 0011 0110 0000 1100 0110 1001 1111
- Pink: 1000 0111 1110 1010 1011 1011 0011 0001
- Martin: 1111 0001 0100 1000 1001 0111 1110 1111
- Blacchi: 1011 0101 1010 0110 1100 1001 1110 1011
- Black: 0101 1000 0011 1111 1101 1100 0000 0001

Comparison between indexes and hash functions

- If the majority of queries has the form
  \[ \text{SELECT } A_1, A_2, \ldots, A_n \text{ FROM } R \text{ WHERE } A_i = C \]
  the hash technique is to be preferred
- motivations:
  - The cost of scanning an index is proportional to the log of the number of values that attribute \( A_i \) has in relation \( R \)
  - In a hash structure the search time is independent from the size of the relation

The trees are to be preferred if the queries use range conditions

\[ \text{SELECT } A_1, A_2, \ldots, A_n \text{ FROM } R \text{ WHERE } C_1 < A_i < C_2 \]

- The hash functions typically do not maintain the order
All relational DBMS provide several commands that allow the designer to specify the physical configuration of the database. Such commands are made available as SQL commands. There is no standard for such commands; however for the main commands the various DBMS adopt a similar syntax.

The most relevant commands are: the command for creating indexes, on one or more columns of a relation, and the command for creating clusters.

A cluster allows one to store on contiguous storage locations the tuples, of one or more relations, that have the same value for one or more columns, called cluster columns.

The command for creating an index in SQL has the following format:

```
CREATE INDEX IndexName
ON RelationName (ColumnNameList) |
ClusterName
[ASC | DESC];
```

where
- IndexName is the name of the index being created
- The ON clause specifies the object on which the index is allocated.

The object on which an index is allocated can be:
- a relation: one must specify the names of the columns on which the index is allocated
- a cluster: the index is automatically allocated on all columns of the cluster

An index can be allocated on several columns.

The ASC and DESC options specify if the values of the index key must be ordered according to an increasing or decreasing order.
- ASC is the default.
Definition of clusters and indexes in SQL – Example

- The indexes are in general implemented as a B+-tree or some variations of it
- Suppose to allocate an index on the column salary of the Employees table
  ```sql
  CREATE INDEX idxsalary ON Employees (salary);
  ```

Definition of clustered indexes (DB2):
```
CREATE INDEX IndexName ON RelationName (ColumnNameList)
CLUSTER;
```
- Only a single clustered index can be allocated on a given table
- If the table is not empty when the clustered index is created, the data are not automatically re-grouped; it is necessary to use a special utility called REORG

Command for the creation of a cluster:
```
CREATE CLUSTER ClusterName
(ColName_1 Domain_1, . . ., ColName_n Domain_n)
[INDEX | HASH IS Expr ]
HASHKEYS n;]
```
- ClusterName is the name of the cluster being defined
- (ColName_1 Domain_1, . . ., ColName_n Domain_n), with n ≥ 1, is the specification of the cluster columns
  - such set of columns is called cluster key

An auxiliary access structure is always associated with each cluster
- Index:
  - The tuples with the same value for the cluster key are clustered and indexed by a B+-tree (default)
  - The index is convenient if there are frequent queries with range predicates on the cluster key or if the relations may frequently change size
- Index cluster
  - Hash:
    - The tuples with the same hash value for the cluster key are clustered and indexed by a hash function
    - The hash function is convenient if there are frequent queries with equality predicates on all cluster key columns and the relations are static
    - Hash cluster
Definition of clusters and indexes in SQL – hash clusters

- The DBMS always provides an internal hash function used as default (very often based on the division method)
- the HASHKEYS option allows one to specify the number of values for the hash function
- If such value $v$ is not a prime number, it is replaced by the system with the first prime number which is greater than $v$
- such value is used as input for the integer remainder function used by the system to generate the values of the hash function

Example

Index cluster:
```
CREATE CLUSTER Personnel(D# NUMBER);
CREATE INDEX idpersonne
ON CLUSTER Personnel;
```

Hash cluster:
```
CREATE CLUSTER Personnel (D# NUMBER) HASHKEYS 10;
```
given that the HASHKEYS option is equal to 10, the number of values generated by the hash function is 11 (first prime number > 10)

Definition of clusters and indexes in SQL – index clusters

- It is possible to modify the hash function to be used through the HASH IS option
- Such option can however only be used if:
  - The cluster key only includes columns containing integer values
  - The expression must return only positive values
  - * other conditions

Definition of clusters and indexes in SQL – index clusters

- If the cluster is of type index, before executing queries or modifications, an index on the cluster must be created through the CREATE INDEX command
- a cluster may include one or more relations
  - Single relation: the cluster is used to group the tuples of the relation that have the same value for the columns that are in the cluster key
  - Multiple relations: the cluster is used to group the tuples of all relations having the same value for the columns that are in the cluster key (the joins on the columns that are part of the cluster key are efficient)
- A relation must be inserted in the cluster when it is created
Definition of clusters and indexes in SQL - Example

- Suppose to insert in the Personnel cluster the Employees and Departments relations
  
  ```
  CREATE TABLE Employees
  (Emp# Decimal(4) NOT NULL,
   Dept# Decimal(2))
  CLUSTER Personnel (Dept#);
  
  CREATE TABLE Departments
  (Dept# Decimal(4) NOT NULL)
  CLUSTER Personnel (Dept#);
  ```

Definition of clusters and indexes in SQL - Example

- The names of the columns on which the clustering of the relations is executed must not necessarily have the same names of the cluster columns; they must however have the same type