Advanced Databases 23-24

Notes

University of Pisa

M.Sc. in Computer Science

Contents

Chapter 1

Introduction

The most common use of information technology is to store and retrieve data, be it text, images, video, or audio files. As the amount of data generated by several processes increases as time goes on, storage systems must evolve to guarantee reliable access to the data, as well as fast and efficient retrieval. Data is typically stored into **databases** (**DBs**), which are housed in a permanent memory.

The technology on which permanent memory is based uses magnetic **disks**, containing a set of platters that rotate at relatively slow speeds (compared to CPU speed), which can be interacted with by using heads attached to moving arms. Each platter has on both surfaces a set of rings, called **tracks**, which, except for the innermost and outermost ones, are used to store information. Each track is subdivided into **sectors** of the same size, which correspond to the smallest unit of transfer allowed by the hardware. Typical sector sizes are 512 bytes, 1 KB, 2 KB, or 4 KB. There are from 500 to 1000 sectors per track, and about 100K tracks per surface of a single platter.

The **access time** needed to read a section of the disk is given by the seek time (needed to move the head), the rotational delay (given by the spinning of the disk itself), and the transfer time (needed to read/write the data). These operations take several milliseconds to be completed, which are definitely slower than any operation relative to the **main memory** (**RAM**), taking only a few nanoseconds in total.

Despite this disparity, disks are still today the preferred technology to store data. Main memory is, in fact, volatile: once the machine stops receiving electricity powering it on, any information on the RAM is lost forever. On the other hand, disks provide reliable storage: the information written on them can be retrieved even if the machine is turned off and on. A newer technology, called **solid state storage**, and, in particular, **flash memory**, has risen in popularity in the last years. It provides the reliability of disks and much faster operations, although they still haven't become the new standard since they tend to be expensive.

Chapter 2

Overview of a DBMS

This chapter will give a general overview of the structure of a centralized **DBMS** (**Data Base Management System**) based on the relational data model, describing its components and their respective functionalities.

2.1 Architecture

A database is a collection of homogeneous sets of data, with relationships defined among them, stored in permanent memory, and used via a DBMS.

DBMS

A DBMS is a software that provides the following functionalities:

- A language to describe the **schema** of the database (a collection of definitions that describe the data structures), restrictions on the allowed data types, and the relationships among data sets;
- The data structures for storage and efficient retrieval of large amounts of data;
- A language to guarantee secure access to the data only to authorized users;
- A **transactions** mechanism to protect data from HW/SW malfunctions and errors during concurrent access.

The architecture of a DBMS provides the following basic components:

- The **Storage Engine**, which includes modules supporting:
    - **Permanent Memory Manager**;
    - **Buffer Manager**;
    - **Storage Structures Manager**;
    - **Access Methods Manager**;
    - **Transaction and Recovery Manager**;
    - **Concurrency Manager**.
- The **Relational Engine**, which includes modules supporting:
    - **Data Definition Language**;
    - **Query Manager**;
    - **Catalog Manager**.

In real systems the functionalities of these modules are not completely separated in different components (as in Figure 2.1), but this overview can help in understanding the purpose of each of them.

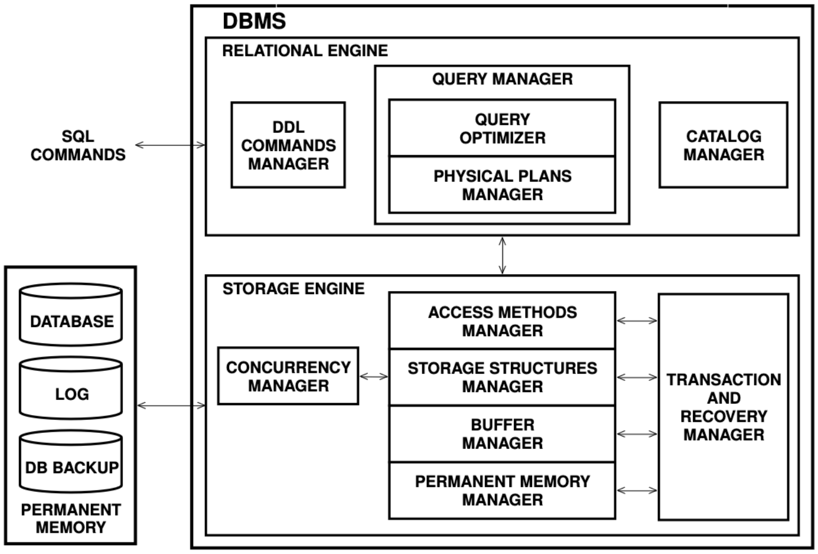

Figure 2.1: The architecture of a DBMS.

2.1.1 Permanent Memory Manager

The PMM manages page allocation and deallocation on disk storage. It hides the disk characteristics and the operating system, as it provides an abstraction of the memory as a a set of databases, each consisting of a set of logical files of **physical pages** (or blocks) of fixed size. The physical pages of a file are numbered consecutively starting from 0, and their number can grow dynamically with the only limitation being the available space in the permanent memory. Each collection of records (table or index) of a database is stored in a logical file, which can also be realized as an actual separate file of the operating system or as part of a file in which the database is stored.

Once a physical page is transferred to main memory, it is called a **page**, and it is represented with a specific, complex structure.

2.1.2 Buffer Manager

The Buffer Manager is tasked with transferring pages between temporary and permanent memory. It allows transactions to get the pages they need minimizing the number of disk accesses. In general, the performance of operations on a database depends on the number of pages transferred to temporary memory. If a big enough buffer is used, and there's a high number of access requests for a specific page, there's a high likelihood that such page will be in the buffer. Figure 2.2 illustrates the basic structure of a Buffer Manager.

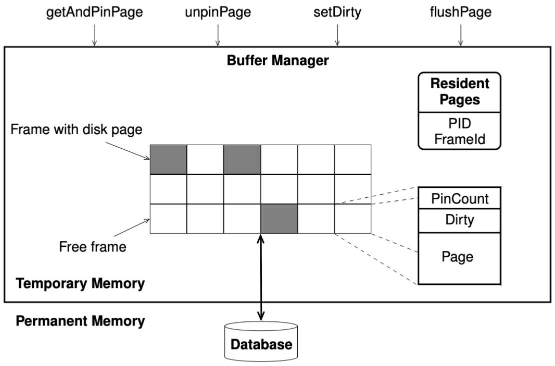

Figure 2.2: The components of the Buffer Manager.

The **buffer pool** is an array of **frames**, each containing a copy of a page present in permanent memory, and some additional bookkeeping information. The pool has a fixed size, so when there are no more free frames, a page must be freed with an appropriate algorithm. Each frame stores two variables, the **pin count** and the **dirty**. The former counts the number of transactions currently using the page hosted on that frame; its value starts at 0, and increases by 1 each time it is requested, and decreases by 1 each time it is released. The latter indicates whether the page was modified since it was copied into the buffer, signaling that the modification must be reflected on disk as well. The **resident pages** table is a hash table that is used to know which page in permanent memory (identified by a PID) is stored in which frame.

A commonly used replacement policy id the **Least Recently Used** (**LRU**) policy. Once the buffer pool is full, the frame chosen to be ejected is the one that was the earliest one to be pinned. The idea is that since the page hasn't been requested for a relatively long time, it probably won't be requested any time soon. However, this policy may not always be the best: for example, in a join loop between two tables, the LRU policy may be optimal for one table, while the optimal one for the other is Most Recently Used (MRU).

2.1.3 Storage Structures Manager

The Storage Structure Manager implements databases as tables of records, representing the files of pages provided by the Permanent Memory Manager. Above the Storage Structure Manager, the unit of access is a record; below, the unit of access is a page. For this reason, the unit of costs considered for now will be a single page access (read or write), and we assume that memory operations have 0 cost, since they're so much faster than disk operations their addition to the overall cost is negligible. The most important type of file is the **heap file**, which stores records in no particular order.

A **record** is a collection of one or more **attributes**, and contains some extra information, called **record header**, needed for record management. We assume records are not larger than a page (a few KB big), and that each attribute is either separated from the others using a separator, or all attributes are stored sequentially and are indexed by using an offset. Each record is uniquely identified by a **RID**, which specifies the page and the offset the record can be found at. Sometimes this offset may be logical, i.e., it actually indicates a position on an array of actual pointers to records; this way, records can be moved around without having to externally modify their RID.

Collections of pages may be stored using different data structures. Usually, pages are stored with two alternatives. The first uses two doubly linked list, one containing free pages, the other containing full ones. The other alternative consists in a **directory**, where each entry contains a pair PID-available space. If the directory grows and cannot be stored in the header page of the file, it is organized as a linked list. For efficiency reasons, the free space existing in different spaces cannot be compacted into new free pages. If the available free space is plenty but there's no actual free pages available, it may be necessary to reorganize the database.

2.2 External Sorting

A frequent operation done in DBMS is **sorting**. Sorting collections of records may be done for different reasons: it may be needed to perform a join operation, delete duplicates, or load them into physical organizations.

Since typically temporary memory cannot hold all of the records of a file at the same time, merging is done by using **external sorting algorithms**, of which most widely used one is **merge-sort**. Let $N_{pag}(R)$ be the number of pages in the file, and B the number of available pages on the buffer. Merge-sort operates in two phases:

1. The **sort phase**, in which B pages are read into the buffer, sorted, and written to disk. This creates exactly $n = \lceil N_{pag}(R)/B \rceil$ sorted subsets of records, called **runs**. Each run is stored in a separate numbered auxiliary file, and contains the same number of pages (except for the last one, which may contain less if the number of file pages isn't divisible by the number of free buffer pages);

2. The **merge phase**, which fuses the sorted runs to reconstruct the file. In each merge pass, $Z = B - 1$ runs are merged using one buffer page left free to produce the output. The number of runs at the end of a merge pass becomes $n = \lceil n/Z \rceil$. Merges are repeated until $n < 1$.

Once the algorithm terminates, the final auxiliary file contains the sorted data. Z is called the **merge order**, and a total of $Z + 1$ buffer pages are needed to execute a Z-merge (since as stated above, one page must be left free).

The cost of the algorithm is evaluated in terms of how many read/write operations are needed in total, Since we're ignoring operations directly involving records. The overall cost is given by two terms:

$$C_{sort}(R) = SortCost + MergeCost =$$
$$= 2 \times N_{pag}(R) + 2 \times N_{pag}(R) \times MergePasses$$

If the number of file pages is less than B^2, (or, more precisely, $B(B-1)$), the data can be sorted in a single pass, so the cost becomes:

$$C_{sort} = 2 \times N_{pag}(R) + 2 \times N_{pag}(R) \times 1 = 4 \times N_{pag}(R)$$

The number of passes is a function of the number of file pages $N_{pag}(R)$, the number of initial runs S, and Z. S also depends on the number of file pages and the number of buffer pages available at the start:

$$S = \lceil N_{pag}/B \rceil$$

At each merge pass, the algorithm merges together Z runs. At the start, the runs will be S. After one merge pass, they will be S/Z. At the second pass, they will be $S/Z * 1/Z = S/Z^2$, and so on until only one run remains. Since the number of runs decreases exponentially, we can write that the total number of passes will be:

$$k = \lceil \log_Z S \rceil$$

The total cost can be rewritten as:

$$C_{sort} = 2 \times N_{pag}(R) + 2 \times N_{pag}(R) \times \lceil \log_Z(S) \rceil \ .$$

2.3 Data Organizations

2.3.1 Heap and Sequential Organizations

The data can be arranged either via **heap organization**, or **sequential organization**. With heap organization, every new record is added to the end of the file: insertion is easy and efficient in terms of memory used. It is ideal for situations where insertion is more common than search, or files where massive search is common. This is also the standard organization for DBMS.

With sequential organization, data is kept sorted on a **search key** K, picked as a single attribute of the records. This makes equality and range search on K very efficient. On the other hand, insertion is more problematic, since the ordering of the records must be maintained at all times. Insertion may use a **static solution**, where each page is filled normally, and for each insertion, the record is placed at the correct spot in the ordering, moving all other records after it. A **dynamic solution** instead keeps some fraction of the total space in a page free. Once a page has filled up enough, its contents are split into new pages. This way, pages always have some extra space at the end to accommodate new insertions: when a record is added, the shifting of the records after it will only involve the ones in the same page. Alternatively, a **differential file** may be used to keep track of which changes must be applied to which pages, so that all insertions can be done all at once in a second moment.

Table 2.1 shows a comparison between the two organization types. $N_{pag}(R)$ refers to the number of pages required to store the records. The **selectivity factor** sf is an estimate of the fraction of pages occupied by records that satisfy the condition of a range search, and is calculated as:

$$sf = \frac{(k_2 - k_1)}{(k_{max} - k_{min})} \ ,$$

Type	**Memory**	**Eq. Search** (C_s)	**Range Search**	**Insertion**	**Deletion**
Heap	$N_{pag}(R)$	$\left\lceil \frac{N_{pag}(R)}{2} \right\rceil$	$N_{pag}(R)$	2	$C_s + 1$
Seq.	$N_{pag}(R)$	$\lceil \log_2 N_{pag}(R) \rceil$	$C_s - 1 +$ $\lceil sf \times N_{pag}(R) \rceil$	$C_s + 1$ $+N_{pag}(R)$	$C_s + 1$

Table 2.1: Comparison between heap and sequential organization.

with k_1 and k_2 being the two extremes of the range, and k_{max} and k_{min} the highest and lowest values in the domain of the attribute.

The equality search is faster for the sequential one since it uses a binary search algorithm, while the heap one has to compare the record against all pages since no specific ordering is imposed. The cost estimation is also only valid if the data distribution is uniform; if it follows some other distribution, e.g., Gaussian, the actual cost may be high (worst case exactly $N_{pag}(R)$).

The range search for the heap organization costs $N_{pag}(R)$ since it must read all pages to make sure it collects all records falling within the specified range. For sequential organization, it costs an equality search to find the starting record of the range, plus the number of pages needed to store the records in that range. The "-1" is added because the first page has already been found with the binary search.

The final biggest difference lies in the costs for insertions: it is constant for heap organization, while for sequential organization it costs a search to find the spot to insert the record, and all subsequent $N_{pag}/2$ pages must be read and written to move their records forward. If the page the record is added to is not completely full, the insertion will still cost $C_s + 1$.

2.3.2 Key-based Organizations

A table organization based on a key allows the retrieval of a record with a specified key in as few accesses as possible, 1 being the optimum. The set of records in the table is **mapped** to a set of keys, via either a **primary organization** or a **secondary organization**.

Primary and Secondary Organizations

A table organization is primary if it determines the way the records are physically stored, and therefore how they can be retrieved. Otherwise, it is a secondary organization.

For a primary organization, the mapping can be done using a **hash function** or a **tree structure**. It can be either **static** or **dynamic**.

Static and Dynamic Primary Organization

A primary organization is static if the performance degrades gradually as insertions and deletions are performed, requiring reorganization.
A primary organization is dynamic if once created, it evolves with insertions and deletions, preserving efficiency of operations.

In a secondary organization, the mapping from key to record is implemented with the **tabular method/index**, listing all inputs and outputs.

Static Hashing Organization

This is the simplest methods for a primary table organization. We assume to have N records all of the same fixed size, and that keys are integers. The records of the same table R are stored in the **primary area**, divided into M buckets, each of which may consist of one or several pages. For now, we will assume each bucket to contain only one page of capacity c, and that pages are numbered from 0 to $M - 1$.

A record is inserted into a specific bucket chosen by calculating its address via a **hashing function** H, applied to the record key value. The ratio:

$$d = \frac{N}{(M \times c)}$$

is called the primary area **loading factor**, which represents how full the primary area is. The hash function should produce addresses uniformly distributed in the interval $[0, M - 1]$, and it may return the same address for different keys. This causes a **collision**. Records hashed to the same page are stored in order of insertion. When an insertion is attempted in a page that is completely full, an **overflow** occurs, and must be appropriately managed.

The design of a static hashing organization depends on the choices done for the following parameters:

1. The **hashing function**: a good hashing function must randomly assigning keys to elements in the address space. There's no single "ideal" hash function, but generally, simpler functions tend to perform better than complex ones. A common choice is the **modulo function**: $H(k) = k \mod M$, with M a prime number.

2. The **overflow management technique**: two commonly used techniques are **open overflow** and **chained overflow**. Open overflow performs a primary area linear search to find the first empty space to insert the record; when the last page has been searched, the process restarts from the initial page. Chained overflow collects overflow records chained together in a separate area, pointed to from the home page.

3. The **loading factor**: low loading factors and higher page capacities give better performances, but occupy more memory. For low ($d < 0.7$) loading factors, retrieving a record requires a single access on average. For high values ($d > 0.8$), the primary area size is reduced, increasing the probability of overflow; open overflow deteriorates rapidly, while chained overflow still performs well.

4. The **page capacity**: higher values of page capacity reduce the number of overflows, which are the main culprit in performance degradation of hashing organizations.

As for overall performances, for page capacities less than 10 it is preferable to give up hash organizations. A static hashing has excellent performances as long as there are no overflows to manage, with the average cost of an equality search being 1. As overflows start to happen, reorganization is needed, which requires the creation of a new primary area, choosing a new hash function, and reloading all the data.

A big drawback of static hashing is that is does not support efficient range queries, since records with similar keys will typically not end up in the same bucket.

Dynamic Hashing Organization

Dynamic hashing organizations can be divided into two groups: those that use a primary area and an auxiliary data structure whose size changes with the primary area size, and those in which only the primary area size changes dynamically. In both cases, the hashing function changes automatically when the structure changes dimension, maintaining the average access time equal to 1. We will see two types of dynamic organizations with auxiliary data structures (Virtual Hashing and Extendible Hashing), and two without (Linear Hashing and Spiral Hashing).

Virtual Hashing Virtual hashing works as follows:

1. The data area initially contains M contiguous pages of capacity c. Each page is identified by its address (between 0 and $M-1$).

2. A bit vector $\mathcal{B}$ is created, indicating with a 1 which page contains at least a record.

3. An initial function H_0 is used to map each key to and address m. If an overflow happens, then the data area is doubled, maintaining the pages as contiguous, the hashing function is replaced with a new one (H_1) that maps keys to addresses in the range $[0, 2M-1]$, and the hashing function is applied to all keys and all records in the overflowing page m. These records end up being distributed between m itself and some other new page m'.

This method defines a series of hashing functions,

$$H_0, H_1, H_2, \dots, H_r \ ,$$

where H_i produces a page address in the range $[0, 2^i M - 1]$. The function chosen as the hash function must satisfy the following constraints:

$$\begin{aligned} & H_{j+1}(k) = H_j(k) \\ \text{or} & \\ & H_{j+1}(k) = H_j(k) + 2^j \times M, \ j = r, r-1, \dots, 0 \end{aligned}$$

for all keys k. This means that the new hash function chosen either returns the same address the key already corresponds to, or a new address that is equal to the original one plus half of the new address space. A common function is $H_r(k) = k \mod (2^r \times M)$.

To find a record, known its key and r (the number of times the data area has been doubled), a recursive function is used: This technique requires memory equal to the one occupied by the data area and the bit vector $\mathcal{B}$. The memory is not very well used however, because of the frequent need to double the data area.

Extendible Hashing Instead of using a bit vector, extendible hashing uses a fixed set of data pages with a **directory** $\mathcal{B}$, containing a set of pointers to data pages. The directory is smaller in size than the primary area, and is doubled as needed.

Let r be a record with key k. The value produced by $H(k)$ is a binary value of b bits (usually 32), called hash key. The hash key does not represent an actual address;

Algorithm 1 PageSearch pseudocode.

```
if $r < 0$ then
    The key does not exist.
else if $\mathcal{B}(H_r(k)) == 1$ then
    Return $H_r(k)$
else
    PageSearch($r - 1$, $k$)
end if
```

instead, pages are allocated on demand as records are inserted into the file, considering only the initial p bits of b, which are used as an offset into the directory $\mathcal{B}$. The value of p grows and shrinks with the number of pages used by data, and the number of entries in the directory is always 2^p. p is called the **directory level**. Each entry in the directory is a pointer to a data page containing records with the same first p' bits of their hash key, with $p' \in [0, p]$. p' is called **data page level**.

The hash structure is initially empty, with $p = 0$, and is a directory with one entry containing a pointer to an empty page of capacity c. The first c records are inserted in the page; as we try to insert a new record into a full page, there are two possibilities:

- If $p' = p$, $\mathcal{B}$ is doubled and p becomes $p + 1$. Let w be the bits of the previous value of p. Then, the entries in the doubled directory indexed by w_0 and w_1 each contain a pointer to the same data page that w used to point to.

- If $p' < p$, then the data page is split in two, creating a new page, and each of the halves' level p' take value $p' + 1$. The records in the original page are distributed across the halves, based on the value of the first high-order bit of their hash keys. Records whose key has 0 in the $(p' + 1)^{th}$ bit stay in the old page, while those with a 1 will go in the new one. The pointers in the directory are updated so that those that pointed to the original page now point to the new half, depending on the value of the $(p' + 1)^{th}$ bit.

The advantage of this method is that performance does not degrade as the file grows, and the directory $\mathcal{B}$ keeps the memory overhead low. The retrieval of a record has an additional level of indirection since $\lfloor$ must be accessed first, but this has very little impact on the performance, since most of the directory ill be in main memory.

Linear Hashing Linear hashing increases the number of data pages as soon as an overflow occurs, but the page which is split is not the one that flows over; instead, it is the page pointed by the current pointer p, initially equal to 0, and incremented to 1 each time an overflow happens.

Initially, M pages are allocated, and the hash function used is $H_0(k) = k \mod M$. When an overflow happens in a page with address $m \geq p$, an overflow chain is maintained for page m, and a new page is also added. All records in page p are distributed between page p and the new page, using the new hash function $H_1(k) = k \mod 2M$.

As page M overflows, a total of M duplications have happened, bringing the memory to $2M$ pages. Pointer p is reset, and H_0 is replaced by H_1. H_1 is in turn replaced by $H_2(k) = k \mod 2^2M$, and so on. After r doublings, the function $H_r(k) = k \mod 2^rM$ will be used. To retrieve a record with key value k, the page address is calculated as:

Algorithm 2 PageAddress pseudocode.

```
if $H_i(k) < p$ then
    $H_{i+1}(k)$
else
    $H_i(k)$
end if
```

Linear hashing has similar performances to extendible hashing.

Spiral Hashing Spiral hashing considers the memory as if it were organized on a spiral instead of a line. Like linear hashing, spiral hashing requires no index, but has better performances and storage utilization because of three particular property: the hashing function distributes records unevenly, accumulating records in the pages at the beginning of the address space, while the pages at the end have a lower load. The page that is split is one that is very unlikely to overflow.

Tree-structure Organizations

All the previous organizations have the big disadvantage of not supporting the range equality search operation. An alternative organization commonly used in DBMSs use dynamic tree structures to store pages. The **order** of a tree is the maximum number of children a node can have. The **level** of a node is the number of nodes encountered in the path from the root to the node itself. The **height** of the tree is the macimum level of a node. A tree is **balanced** if the levels of all leaf nodes differ by at most 1.

The types of trees most commonly used are B-trees and B$^+$-trees, since unlike binary trees they manage to keep a relatively low height even with an high number of pages. One solution may be to store the nodes of the binary tree in main memory, such that each page contains the same number of nodes. In the example shown in Figure 2.3, each page contains 8 nodes, each of which refers to 8 different pages. Using this strategy, the depth of the tree in terms of pages to access is greatly reduced: an equality search has a complexity of $\log_8(N_{pag}(R))$ instead of $\log_2(N_{pag}(R))$.

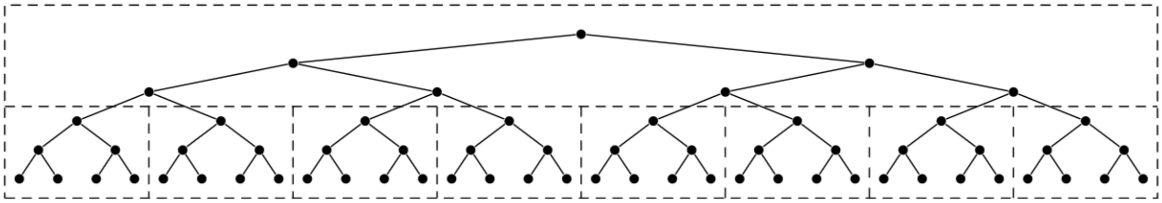

Figure 2.3: A paged binary tree: each page in main memory is delimited by a dashed line.

Still, this structure must be kept balanced when insertions or deletions are performed, and algorithms that maintain binary trees can be very costly. The following sections will explain how using multiway trees can be a solution.

B-trees A B-tree is a perfectly balanced search tree, in which each node has a variable number of children. We will indicate a key as k and the full record associated with it $k*$. Also, we'll assume that all keys are integers and that all records have the same fixed size. An example of B-tree can be seen in Figure 2.4.

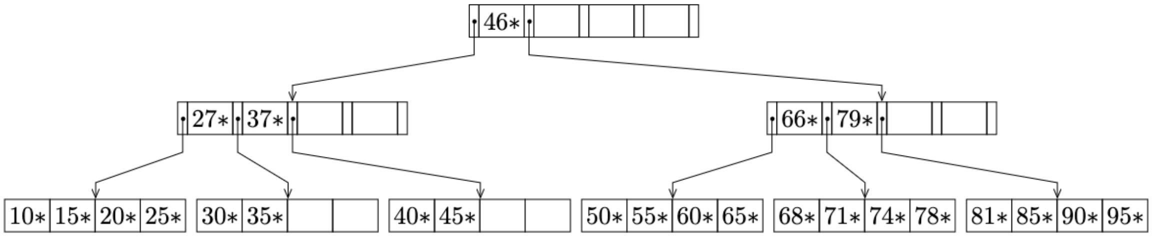

Figure 2.4: A B-tree.

A B-tree is defined as follows:

B-tree

A B-tree of order $m \geq 3$ is an m-way search tree that is either empty or of height $h \geq 1$, and satisfies the following properties:

- Each node has at most $m-1$ keys, and, except for the root, at least $\lceil m/2 \rceil - 1$ keys;
- A node with j keys will have $j+1$ pointers to children, undefined in the leaves, and $K(p_i)$ is the set of all keys in the i^{th} child node;
- All leaves are on the same level;
- Each non-leaf node has the same following structure:

$$[p_0, k_1*, p_1, k_2*, \ldots, k_j*, p_j],$$

where each p_i is a pointer to a child node such that, for all $0 < i < j$:

$$\forall k \in K(p_i),\ \ k_{i-1} < k < k_{i+1}$$

There is a strict relationship between the height of the tree h, the order m, and the number of keys N. Since the non-root nodes are constrained in the number of keys they must maintain, the following inequality holds true:

$$\log_m(N+1) \leq h \leq \log_{\lceil m/2 \rceil} \frac{N+1}{2} + 1$$

The left size of the inequality corresponds to a B-tree with all of its nodes completely filled, while the right side to a B-tree where each node has the least acceptable amount of keys.

The following is a summary of the costs of operations using a B-tree. An equality search for a specific key k starts at the root; if the key is not in the root (and $h > 1$), the search continues in the child that will likely contain the key (since keys are ordered, the chosen pointer is the one right after the biggest key smaller than k). The overall cost is $1 \leq C_s \leq h$.

As for the range search, to retrieve all records with keys in increasing order, the tree must be visited in the **in-order traversal**, starting from the leftmost leaf and gradually moving to the right. Let $sf = (k_2 - k_1)/(k_{max} - k_{min})$ be the selectivity factor for the search; the overall cost will be $C_{range} = sf \times N_{nodes}$, since keys will be “scattered” across different nodes of the tree at different levels. The following bond holds true: $h \leq C_{range} \leq N_{nodes}$. It will always require traveling to a leaf, and in the worst case scenario, it may have to read all nodes in the tree.

Insertion in a non-full leaf is easy: the new key is simply added to a leaf so that the keys are correctly sorted. The cost is h reads and 1 write. If the leaf is full, then the node must be split, so that the old node will retain the first half of keys, and the new node will get the second half. The median key is inserted into the parent node, and the new node is pointed by the pointer on its right. In case the parent node is full as well, the operation repeats. The cost for a worst case scenario is h reads and $2h + 1$ writes.

For deletion, there's three possibilities:

- If the key is in a leaf, and the removal keeps the number of keys within the acceptable range, the cost is h reads and 1 write;
- If the key is a node, and no other operations are needed, the cost is h reads and 2 writes (the key is replaced with the next following one);
- If the key is in a leaf node and the final number of keys is less than $\lceil m/2 \rceil - 1$ elements, then a **rotation** or a **merge** are needed.

  A node is merged with one of its brothers which contains $\lceil m/2 \rceil - 1$ keys, moving all of them to the first node. Additionally, the key in the parent node that was between the pointers of the two children involved in the merging is also removed and added to the merged child. In case this produces an underflow in the parent node, the operation is repeated until the whole tree is balanced.

  When a merge is not possible because all brothers are too full, a rotation is performed instead. When the key is deleted, the maximum key from the left brother is moved into the parent, and the key in the parent is moved into the underfull node.

  When either of these operations are needed for all nodes from root to leaf, the cost is $2h - 1$ reads and $h + 1$ writes.

Table 2.2 summarizes all the operation costs.

B$^+$-trees B$^+$-trees are a variant of B-trees that perform especially well for range searches. In a B$^+$-tree, all the records $k*$ are stored sorted in the leaf nodes, organized in a doubly linked list. Each non-leaf node stores the highest key of the child pointed by the previous pointer. An example of B$^+$-tree can be seen in Figure 2.5.

It can be thought of as the combination of a sparse index and a sequential file. Records are part of the tree structure stored in one file, so to read all records in a sorted order, the tree structure must be necessarily used to locate the first data page.

	Eq. Search (C_s)	**Range Search**	**Insertion**	**Deletion**
Best case	1	$h \leq sf \times N_{nodes} \leq N_{nodes}$	$h + 1$	$h + 1$ or $(h + 2)$
Worst case	h	$h \leq sf \times N_{nodes} \leq N_{nodes}$	$2h + 1$	$(2h - 1) + (h + 1)$

Table 2.2: Costs for B-tree organization.

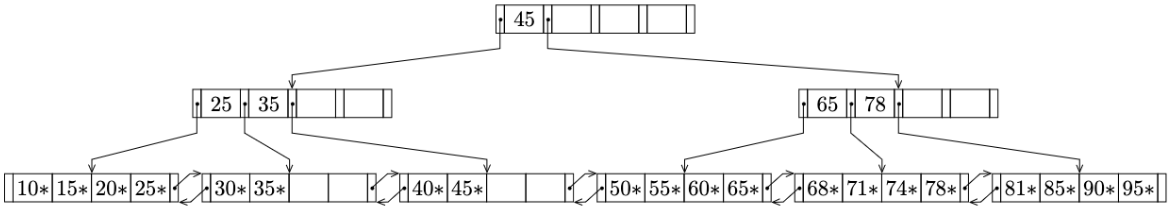

Figure 2.5: A B$^+$-tree.

Compared to B-trees, B$^+$-trees tend to be much shallower, since the non-leaf nodes only contain keys but not records, which can be found in the leaves, connected together. This makes any sequential/range scan of the data faster: the cost for a ranged search is $sf \times N_{leaves}$. For the equality search, since we're assuming the tree to have a level equal to no more than 3, will take from 1 to 3 read operations.

Another big difference is that in deletion, there's no need to replace it in the father node, unless the deletion requires a merge or rotation (in which case the operations are the same as the B-tree).

Index (Secondary) Organizations

A secondary organization is defined as follows:

Secondary Organization

An index I on a key K of a set of records R is a sorted table $I(K, RID)$ on K, where $N_{rec}(R) = N_{rec}(I)$. An element of the index is a pair (k_i, r_i), where k_i is the key of a record, and r_i is the corresponding RID.

An index is a tabular data structure that supports fast retrieval of records by exploiting the ordering of the keys. It can be defined on one or more keys. An index is typically stored in a B$^+$-tree structure.

If the order of the data records is approximately the same as the order of the entries in the index, then it is called a **clustered index**. When the index is first created, a sort is performed on the actual data records, matching the index order. As insertions are performed, this ordering may be gradually lost, also reducing the effectiveness of such organization, so the clustered index may have to be recreated from time to time. If instead the data records do not follow the same order as the index records, it is called an **unclustered index**.

The cost for searches done using this organization can be broken down into two terms:

$$C_s = C_I + C_D \, ,$$

where C_I is the cost of accessing the index pages to find the RIDs needed, and C_D is the cost of accessing the actual data pages containing the records. Table 2.3 summarizes the differences between the two types of indexes.

	Eq. Search (C_s)	**Range Search**
Clustered	$C_I = 1$, $C_D = 1$	$C_I = sf \times N_{leaf}$, $C_D = sf \times N_{pag}$
Unclustered	$C_I = 1$, $C_D = 1$	$C_I = sf \times N_{leaf}$, $C_D = sf \times N_{rec}$

Table 2.3: Costs clustered vs. unclustered indexes.

In ranged search, both types need the same time to retrieve the relevant indexes: either way, they are always sorted by key and easily accessed since they're all part of the same doubly linked list; we're still assuming that since a B$^+$-tree structure is used, the tree will be pretty shallow. Using clustered indexes, the order in which indexes are found is the same as the order of the actual data on disk: the cost of accessing the data depends only on the number of pages the file is made up of, and we will only have to access sf of them in order. With unclustered indexes, the data is not ordered in the same way as the index. There's no way to know exactly where each record is stored in relation to the pages, so each RID returned by the key retrieval corresponds to an individual page access.

2.3.3 Non-Key Attribute Organizations

Up until now, all operations were done on keys, i.e., attributes that uniquely identify records. In many cases, however, we may be interested in retrieving records based on the values taken by other attributes. For example, imagine a table representing students attending the same school, each uniquely identified by a numeric code, and containing information about their name and age. An operation we may want to find all students within a specific age range, or all students who share the same surname. This section will describe how such operations can be done efficiently.

Specifically, the three types of operations that can be performed on non-key attributes are the equality search, the range search, and the **boolean search**, which consists in the previous operations combined together with boolean logical operators.

Inverted Indexes

Inverted Index

An inverted index Idx on a non-key attribute K of a table R is a sorted collection of entries, each in the form

$$(k_1, n, p_1, p_2, \ldots, p_n)\,,$$

where each value k_i of K is followed by the number of records n with that value, and the **sorted** RID list of these records.

Each entry in the inverted index has variable length, depending on how many records in the table R have the same value for the attribute. Also, RIDs are added or removed as records are added or removed from the table. Despite the need to manage these indexes, they are still widely used, especially for cases in which searches are more common than insertions of deletions.

To evaluate performances, we will introduce these terms: $N_{key}(Idx)$ and $N_{leaf}(Idx)$, which are the number of distinct keys and leaf nodes in the index Idx. Also, all estimates are done assuming that index-key values are uniformly distributed, as well as records, and the index organization is a B$^+$-tree with the RID lists stored in the leaves. Each cost will be broken down into C_I and C_D, as seen before.

For the equality search, the cost of accessing the index is simply $sf(\psi) \times N_{leaf}(Idx)$, or, alternatively, $\lceil N_{leaf}(Idx)/N_{key}(Idx)\rceil$. Here, sf is calculated as:

$$sf(\psi) = \frac{1}{N_{key}(Idx)} \ ,$$

since we're assuming uniform distribution of the values. All RIDs can be found close together since the leaves are sorted by key (i.e., the non-key attribute of the original table).

For C_D, the cost is different whether the data is sorted or not on the index key, so whether the index is clustered or unclustered. If it is unclustered, the operation must read all relevant records with no way to estimate where they are; for each record, the whole page must be read. Potentially, we may need to read an entire RID list worth of records, whose size is given by:

$$E_{rec} = sf(\psi) \times N_{rec}(R) = \left\lceil \frac{N_{rec}(R)}{N_{key}(Idx)} \right\rceil$$

The cost of retrieving the data is:

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

where $\Phi()$ is called **Cardenas' formula**, and is estimated as:

$$\Phi(k, n) = n(1 - (1 - \frac{1}{n})^k) \leq \min(k, n)$$

So, the cost will be less or equal than the smallest term: if there's a lot more records than pages, chances are that a single page may contain multiple relevant records, while if the number of records is lower than that of pages, records will rarely appear together in the same page. If the index is clustered, then the cost is:

$$C_D = \lceil sf(\Psi) \times N_{pag}(R) \rceil$$

Also, if the RID lists are unsorted, then the cost is always E_{rec}.

For the range search, C_I remains the same, except that $sf(\Psi)$ is calculated as the ratio between the interval and the attribute's range. C_D is calculated as the product between the number of index key values, and the number of pages to access to retrieve the records indicated by the RID lists. The first term is $\lceil sf(\Psi) \times N_{key}(Idx) \rceil$, because we will retrieve a certain number of records for each index key included in the range. The second term again depends on whether the index is clustered or unclustered.

If the index is unclustered, then C_D is estimated as:

$$C_D = \lceil sf(\Psi) \times N_{key}(Idx) \rceil \times \left\lceil \Phi(\left\lceil \frac{N_{rec}(R)}{N_{key}(Idx)} \right\rceil, N_{pag}(R)) \right\rceil ,$$

while, if it is clustered, it is:

$$C_D = \lceil sf(\Psi) \times N_{key}(Idx) \rceil \times \left\lceil \frac{1}{N_{key}(Idx)} \times N_{pag}(R) \right\rceil = \lceil sf(\Phi) \times N_{pag}(R) \rceil$$

If the RID lists are unsorted, then the second term is always $\lceil N_{rec}(R)/N_{key}(Idx) \rceil$.

The summary of performances is shown in Table 2.4.

	Eq. Search	**Range Search**
Sorted RID lists, unclustered	$C_I = \frac{1}{N_{key}(Idx)} \times N_{leaves}(Idx)$, $C_D = \Phi(E_{rec}, N_{pag})$	$C_I = \frac{v2 - v1}{v_{max} - v_{min}} \times N_{leaves}(Idx)$, $C_D = sf(\Psi) \times N_{key}(Idx) \times$ $\Phi(\frac{N_{rec}(R)}{N_{key}(Idx)}, N_{pag}(R))$
Sorted RID lists, clustered	$C_I =$ as above, $C_D = \frac{1}{N_{key}(Idx)} \times N_{pag}(R)$	$C_I =$ as above, $C_D = sf(\Psi) \times N_{pag}(R)$
Unsorted RID lists	$C_I =$ as above, $C_D = E_{rec}$	$C_I =$ as above, $C_D =$ $sf(\Psi) \times N_{key}(Idx) \times \frac{N_{rec}(R)}{N_{key}(Idx)}$

Table 2.4: Costs for inverted indexes.

Bitmap Indexes

Bitmap Index

A bitmap index Idx on a non-key attribute K of a table R with N records, is a sorted collection of entries in the form (k_i, B), where each k_i of K is followed by a sequence of N bits such that the j^{th} bit is set to 1 if the j^{th} record has value k_i for attribute K, 0 otherwise.

Bitmap indexes are used in DBMS where data is never updated, such as data warehouses, since operations that modify this type of index can be complex, especially when it's compressed. They can also easily solve multi-attribute queries, since the answer can be found by doing a bit-wise AND between two or more bitmaps.

Indicating with L_k and L_{RID} the amount of bytes needed to store a key k and a RID, and D_{pag} the page size of the leaves, the number of leaves of a full inverted index is:

$$N_{leaf} = \frac{N_{key} \times L_k + N_{rec} \times L_{RID}}{D_{pag}} \approx \frac{N_{rec} \times L_{RID}}{D_{pag}}$$

while the number of leaves for a bitmap index is:

$$N_{leaf} = \frac{N_{key} \times L_k + N_{key} \times N_{rec}/8}{D_{pag}} \approx N_{key} \times \frac{N_{rec}}{D_{pag} \times 8}$$

Using these approximations, if the number of distinct values of the attribute is low, then a bitmap index is more convenient than an inverted index (as seen in Figure 2.6).

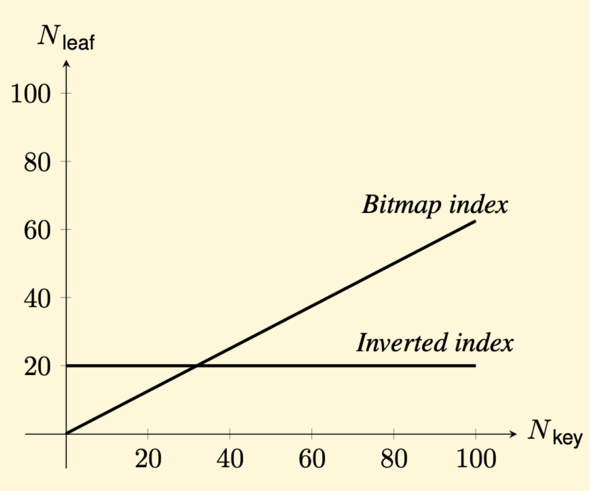

Figure 2.6: Memory usage of bitmap and inverted indexes.

2.3.4 Multidimensional Data Organization

Multidimensional (or spatial) data is used to represent geometric objects and their position in a multidimensional space. Each record represents a point in the space, has a certain number of attributes that each represent the coordinates. Some common queries in multidimensional datasets are searching for points that fall within a specified rectangular area, and searching for a point's nearest neighbor(s). A typical organization with a B$^+$-tree may not be a good solution, since it does not capture "closeness" among points on more than one attribute at a time (the one chosen as key).

A way to solve this issue is to partition the space into areas with the same amount of points, so that each partition can be mapped to a separate page and allow quick retrieval of points that are spatially close together. Consider the dataset represented in Figure 2.7, and suppose that pages have a capacity of 2. The data space is first divided choosing a division value d on one coordinate, so that all points whose attribute value for that coordinate is less than d are inserted into a page, those with a higher value are inserted into the other one. d is usually chosen as the half of the range or the median value. The first split is in Figure 2.8 (a), done on the x axis. If the partitions are still too big to fit into pages, then a split is repeated considering another axis (Figure 2.8 (b)). The splits continue alternating axes until each partition contains a small enough number of records. All the records belonging to the same partition will be found in the same page.

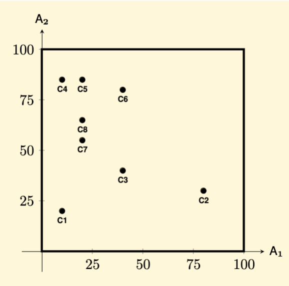

Figure 2.7: Graphical representation of a two-dimensional dataset.

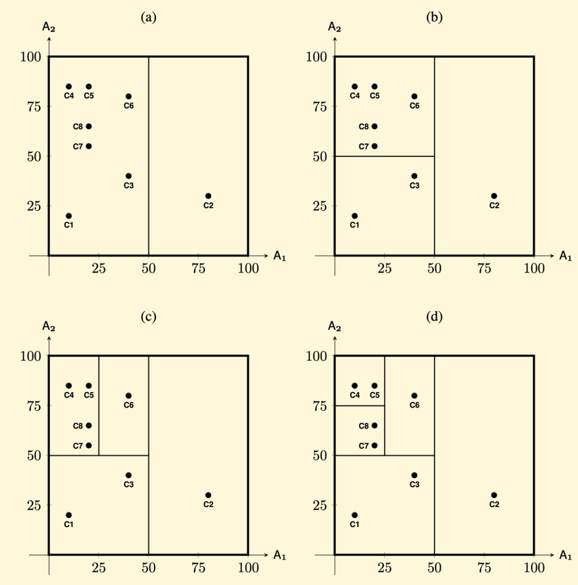

Figure 2.8: Division of the space into partitions.

G-trees

G-trees are the data structure used to store multidimensional indexes, where each partition is identified by a **partition code**. As the space is partitioned, a sort of "decision tree" is built, where each node corresponds to an attribute test condition, alternating the attributes at each level. Partition codes are assigned as follows:

- The initial, intact, region is identified by the empty string;

- After the first split, the two partitions produced are identified with the strings “0” and “1’;
- When each partition of the previous step is split along the other axis, the new partitions will be “00” and “01”, and “10” and “11”, and so on.
- In general, when a partition R is split, the subpartitions will have a code that is equal to the code of the parent and 0/1 appended at the end.

Each partition code is then padded so that they all reach M bits, where M is the maximum number of splits made. The G-tree stores these codes like a B$^+$-tree, where the leaves contain all the codes and the internal nodes alternate pointers to leaves and duplicate keys. After the tree has been constructed, point search, insertion, deletion, and spatial range search can be done efficiently. To search a point P with coordinates (x, y), the partition code of the point is retrieved (if it is present), and the code is then used to search in the G-tree.

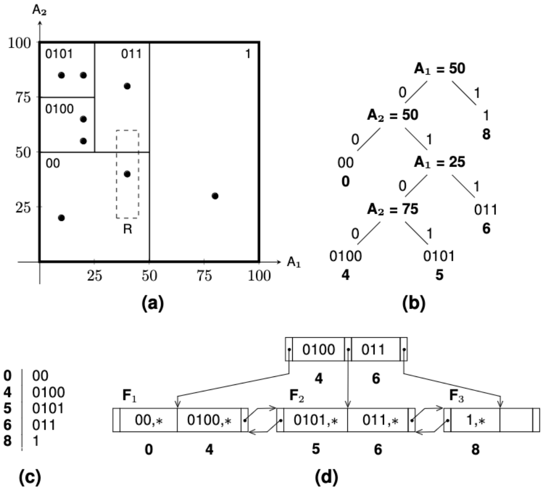

Figure 2.9: Example of partition coding.

Range search is done by specifying a range for each axis. The search starts by identifying the lower left and upper right vertices of the rectangle area; the G-tree is searched for the nodes that contain these vertices, and for each leaf between them, the elements are searched, selecting all leaves that directly intersect with the search region.

For point insertion, first the G-tree is searched to find the partition that should contain the point; then, if that partition/leaf is not full, the point is simply inserted, otherwise, the partition must be split into two. Each partition is associated with the correct partition code, and if needed (the split adds a new level to the split tree) all other codes’ padding is adjusted. The points in the original partition are distributed between the pages referred by the two leaves accordingly, and the parent node is updated with the new pointer and the duplicate key. If an overflow happens in the parent node, the same procedure seen for B^+-trees happens.

To delete a point, it is first searched in the G-tree, and the record is deleted. If the partition becomes empty, its code is removed from the tree. If the partition is the result of a split, and it can be merged with its sibling, then the merge is done and the two partition codes are replaced by the partition code of their parent.

Chapter 3

Access Method Management

The Access Methods Manager provides an interface with several operations to interact with the organizations and indexes implemented by the Storage Structure Manager, so that data can be transferred between main and permanent memory. The language used to implement these operations transform the machine into an **abstract database machine**, called the database management system. Abstract database machines are divided into two parts:

- **Relational Engine**, or abstract machine for the logical data model. Includes modules to support the execution of SQL commands;
- **Storage Engine**, or abstract machine for physical data model. Includes modules to execute operations on the data in permanent memory.

3.1 Storage Engine

The interface of the Storage Engine depends on the data structures used in permanent memory. Normally, it is not directly available to the user, who will instead interact with the Relational Engine which in turn will communicate with the Storage Engine. We will consider an interface inspired by that of the relational system JRS, which stores relations into heap files and provides B$^+$-tree indexes.

Data and Transactions

- $beginTransaction : null \mapsto TransactionId$
- $commit : TransactionId \mapsto null$
- $abort : TransactionId \mapsto null$

- $createDB : Path \times DBName \times TransactionId \mapsto DB$
- $createHF : DB \times Path \times HFName \times TransactionId \mapsto HF$
- $createIdx : DB \times Path \times IdxName \times HFName \times Attr \times Ord \times Unique \times TransactionId \mapsto Idx$
- $dropBD : DBName \times TransactionId \mapsto null$
- $dropHF : HFName \times TransactionId \mapsto null$
- $dropIdx : IdxName \times TransactionId \mapsto null$

Heap File

- $HFopen : DB \times HFName \times TransactionId \mapsto HF$
- $HFCcose : HF \mapsto null$
- $HFgetRecord : HF \times RID \mapsto Record$
- $HFdeleteRecord : HF \times RID \mapsto null$
- $HFupdateRecord : HF \times RID \times FieldNum \times NewField \mapsto null$
- $HFinsertRecord : HF \times Record \mapsto RID$
- $HFgetNPage : HF \mapsto int$
- $HFgetNRec : HF \mapsto int$

Indexes

- $Iopen : DB \times IdxName \times TransactionId \mapsto Idx$
- $Iclose : Idx \mapsto null$
- $IdeleteEntry : Idx \times Entry \mapsto null$ $(Entry = Value \times RID)$
- $IinsertEntry : Idx \times Entry \mapsto null$
- $IgetNKey : Idx \mapsto int$
- $IgetNLeaf : Idx \mapsto int$
- $IgetMin : Idx \mapsto Value$
- $IgetMax : Idx \mapsto Value$

3.2 Access Method Operators

The following operations transfer data between main and permanent memory. Records of a heap file or of an index are accessed by scans: a heap file scan operator reads each record one after the other, while an index scan operator provides a way to efficiently retrieve the RID of the records. Heap file and index scan operators are implemented using a **cursor** (or **iterator**), which is an object with methods that can return one record at a time and move across records. The typical structure of program that scans heap files/indexes is: Here C is the cursor object.

Algorithm 3 Typical structure of program that uses scan operators.

```
while !$C.isDone()$ do
    $Val = C.getCurrent()$
    ...
    $C.next()$
end while
```

Heap File Scan

- $HFSopen : HF \mapsto HFS$
- $HFSisDone : HFS \mapsto bool$
- $HFSgetCurrent : HFS \mapsto RID$
- $HFSnext : HFS \mapsto null$
- $HFSreset : HFS \mapsto null$
- $HFSclose : HFS \mapsto null$

Index Scan

- $ISopen : Idx \times fstKey \times lstKey \mapsto IS$
- $ISisDone : IS \mapsto bool$
- $ISgetCurrent : IS \mapsto null$
- $ISreset : IS \mapsto null$
- $ISclose : IS \mapsto null$

3.3 Physical Plans

When a SQL query must be executed, it is first represented as a **logical plan**, which is a tree representation of the query, and is eventually transformed in a form that can be more efficiently evaluated. This transformed logical plan is then translated into a **physical plan**, which contains as nodes the actual physical operators that can implement that query. Each operator in a plan is an iterator that uses a "pull" interface: when an operator receives a request from above, it "pulls" on its input node(s) and computes the result, returning it to its parent operator. An operator interface provides the necessary methods *open*, *next*, *isDone*, and *close*, implemented using the Storage Engine interface.

Chapter 4

Physical Relational Operators

One of the most important components in a DBMS is the **Query Manager**, which is responsible of scheduling queries and directing them to the correct tables. Part of the Query Manager is the **Query Optimizer**, which has the task of determining how to execute a query in the most efficient way possible, considering the physical parameters involved, the data organization, and the presence or absence of indexes.

This chapter will deal with how different physical operators are implemented, for the following operations:

- Projection;
- Selection;
- Grouping;
- Set operations;
- Join.

Then, it will discuss how the optimizer uses these operators to generate efficient physical plans. In general, the problem will be studied under certain assumptions, illustrated in the next sections.

4.1 Selectivity Factors

The selectivity factor of a condition is an estimate of the percentage of the records in a relation which satisfy that condition. The simplest way to estimate this percentage is by assuming the data is uniformly distributed. The selectivity factor of different conditions in reported in table 4.1. The last column is a constant value that is used if not enough information is known to calculate the actual sf, or when the attribute is non-numeric.

Condition	Calculated sf	Approx. sf
$A = v$	$\frac{1}{N_{key}}$	$\frac{1}{10}$
$A > v$	$\frac{\max(A) - v}{\max(A) - \min(A)}$	$\frac{1}{3}$
$A < v$	$\frac{v - \min(A)}{\max(A) - \min(A)}$	$\frac{1}{3}$
$v_1 < A < v_2$	$\frac{v_2 - v_1}{\max(A) - \min(A)}$	$\frac{1}{4}$
$A_1 = A_2$	$\frac{1}{\max(N_{key}(A), N_{key}(B))}$	$\frac{1}{10}$
$\psi_1 \wedge \psi_2$	$sf(\psi_1) \times sf(\psi_2)$	-
$\psi_1 \vee \psi_2$	$sf(\psi_1) + sf(\psi_2) -$ $sf(\psi_1) \times sf(\psi_2)$	-

Table 4.1: Selectivity factors of different conditions.

In many cases, however, attribute values follow non-uniform distributions, making these estimates wrong. The selectivity factor of a condition can be better approximated by knowing the actual distribution of the data, but storing the information needed to have full knowledge about it would occupy too much space. The solution preferred by DBMSs is to use an histogram with binned ranges of values in order to approximate the actual distribution.

There's two types of histograms: **equi-width** and **equi-height**. Equi-width histograms are obtained by binning values so that each bin has the same amount of elements n. For each bin, the sum of the counts of all elements inside that bin is stored. To find the selectivity factor for an equality search given a value v, it will be given by the sum associated with the bin v belongs to, divided by n. For inequality/range searches, the selectivity factor will consider the sum associated with all the bins completely included in the range, plus the term that is given by the bin(s) corresponding to the extreme(s) of the condition.

The problem with equi-width histograms is that while they provide better approximations than a blind uniform distribution assumption, they are not able to correctly approximate the distribution of data to a sufficiently high precision. This is why equi-height histograms are used instead. These histograms are divided into bins such that the sum of counts of values within each (their "height") is equal across all of them. To store this type of histogram, the only information needed is the number of elements in each bin.

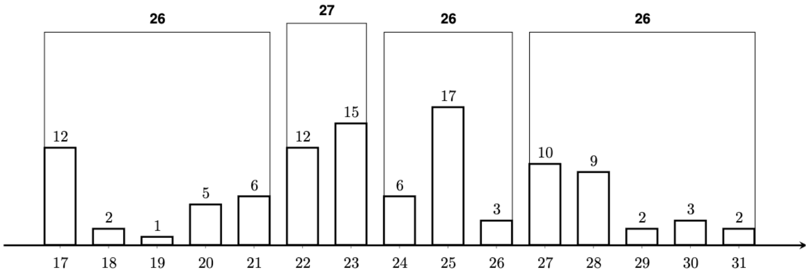

Figure 4.1: An equi-height histogram.

Still, the approximation done by these histograms may not be accurate if the distribution within a bin is not uniform. For example, in Image 4.1, the third bin has follows a Gaussian distribution. If a query requests all records whose values for that attribute is equal to 24, the selectivity factor approximation will be much higher than the real one; if a query instead requests records with value 25, the approximation will be much lower.

4.2 Physical Operators

4.2.1 Operators for Relation

TableScan(R) Returns all the records in R, in the same order as they are stores. It costs

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

The result size is

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

SortScan$(R, \{A_i\})$ Returns all the records in R sorted in ascending order on the attribute A_i. Sorting is done with a merge sort algorithm. It costs

$$C = \begin{cases} N_{pag}(R) & N_{pag}(R) < B \\ 3 \times N_{pag}(R) & N_{pag}(R) \leq B \times (B-1) \\ N_{pag}(R) + 2 \times N_{pag}(R) \times \lceil log_{B-1}(N_{pag}(R)/B) \rceil & \text{else} \end{cases}$$

The result size is

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

IndexScan(R, I) Returns the records of R sorted by the attribute the index I is defined on. It costs

$$C = \begin{cases} N_{leaf}(I) + N_{pag}(R) & \text{if } I \text{ is clustered} \\ N_{leaf}(I) + N_{rec}(R) & \text{if } I \text{ is on a key of } R \\ N_{leaf}(I) + \lceil N_{key}(I) \times \phi(\lceil N_{pag}(R)/N_{pag}(I) \rceil, N_{pag}(R)) \rceil & \text{else} \end{cases}$$

The result size is

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

IndexSequentialScan(R, I) Returns the records of R, stored with the primary organization I, sorted in ascending order on the primary key values. It costs

$$C = N_{leaf}(I) \tag{4.1}$$

The result size is

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

4.2.2 Operators for Projection

Project$(O, \{A_i\})$ Projects the records of O over the attributes $\{A_i\}$. It costs

$$C = C(O)$$

The result size is

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

IndexOnlyScan($R, I, \{A_i\}$) Returns the sorted records of R, projecting them over the attributes $\{A_i\}$ on which the index I is on (or contains them as prefix). It costs

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

If a tuple of values for the attributes $\{A_i\}$ is associated with n different RIDs, it is returned n times. The result will not contain duplicates if the attributes are relation keys (they uniquely identify records). The result size is

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

4.2.3 Operators for Duplicate Elimination

Distinct(O) Returns the records of O eliminating all duplicates. This operator requires that the records of O are **grouped** (if $r_i = r_j$, and $i < l < j$, then $r_i = r_l = r_j$). When a collection of records is sorted, it is also grouped. It costs

$$C = C(O)$$

If there's only one attribute in O, then the result size is

$$E_{rec} = N_{key}(A)$$

If instead it contains multiple attributes, the result size is

$$E_{rec} = \min(|O|/2, \prod_i N_{key}(A_i))$$

This is a pessimistic estimate: it assumes that there is a record for each set of values taken from the attributes in O, but this is often not the actual result. For example, imagine a database containing data about students enrolled at a university, and the two attributes represent their first name and last name respectively: it is unrealistic to expect that there will be a different student for each first name-last name combination, since the two attributes are loosely correlated.

HashDistinct(O) Returns the records of O without duplicates using and hash technique. This technique has two phases: **partitioning** and **duplicate elimination**. Assume the query processor has $B + 1$ buffer pages. In the partitioning phase, for each record in O the hash function h_1 is applied, distributing records uniformly across the B pages. Once a page i is full, it is written to the T_i partition file. At the end of the phase, all records will be scattered across B files, each of which contains records with the same hash value; this means that duplicates are found in the same partition.

In the duplicate elimination phase, the process becomes an intra-partition problem. Each T_i file is read page-by-page, eliminating duplicates using the hash function h_2. A record is deleted when it collides with another record with the same hash value according to h_2 and the two records are identical. Assuming each partition occupies at most B pages, at the end of the partition, the B pages are cleared, and the duplicate elimination is applied to the records in the next partition. If the number of pages is greater than B, then a hash-based projection technique is applied recursively by dividing the partition into subpartitions. This degrades performances. The operator costs:

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

The result size is the same as Distinct, so

$$E_{rec} = N_{key}(A)$$

if there's only one attribute in O, and

$$E_{rec} = \min(|O|/2, \prod_i N_{key}(A_i))$$

if O contains multiple attributes.

4.2.4 Operators for Sort

Sort$(O, \{A_i\})$ Returns the records of O sorted on the attributes $\{A_i\}$. The sorting algorithm used is merge sort, so its cost is

$$C = \begin{cases} C(O) & N_{pag}(R) < B \\ C(O) + 2 \times N_{pag}(O) & N_{pag}(O) \leq B \times (B-1) \\ C(O) + 2 \times N_{pag}(O) \times \lceil log_{B-1}(N_{pag}(O)/B) \rceil & \text{else} \end{cases}$$

The result size is

$$E_{rec} = N_{rec}(O)$$

4.2.5 Operators for Selection

Filter(O, ψ) Returns the records of O that satisfy the condition ψ. It costs

$$C = C(O)$$

The result size is

$$\lceil sf(\psi) \times N_{rec}(O) \rceil$$

IndexFilter(R, I, ψ) Returns the records of R that satisfy the condition ψ using the index I, defined on the attributes involved in ψ, sorted according to I. The condition is a predicate or a conjunction of predicates that only involve the attributes found in the prefix of the index search key.

IndexFilter always appears as a leaf node in a physical plan. This operator uses the index to find the sorted set of RIDs of all records that satisfy the condition, then it retrieves the records from disk. The cost can be broken down as

$$C = C_I + C_D$$

- If the index is clustered:

$$C_I = \lceil sf(\psi) \times N_{leaf}(I) \rceil$$
$$C_D = \lceil sf(\psi) \times N_{pag}(R) \rceil$$

- If the index is unclustered:

$$C_I = \lceil sf(\psi) \times N_{leaf}(I) \rceil$$
$$C_D = \lceil sf(\psi) \times N_{key}(I) \rceil \times \lceil \Phi(\lceil N_{rec}(R)/N_{key}(I) \rceil, N_{pag}(R)) \rceil$$

If the index is defined on a key of R, then

$$C_D = \lceil sf(\psi) \times N_{rec}(R) \rceil$$

The result size is

$$E_{rec} = \lceil sf(\psi) \times N_{rec}(R) \rceil$$

IndexSequentialFilter(R, I, ψ) Returns the sorted records of R, stored with the primary organization I, satisfying the condition ψ, which involves only the attributes of the index search key. It costs

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

The result size is

$$E_{rec} = \lceil sf(\psi) \times N_{rec}(R) \rceil$$

IndexOnlyFilter$(R, I, \{A_i\}, \psi)$ Returns the sorted records of the projection on R returning only the values for $\{A_i\}$ that satisfy ψ, using only the index I. It costs

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

The result size is

$$E_{rec} = \lceil sf(\psi) \times N_{rec}(R) \rceil$$

4.2.6 Operators for Grouping

GroupBy($O, \{A_i\}, \{f_i\}$) Returns the records of O sorted on $\{A_i\}$, applying the aggregation functions $\{f_i\}$. The records in O must already be sorted beforehand. It costs

$$C = C(O)$$

HashGroupBy($O, \{A_i\}, \{f_i\}$) Returns the records of O grouped by $\{A_i\}$, applying the aggregation functions $\{f_i\}$. The records are not sorted on $\{A_i\}$. The grouping is done using two phases, like HashDistinct. In the first phase, called partitioning phase, a partition is created using the hash function h_1; in the second phase, called grouping, the records of each partition are grouped using the hash function h_2 applied to all grouping attributes. When two records with the same grouping attributes are found, a step to compute the aggregate function is applied. The operator costs

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

For both the previous two operators, the result size is calculated as for the duplicate elimination. If there's only one attribute in O, then the result size is

$$E_{rec} = N_{key}(A)$$

If instead it contains multiple attributes, the result size is

$$E_{rec} = \min(|O|/2, \prod_i N_{key}(A_i))$$

4.2.7 Operators for Join

NestedLoop(O_E, O_I, ψ_J) Joins the external operand O_E with the internal operand O_I with the following algorithm:

```
for $r \in O_E$ do
    for $s \in O_I$ do
        if $\psi_J$ then
            If $\psi_J$, add $< r, s >$ to the result.
        end if
    end for
end for
```

It costs

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

The result size is

$$E_{rec} = sf(C_j) \times E_{rec}(O_E) \times E_{rec}(O_I)$$

PageNestedLoop(O_E, O_I, ψ_J) Joins the external operand with the internal operand by scanning O_I once per page of O_E (and not once per record, as for NestedLoop). The algorithm used is the following:

```
for $p_r$ of $O_E$ do
    for $p_s$ of $O_I$ do
        for $r \in p_r$ do
            for $s \in p_s$ do
                If $\psi_J$, add $< r, s >$ to the result.
            end for
        end for
    end for
end for
```

The cost of the operator is

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

The algorithm cost is lower when the external operand is the one with fewer pages. The result size is

$$E_{rec} = sf(C_j) \times E_{rec}(O_E) \times E_{rec}(O_I)$$

BlockNestedLoop(O_E, O_I, ψ_J) Joins the external operand O_E with the internal operand O_I by extending PageNestedLoop using more memory for a group of pages of the external operand. Assume the operands are TableScan of tables R and S, and that the query processor has $B+2$ pages in the buffer. B pages are used for the external operand, 1 page for an input page of S, and 1 page is reserved as the output buffer. For each record r of a page group of R, and for each joining record s of a page in S, $< r, s >$ is written to the output buffer page.

The cost of the operator is

$$C = N_{pag}(R) + \lceil N_{pag}(R)/B \rceil \times N_{pag}(S)$$

The cost is lower if the external relation has fewer pages than the internal one. If the B pages are enough to contain one of the two relations, then the cost is reduced to

$$N_{pag}(R) + N_{pag}(S)$$

The result size is

$$E_{rec} = sf(C_j) \times E_{rec}(O_E) \times E_{rec}(O_I)$$

This operator is not convenient to use when the operators require too many pages (i.e., $N_{pag}R \geq B^2$).

IndexNestedLoop(O_E, O_I, ψ_J) This operator requires that there is an index on the join column of the internal operand, and performs a join with the following algorithm:

```
for $r \in O_E$ do
    for $s \in$ IndexFilter($O_I, I, O_E.e1 = O_I.i1$) do
        If $\psi_J$, add $< r, s >$ to the result.
    end for
end for
```

It costs

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

where C_I and C_D are the costs to retrieve the relevant index records and the data from disk. If the internal operand is an IndexFilter(S, I, ψ_J), the result size is

$$E_{rec} = \lceil sf(\psi_J) \times E_{rec}(O_E) \times N_{rec}(S) \rceil$$

if instead it is a Filter(IndexFilter(S, I, ψ_J), ψ), the result size is

$$E_{rec} = \lceil sf(\psi_J) \times E_{rec}(O_E) \times (sf(\psi) \times_{rec} (S)) \rceil$$

MergeJoin(O_E, O_I, ψ_J) This operator requires that O_E and O_I are sorted on the same join attributes, and that in the join condition, $O_E.A_i$ is a key of O_E. Since this join attribute has distinct values in O_E, the algorithm reads the records of O_E one by one, and reads all records of O_I with the same values (which will be found one after the other). This operator costs

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

The result size is

$$E_{rec} = \lceil sf(\psi_J) \times E_{rec}(O_E) \times E_{rec}(O_I) \rceil$$

HashJoin(O_E, O_I, ψ_J) Returns the join result with a hash technique in two phases. In the first phase, called partitioning phase, the records of both operands are partitioned using the hash function h_1, similarly to HashDistinct. In the second phase, called **probing** (or **matching**), for each B_i partition, the records of O_E are read and inserted into the buffer hash table with B pages using the hash function h_2. The records of O_I are read one page at a time, h_2 is applied to them, and if there is a match with the records in O_E, the joined record is added to the result.

Assuming $N_{pag}(O_E)/B < B$ and that the pages are uniform, the cost of the operator is

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

where $(C(O_E) + C(O_I) + (N_{pag}(O_E) + N_{pag}(O_I))$ is the cost of the partitioning phase, and $(N_{pag}(O_E) + N_{pag}(O_I))$ is the cost of the probing phase. However, if the pages are not uniform, the resulting partitions will not have the same size, they may not fit in B. The cost can be generalized to

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

If $N_{pag}(O_E) < B$, the cost is 0.

The result size is

$$E_{rec} = \lceil sf(\psi_J) \times E_{rec}(O_E) \times E_{rec}(O_I) \rceil$$

Bibliography

[1] A. Albano, D. Colazzo, G. Ghelli, and R. Orsini. *Relational DBMS Internals*. 2020.