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.

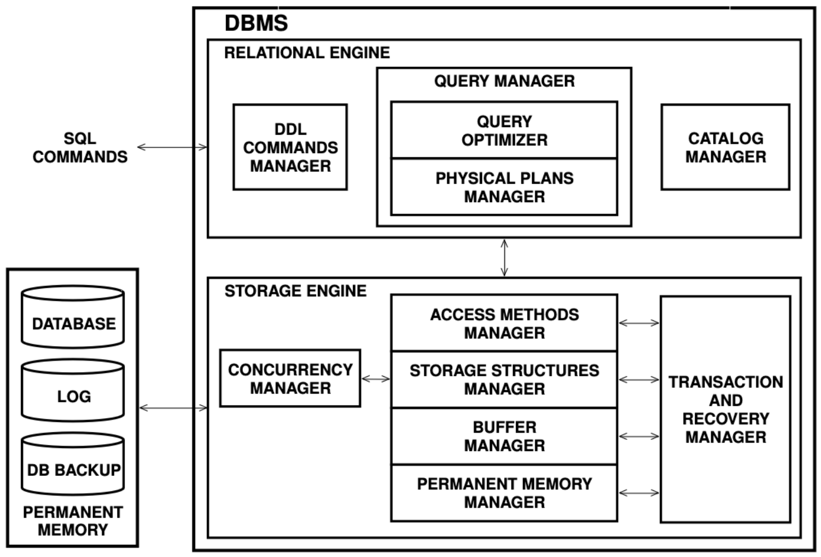

Figure 2.1: The architecture of a DBMS.

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.

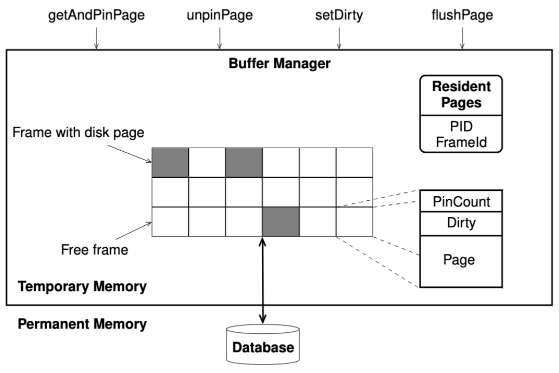

Figure 2.2: The components of the Buffer Manager.

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 Data Organizations

2.2.1 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:

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

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

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.

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 $C_s + 1$.

Bibliography

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