Relational DBMS Internals

Antonio Albano, Giorgio Ghelli
University of Pisa
Department of Computer Science
tonio.albano@gmail.com giorgio.ghelli@gmail.com

Collaborators

Dario Colazzo
University Paris-Dauphine
LAMSADE

Renzo Orsini
University of Venezia
Department of Environmental Sciences,
Informatics and Statistics

February 13, 2015
Revision, February 12, 2021

Contents

Preface **VII**

1 DBMS Functionalities and Architecture **1**
1.1 Overview of a DBMS . . . 1
1.2 A DBMS Architecture . . . 2
1.3 The JRS System . . . 4
1.4 Summary . . . 5

2 Permanent Memory and Buffer Management **7**
2.1 Permanent Memory . . . 7
2.2 Permanent Memory Manager . . . 9
2.3 Buffer Manager . . . 10
2.4 Summary . . . 11

3 Heap and Sequential Organizations **15**
3.1 Storing Collections of Records . . . 15
3.2 Cost Model . . . 18
3.3 Heap Organization . . . 19
3.4 Sequential Organization . . . 19
3.5 Comparison of Costs . . . 20
3.6 External Sorting . . . 21
3.7 Summary . . . 23

4 Hashing Organizations **27**
4.1 Table Organizations Based on a Key . . . 27
4.2 Static Hashing Organization . . . 28
4.3 Dynamic Hashing Organizations . . . 30
4.4 Summary . . . 35

5 Dynamic Tree-Structure Organizations **39**
5.1 Storing Trees in the Permanent Memory . . . 39
5.2 B–trees . . . 40
5.3 Performance Evaluation . . . 43
5.4 B^+–trees . . . 46
5.5 Index Organization . . . 47
5.6 Summary . . . 50

6 Non-Key Attribute Organizations **53**
6.1 Non-Key Attribute Search . . . 53
6.2 Inverted Indexes . . . 54
6.3 Bitmap indexes . . . 58
6.4 Multi-attribute Index . . . 61
6.5 Summary . . . 61

7 Multidimensional Data Organizations 63
7.1 Types of Data and Queries . . . 63
7.2 G–trees . . . 66
7.3 B*–trees * . . . 70
7.4 Summary . . . 74

8 Access Methods Management 77
8.1 The Storage Engine . . . 77
8.2 Operators on Databases . . . 78
8.3 Operators on Heap Files . . . 78
8.4 Operators on Indexes . . . 79
8.5 Access Method Operators . . . 79
8.6 Examples of Query Execution Plans . . . 80
8.7 Summary . . . 81

9 Transaction and Recovery Management 83
9.1 Transactions . . . 83
9.2 Types of Failures . . . 87
9.3 Database System Model . . . 88
9.4 Data Protection . . . 89
9.5 Recovery Algorithms . . . 92
9.6 Recovery Manager Operations . . . 96
9.7 Recovery from System and Media Failures . . . 98
9.8 The ARIES Algorithm * . . . 100
9.9 Summary . . . 101

10 Concurrency Management 105
10.1 Introduction . . . 105
10.2 Histories . . . 106
10.3 Serializable History . . . 108
10.4 Serializability with Locking . . . 113
10.5 Serializability without Locking . . . 117
10.6 Multiple-Granularity Locking * . . . 118
10.7 Locking for Dynamic Databases * . . . 119
10.8 Summary . . . 121

11 Implementation of Relational Operators 125
11.1 Assumptions and Notation . . . 125
11.2 Selectivity Factor of Conditions . . . 129
11.3 Physical Operators for Relation (R) . . . 134
11.4 Physical Operator for Projection (π^b) . . . 135
11.5 Physical Operators for Duplicate Elimination (δ) . . . 135
11.6 Physical Operators for Sort (τ) . . . 137
11.7 Physical Operators for Selection (σ) . . . 137
11.8 Physical Operators for Grouping (γ) . . . 141
11.9 Physical Operators for Join $(\bowtie)$ . . . 142
11.10 Physical Operators for Set and Multiset Union, Intersection and Difference . . . 151
11.11 Summary . . . 152

12 Query Optimization 155
12.1 Introduction . . . 155
12.2 Query Analysis Phase . . . 156

12.3 Query Transformation Phase . . . 156
12.4 Physical Plan Generation Phase . . . 171
12.4.2 Single-Relation Queries . . . 174
12.4.3 Multiple-Relation Queries . . . 175
12.4.4 Other Types of Queries . . . 180
12.5 Summary . . . 191

13 Physical Database Design and Tuning 195
13.1 Physical Database Design . . . 195
13.2 Database Tuning . . . 206
13.3 DBMS Tuning . . . 212
13.4 Summary . . . 213

A Formulary 217

Bibliography 223

Subject Index 227

PREFACE

The preface to the previous edition of this book in Italian of 2001 starts with "After ten years of the publication of the book *Databases: structures and algorithms*, the evolution of the technology of databases and the new organization of university teaching suggest a substantial revision of the material."

Today, another reason that suggested to review the material, and to write it in English, has been the internationalisation of the master's degree programs offered by the Department of Computer Science, University of Pisa, which have the participation of students with different backgrounds who have had an introductory course in databases in different universities.

Consequently, the goal in writing this new shorter edition of the textbook is to focus on the basic concepts of classical centralized DBMS implementation. Database systems occupy a central position in our information-based society, and computer scientist and database application designers should have a good knowledge about both the theoretical and the engineering concepts that underline these systems to ensure the desired application performance.

The book starts with an analysis of relational DBMS architecture and then presents the main structures and algorithms to implement the modules for the management of permanent memory, the buffer, the storage structures, the access methods, the transactions and recovery, the concurrency, the cost-based query optimization. Finally, an overview of physical database design and tuning is presented.

An original aspect of the material is that, to illustrate many of the issues in query optimization, and to allow the students to practise with query optimization problems, the solutions adopted for the relational system **JRS** (*Java Relational System*) will be used, the result of a project developed in Java at the Department of Computer Science, University of Pisa, by A. Albano with the collaboration of several students, with their degree thesis, and of R. Orsini.

Organization

The material of the previous edition has been reduced and updated in almost all the chapters, to make the book more suitable for use by the students of an advanced database course, who have only had an introductory undergraduate course in databases. Moreover, it has been decided to make this edition available for free on the web.

Chapter 1 presents the architecture of a relational DBMS and the characteristics of the modules that compose it. **Chapter 2** presents the characteristics of the permanent memory and buffer managers. **Chapter 3** shows how to store data in permanent memory using files and presents the simplest data organizations, the *heap* and *sequential*. It also shows an approach to performance evaluation of data organizations.

Chapter 4 describes the data *primary organizations*, *static* or *dynamic*, based on hashing techniques. **Chapter 5** continues the description of *primary dynamic organizations* using tree structures, and the *key secondary organizations* with clustered and unclustered indexes. **Chapter 6** describes the *non-key secondary organizations* with indexes to support search queries to retrieve small subsets of records, while **Chapter 7** presents the basic idea on *multi-dimensional data organizations*. **Chapter 8** describes the access methods of the JRS Storage Engine to implement the physical operators used by the query manager. **Chapters 9** and **10** describe transaction recovery and concurrency control management techniques. **Chapters 11** and **12** describe the JRS physical operators to implement relational algebra operators, and then how they are used by the query optimizer to generate a physical query plan to execute a SQL query. Finally, **Chapter 13** describes solutions for the physical database design and tuning problem.

Sections marked as advanced, using the symbol "*", may be omitted if so desired, without a loss of continuity.

Acknowledgments

We would like to thank the following students, who provided useful feedback on draft versions of the book: P. Barra. G. Galanti, G. Lo Conte, G. Miraglia, L. Morlino and A. Vannini.

A. A.
G. G.
D. C.
R. O.

Chapter 1

DBMS FUNCTIONALITIES AND ARCHITECTURE

This chapter is an introduction to the structure of a typical centralized DBMS (*Data Base Management System*) based on the relational data model. A brief description is given of the basic functionalities of the main DBMS modules, and of the problems to be addressed in their implementation, which will be discussed in the following chapters.

1.1 Overview of a DBMS

The most common use of information technology is to store and retrieve information represented as data with a predefined structure and fields with different formats, such as numbers, characters, text, images, graphics, video and audio. The technology used is mainly that of the databases, now available on any type of computers.

A database (DB) is a collection of homogeneous sets of data, with relationships defined among them, stored in a permanent memory and used by means of a DBMS, a piece of software that provides the following key features:

1. A language for the database ***schema*** definition, a collection of definitions which describe the data structure, the restrictions on allowable values of the data (***integrity constraints***), and the relationships among data sets. The data structure and relationships are described in the schema using suitable abstraction mechanisms that depend on ***data model*** adopted by the DBMS.
2. The data structures for the storage and efficient retrieval of large amounts of data in permanent memory.
3. A language to allow authorized users to store and manipulate data, interactively or by means of programs, respecting the constraints defined in the schema, or to rapidly retrieve interesting subsets of the data from a specification of their features.
4. A ***transactions*** mechanism to protect data from hardware and software malfunctions and unwanted interference during concurrent access by multiple users.

Databases and DBMSs can be studied from different points of view depending on the needs of people who must use them. Leaving aside the application's end-users, who are not required to know nor understand the underlying DBMS, the other users can be classified into the following categories:

- ***Non-programmer users***: they are interested in how to use a DBMS interactively to store, modify and retrieve data organized in a DB.
- ***DB Designers***: they are interested in how to design a DB and the applications that use them.
- ***DB Application developers***: they are interested in how to use a DBMS from programs to develop applications that allow non-specialist users to perform predefined tasks.
- ***DB Administrators***: they are interested in how to install, run and tune a DB to ensure desired performance for applications that use the data.
- ***DBMS Developers***: they are interested in how to design and build the DBMS product using the fundamental structures and algorithms suitable for realizing its capabilities.

In this book the focus is on how to implement a DBMS, assuming that the reader already has a working knowledge of databases at least according to the first two points of view, to the level of depth presented, for example, in [Albano et al., 2005] or in other texts of a similar nature cited in the bibliography. As a point of reference we consider the relational DBMS, studied since the seventies and for which solutions have been proposed that have become the classic reference point for all other types of database systems.

The knowledge of the structures and algorithms discussed in the following is useful not only for those who will implement modules with features typical of those provided by a DBMS, but also for application designers or database administrators. For example, the knowledge of the principles of query optimization is important both to improve the performance of applications by formulating SQL queries with a better chance of being efficiently executed, and to improve database logical and physical design.

In the following we will present an architecture for relational DBMSs and a brief description of the functionality of various modules, the implementation of which will be the subject of later chapters.

1.2 A DBMS Architecture

A simplified model of the architecture of a centralized relational DBMS provides the following basic components (Figure 1.1):

- The ***Storage Engine***, which includes modules to support the following facilities:
  - The ***Permanent Memory Manager***, which manages the page allocation and de-allocation on disk storage.
  - The ***Buffer Manager***, which manages the transfer of data pages between the permanent memory and the main memory.
  - The ***Storage Structures Manager***, which manages the data structures to store and retrieve data efficiently.
  - The ***Access Methods Manager***, which provides the storage engine operators to create and destroy databases, files, indexes, and the data access methods for table scan, and index scan.
  - The ***Transaction and Recovery Manager***, which ensures that the database's consistency is maintained despite transaction and system failures.
  - The ***Concurrency Manager***, which ensures that there is no conflict between concurrent access to the database.

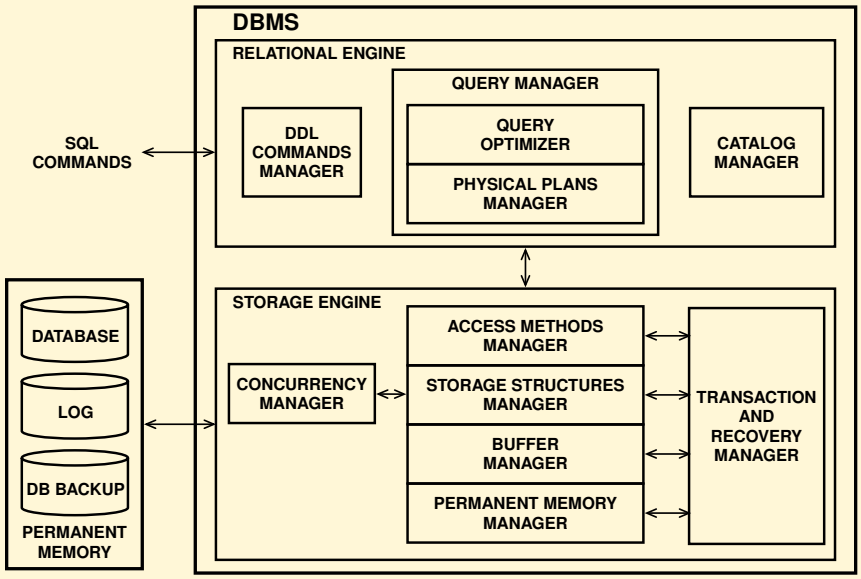


Figure 1.1: Architecture of a DBMS

– The ***Relational Engine***, which includes modules to support the following facilities:
  – The ***Data Definition Language (DDL) manager***, which processes a user's database schema definition.
  – The ***Query Manager***, which processes a user's query by transforming it into an equivalent but more efficient form, thus finding a good strategy for its execution.
  – The ***Catalog Manager***, which manages special data, called *metadata*, about the schemas of the existing databases (views, storage structures and indexes), and security and authorization information that describes each user's privileges to access specific database, relations and views, and the owner of each of them. The catalog is stored as a database which allows the other DBMS modules to access and manipulate its content.

In real systems the functionalities of these modules are not as clearly separated as the figure shows, but this diagram helps in understanding the purposes of each of them.

Let us briefly examine the modules that will be considered in the following chapters, by describing the level of abstraction provided and the features that are made available to other modules.

The ***Permanent Memory Manager*** provides a vision of the memory as a set of databases each consisting of a set of files of physical pages of fixed size. It allows other levels to use the permanent memory, abstracting from the different ways used by operating systems to manage files. This module is discussed in Chapter 2.

The performance of a database query depends on the number of pages transferred from the disk in temporary memory. The execution cost of some queries can be reduced using a buffer capable of containing many pages, so that, while executing the queries, if there are repeated accesses to the same page, the likelihood that the desired page is already in memory increases. The ***Buffer Manager*** is the module that takes

care of properly managing this limited resource, and the transfer of pages between temporary and permanent memories, thus offering to the other levels a vision of permanent memory as a set of usable pages in the main memory, abstracting from when they are transferred from the permanent memory to the temporary buffer memory, and vice versa. Buffer management is discussed in Chapter 2.

The ***Storage Structures Manager*** provides the other system levels with a view of the permanent data organized into collections of records and indexes, abstracting from the structures used to store them in the permanent memory (heap, index sequential, hash static or dynamic, tree indexes, etc.). Permanent data organization and indexes are considered in Chapters 3 through 7.

The ***Access Methods Manager*** provides a vision of permanent data organized in collections of records accessible one after the other in the order in which are stored, or through indexes, abstracting from their physical organization. The interface of this level is the one that is used to translate the SQL commands into low-level instructions (*access plans*). This module is discussed in Chapter 8.

The ***Transaction and Recovery Manager*** provides the other system levels with a vision of the permanent memory as a set of pages in temporary memory without regard to failures (interruption of the execution of transactions, interruption of operation of the system, devices failures) and thus ensuring that the data always reflects the permanent effects of all the transactions completed normally. This module is discussed in Chapter 9.

The ***Concurrency Manager*** provides the other system levels with a vision of permanent memory as a set pages in memory without regard to concurrent access, thus ensuring that the concurrent execution of several transactions takes place as if they were executed one after the other, in some order, by avoiding undesired interference. This module is discussed in Chapter 10. The set of modules including transaction and recovery manager, the concurrency manager, the buffer manager, and the permanent memory manager is also called *Storage engine*.

The ***Query Manager*** provides a vision of permanent data as a set of relational tables on which a user operates with SQL commands. The tasks of the SQL manager commands are: ensure that only authorized users will use the data, manage the metadata catalog, and translate queries into optimized access plans. The basic ideas of query processing and query optimization are discussed in detail in Chapters 11 and 12.

Once the main features of the relational DBMS modules have been presented, Chapter 13 discusses various aspects of database tuning, including the choice of indexes and methods for tuning the schema, to achieve the desired performance of the applications that use the data.

Finally, there are other interesting DBMS features that are beyond the scope of this book: (a) distribution and parallelism, (b) the extension of the relational data model with object orientation.

1.3 The JRS System

The implementation of the relational DBMS modules will be discussed first in general and then with respect to the solutions adopted for the system **JRS** (*Java Relational System*), developed in Java at the Department of Computer Science, University of Pisa, by A. Albano with the degree thesis of several students[1] and the collaboration

1. Lorenzo Brandimarte, Leonardo Candela, Giovanna Colucci, Patrizia Dedato, Stefano Fantechi, Stefano Dinelli, Martina Filippeschi, Simone Marchi, Cinzia Partigliani, Marco Sbaffi and Ciro Valisena.

of R. Orsini.

A unique feature of the system is that it has been designed to support both the teacher and student to experiment not only with the SQL language to query a database but also (1) to analyze the logical and physical query plans generated by different cost-based query optimizers, (2) to experiment with graphical editors both the execution of a logical plan defined with relational algebra, and the execution of physical plans defined with the physical operators of the database system. The software with examples of databases and logical and physical plans, is downloadable for free at this URL: http://fondamentidibasididati.it

1.4 Summary

1. DBMS is the most widespread technology for managing permanent collections of structured data. A DBMS is a centralized or distributed system that enables us (*a*) to define database schemas, (*b*) to choose the data structures for storing and accessing data, (*c*) to store, retrieve and update data, interactively or with programs, by authorized users and within the constraints defined in the schema. The data managed by a DBMS is a shared resource, available for concurrent use and is protected from hardware and software failures.
2. The main modules of a DBMS are the *Storage Structures Manager*, the *Transaction Manager*, and the *Query Manager*.
3. A relational database is organized in terms of relations (or tables) of tuples (or records) on which appropriate algebraic operators are defined.
4. SQL is the standard language for relational DBMSs to define and use databases. The efficient execution of SQL queries is one of the fundamental problems resolved by the relational DBMSs, determining their diffusion on every type of computer.

Bibliographic Notes

There are many books that deal with the problem of DBMS implementation at different levels of detail. Among the most interesting ones there are [Ramakrishnan and Gehrke, 2003], [Silberschatz et al., 2010], [O'Neil and O'Neil, 2000], [Kifer et al., 2005].

A book dedicated to the implementation of relational database systems is [Garcia-Molina et al., 1999], while [Gray and Reuter, 1997] gives details about the implementation of the relational storage engine.

Chapter 2

PERMANENT MEMORY AND BUFFER MANAGEMENT

The first problem to be solved in implementing a DBMS is to provide a level of abstraction of the permanent memory that makes the other modules of the system independent of its characteristics and of those of the storage system. The desired abstraction is achieved with the *Permanent Memory Manager* and *Buffer Manager*. The functionality of these modules is presented after a description of the most significant characteristics of magnetic disk memories, which are those generally used for each type of processing system.

2.1 Permanent Memory

The memory managed by a DBMS is usually organized in a two-level hierarchy: the temporary memory (or main) and the permanent memory (or secondary). The characteristics of the two memories are:[1]

1. Main memory
   (a) Fast access to data (about 10-100 ns).
   (b) Small capacity (some gigabyte).
   (c) Volatile (the information stored is lost during power failures and crashes).
   (d) Expensive.
2. Permanent memory with magnetic disks
   (a) Slow access to data (about 5-10 ms access time for a block).
   (b) Large capacity (hundreds of gigabytes).
   (c) Non volatile (persistent).
   (d) Cheap.
3. Permanent memory with NAND flash memory
   (a) Relatively fast access to data (about 100 μs to read, 200 μs to write and some ms to erase a block).
   (b) Medium capacity (tens of gigabytes).

1. The values used as typical may become unrealistic with the rapid development of the technology.

(c) Non volatile (persistent).
(d) Relatively expensive.

Although the costs of the three types of memory reduce continuously, their ratio remains constant, with a temporary memory which costs several tens of times more than a permanent memory of the same capacity.

The flash memory, with the decrease in the cost and the increase in their capacity, is destined to establish itself for personal computers as an alternative to magnetic disks.

However, they raise new problems in the implementation of DBMSs due to some peculiarities of the operations of this type of memory (Figure 2.1).

	Access Time		
Memory	**Read**	**Write**	**Erase**
Magnetic Disk	12,7 ms (2 KB)	13,7 ms (2 KB)	
NAND Flash	80 μs (2 KB)	200 μs (2 KB)	1,5 ms (128 KB)
RAM	22,5 ns	22,5 ns	

Figure 2.1: Characteristics of the three types of memory

The reading and writing of data (with writing that requires twice the time of reading) are faster than those of magnetic disks, but the rewriting of data is problematic because the data must be deleted first, a slow operation that requires a few ms. To make matters worse, there is another phenomenon: the erasing of data concerns several blocks (64), called a *memory unit*, which should all be read to be deleted, in order to rewrite the memory unit. In addition, this type of memory becomes unreliable after 100 000 cycles of cancellations/rewrites and their use for DBMS requires new solutions for the management of changing data structures and for transaction management. These themes are still the subject of research and for this reason the topic is outside the scope of this text, and in the following we only consider magnetic disks as permanent memory.

A magnetic disk is composed of a pile of p *platters* with concentric rings called *tracks* used to store data, except two outer surfaces of the first and last platters. Typical platter diameters are 2,5 inches and 3,5 inches. The rotation speed of the disk pack is continuous with values between 5000 and 15 000 rotations per minute.

A *track* is the part of the disk that can be used without moving the read head and it is divided in *sectors* of the same size, which are the smallest unit of transfer allowed by the hardware and cannot be changed. Typical values for a sector size are 512, 1 KB, 2 KB or 4 KB. There are from 500 to 1000 sectors per track and up to 100 K tracks per surface. The tracks, while being of variable size, have the same capacity because the sectors have a different storage density.

A track is logically divided in *blocks* of fixed size, which are unit of data transferred with each I/O operation. The block size is a multiple of the sector size and typical values are 2048 and 4096 bytes, but there are systems that use larger values.

The disk driver has an array of *disk heads*, one per recorded surface. Each head is fixed on a movable arm that displaces the head horizontally on the entire surface of a disk. When one head is positioned over a track, the other heads are in identical positions with respect to their platters.

A *cylinder* is the set of tracks of the surfaces of the disks that can be used without moving the heads. Once positioned the heads on the tracks of a cylinder, only one of them at a time can be made active for transferring data, but the passage of reading from one track to another on the same cylinder requires a very short time and can be considered instantaneous. Instead, the passage from a cylinder to another requires the mechanical displacement of the heads and is considered the slowest operation on the disk.

The time to read or write a block, called the *access time*, has the following components:

- The *seek time* t_s, is the time needed to position the disk heads over the cylinder containing the desired block. This time can range between 5 and 20 ms.
- The *rotational latency* t_r, is the time spent waiting for the desired block to appear under the read/write head. This time depend on the rotation speed of the disk. Typical values are 3 ms for a rotation speed of 10 000 *rpm* (rotations per minutes).
- The *transfer time* t_b, is the time to read or write the data in the block once the head is positioned.

The access time to transfer a block to the temporary memory is then $(t_s + t_r + t_b)$, while the time to transfer k contiguous blocks is $(t_s + t_r + k \times t_b)$.

The access time usually dominates the time taken for database operations. To minimize this time, it is necessary to locate data records strategically on disk. Records frequently used together should be stored close together, e.g. depending on their number, on the same block, the same track, the same cylinder, or adjacent cylinders.

Although many database applications adopt a two levels memory hierarchy, there are also solutions with one or three levels of memory.

In the first case, the database is managed in main memory (*main memory DBMS*). This solution is made possible by the reduction of the cost of main memories, and is used for applications that require fast access to data.

Three levels of memory are needed for applications that use large amounts of data (e.g. millions of gigabytes) not storable in the secondary storage and thus require tertiary storage such as optical disks or tapes, slower than magnetic disk, but with a much larger capacity.

Finally, a solution often adopted is RAID technology (*Redundant Arrays of Independent Disks*), which uses the redundancy, for guarding against data loss due to malfunctions of the disks, and the parallelism, for improving the performances. There are seven ways to organize data in the disks, known as *RAID levels*, with different characteristics in terms of read/write time, data availability and cost [Ramakrishnan and Gehrke, 2003; Garcia-Molina et al., 1999].

2.2 Permanent Memory Manager

The *Permanent Memory Manager* takes care of the allocation and de-allocation of pages within a database, and performs reads and writes of pages to and from the disk. It provides an abstraction of the permanent memory in terms of a set of databases each made of a set of files with page-sized blocks of bytes, called *physical pages*. A database is a directory containing files for the catalog, relations and indexes.

The physical pages of a file are numbered consecutively starting from zero, and their number can grow dynamically with the only limitation given by available space in the permanent memory.

When a physical page is transferred to the main memory it is called a *page* and, as we will see, it is represented with a suitable structure. For this reason, sometimes

we use the term *page* as a synonym for physical page, and it will be clear from the context whether we refer to a block of bytes or to a more complex structure.

Each collection of records (table or index) of a database is stored in a logical file which in turn can be realized as a separate file of the operating system or as a part of a single file in which the entire database is stored.

2.3 Buffer Manager

The role of the *Buffer Manager* is to make pages from the permanent memory available to transactions in the main memory. It is the responsibility of the buffer manager to allow transactions to get the pages they need, while minimizing disk access operations, by implementing a page replacement strategy.

As we will see later, the performance of operations on a database depends on the number of pages transferred to temporary memory. The cost of some operations may be reduced by using a buffer capable of containing many pages, so that, during the execution of the operation, if there are repeated access to the same page, the likelihood that the desired page is already in the memory increases.

The buffer manager uses the following structures to carry out its tasks (Figure 2.2):

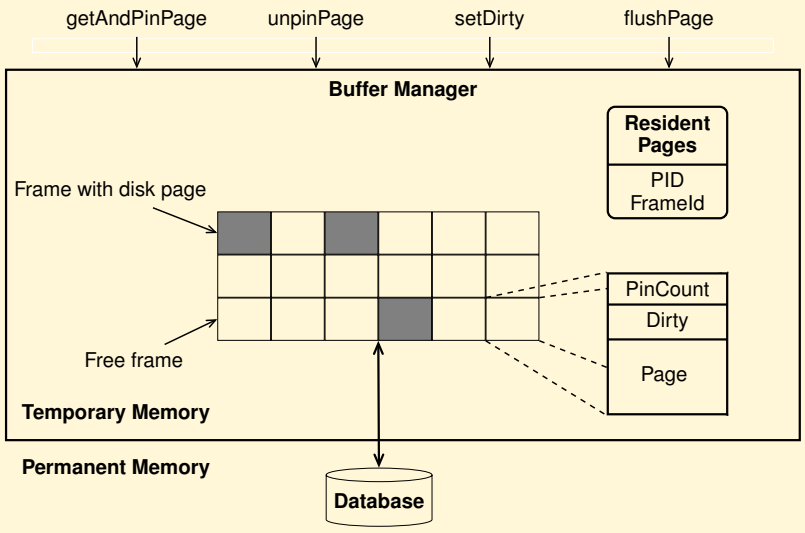


Figure 2.2: The Buffer Manager

1. The *buffer pool*, an array of *frames* containing a copy of a permanent memory page and some bookkeeping information. The buffer pool has a fixed size, therefore, when there are no free frames, to copy a new page from the permanent memory an appropriate algorithm is used in order to free a frame.

   To manage the buffer pool, in a frame are also stored two variables: the *pin count* and *dirty*. Initially the *pin count* for every frame is set to 0, and the boolean variable *dirty* is false. The *pin count* stores the number of times that the page currently in the frame has been requested but not released. Incrementing the *pin count* is called *pinning* the requested page in its frame. The boolean variable *dirty* indicates whether the page has been modified since it was brought into the buffer pool from the permanent memory.

2. A hash *resident pages* table, called *directory*, used to know if a permanent memory page, with a given page identifier PID, is in the buffer pool, and which frame contains it.

The buffer manager provides the following primitives to use the pages in the buffer pool:

1. *getAndPinPage*(P). If a frame contains the requested page, it increments the pin count of that frame and returns the page identifier to the requester (note that the other levels of the system operate directly on a page in the buffer, and not on a copy of it).

   If the requested page is not in the buffer pool, it is brought in as follows:

   (a) A free frame (with pin count 0) is chosen according to the buffer management's replacement policy. Several alternative solutions have been studied, but most systems use some enhancements to the *Least Recently Used* (LRU) to tune the replacement strategy via query plan information or the page type (e.g. the root of a B^+–tree).

   If the frame chosen for replacement is dirty, the buffer manager *flush* it, i.e. write out the page that it contains to the permanent memory.

   If there are no free frames an exception is raised.

   (b) The requested page is read into the frame chosen for replacement and *pinned*, i.e. the pin count is set to 1, and the boolean variable dirty is set to false. The buffer manager will not read another page into a frame until its pin count becomes 0, that is, until all requestors of the page have unpinned it.

   (c) The resident pages table is updated, to delete the entry for the old page and insert an entry for the new page, and the page identifier is returned to the requester.

2. *setDirty*(P). If the requestor modifies a page, it asks the buffer manager to set the dirty bit of the frame.
3. *unpinPage*(P). When the requestor of a page releases the page no longer needed, it asks the buffer manager to unpin it, so that the frame containing the page can be reused if the pin count becomes 0.
4. *flushPage*(P). The requestor of a page asks the buffer manager to write the page to the permanent memory if it is dirty.

As we will see, the decision to unpin or flush a page are taken by the *Transaction and Recovery Manager* and not by the requestor of a page.

2.4 Summary

1. The computer memory is organized as a memory hierarchy: primary or main, secondary and tertiary. The primary memory is volatile and provides fast access to data. The secondary memory is persistent and consists of slower devices, such as magnetic disks. The tertiary memory is the slowest class of storage devices (optical disks and tapes) for large data files.
2. DBMSs typically use magnetic disks because they are inexpensive, reliable and with a growing capacity. The unit of transfer is a block of bytes of fixed size, stored in tracks on the surfaces of the platters. The access time to a block depends on its location on the disk relative to the position of read heads. The access time is

about 10 ms, depending on the time to position the heads (*seek time*), the time for the block to pass under the heads (*rotational delay*), and the time of data transfer (*transfer time*).

3. In a DBMS the *Permanent Memory Manager* stores data in files and provides an abstraction of memory as a set of files of data pages.
4. In a DBMS the data page requests are handled by the *Buffer Manager*, which transfers the request to the *Permanent Memory Manager* only if the page is not already in the buffer pool. The pages in use are pinned and cannot be removed from the buffer pool. The modified pages are considered *dirty* and, when they are not pinned, they are written back to the permanent memory.

Bibliographic Notes

The data storage on permanent memories and the buffer management are discussed in [Ramakrishnan and Gehrke, 2003; Garcia-Molina et al., 1999; Silberschatz et al., 2010] and in [Gray and Reuter, 1997], a text with full implementation details on structures and algorithms to implement DBMS. Interesting papers about buffer management are [Sacco and Schkolnick, 1986; Chou and Witt, 1985].

Exercises

Exercise 2.1 A disk has the following characteristics:

– bytes per sector (bytes/sector) = 512
– sectors per track (sectors/track) = 50
– tracks per surface (tracks/surface) = 2000
– number of platters = 5
– rotation speed = 5400 rpm (rotation/minutes)
– average seek time = 10 ms

Calculate the following parameters.

1. Tracks capacity (bytes), a surface capacity, total capacity of the disk.
2. Number of disk cylinders.
3. Average rotational latency.
4. Average transfer time of a block of 4096 bytes.

256, 2048 and 51 200 are examples of valid block sizes?

Exercise 2.2 Consider the disk of the previous exercise with blocks of 1024 bytes to store a file with 100 000 records, each of 100 bytes and stored completely in a block,

Calculate the following parameters.

1. Number of records per block.
2. Number of blocks to store the file.
3. Number of cylinders to store the file per cylinders.
4. Number of 100 bytes records stored in the disk.
5. If the pages are stored on the disk by cylinder, with page 1 on block 1 of track 1, which page is stored on block 1 of track 1 of the next disk surface? What will change if the disk can read/write in parallel by all the array of heads?

6. What is the time to read serially a file with 100 000 records of 100 bytes? What will change if the disk is able to read/write in parallel from all the array of heads (with the data stored in the best way)?
7. Suppose that every reading of a block involves a seek time and a rotational latency time, what is the time to read the file randomly?

Exercise 2.3 Consider a disk with the following characteristics:

- $2^9 = 512$ bytes/sector
- 1000 sectors/track
- 10 000 cylinders
- 5 platters and 10 surfaces
- rotation speed 10 000 rpm
- the seek time is of 1 ms per track plus 1 ms per 1000 cylinders skipped.

Calculate the following parameters.

1. Total capacity of the disk.
2. The average seek time.
3. The average rotational latency.
4. The transfer time of a block ($2^{14} = 16\,384$ bytes).
5. The average time for accessing 10 continuous blocks in one track on the disk.
6. Suppose that half of the data on the disk are accessed much more frequently than another half (*hot* or *cold* data), and you are given the choice to place the data on the disk to reduce the average seek time. Where do you propose to place the hot data, considering each of the following two cases? (Hint: inner-most tracks, outer-most tracks, middle tracks, random tracks, etc). State your assumptions and show your reasoning.
   (a) There are same number of sectors in all tracks (the density of inner tracks is higher than that of the outer tracks).
   (b) The densities of all tracks are the same (there are less sectors in the inner tracks than in the outer tracks).

Exercise 2.4 Give a brief answer to the following questions:

1. Explain how the read of a page is executed by the buffer manager.
2. When the buffer manager writes a page to the disk?
3. What does it mean that a page is pinned in the buffer? Who puts the pins and who takes them off?

Chapter 3

HEAP AND SEQUENTIAL ORGANIZATIONS

This chapter begins the analysis of data structures provided by the *Storage Structures Manager* for storing a collection of records in the permanent memory. We will begin with the most simple ones, which do not use any type of auxiliary structures to facilitate the operations on them: the *heap* and *sequential organizations*. Other solutions will be presented in the following Chapters 4 – 7. With a heap organization the records are not sorted, while with a sequential organization the records are stored in contiguous pages sorted on a key value. We will also present the cost model, which will be also used in following chapters, and we will show how to evaluate the performance of these simple organizations. The chapter also presents the fundamental algorithm for sorting files.

3.1 Storing Collections of Records

In the previous chapter we have presented how the *Buffer Manager* interacts with the *Permanent Memory Manager* to read pages from, and write pages to, the disk.

A database is primarily made of tables of records, each one implemented by the *Storage Structures Manager* as a *file of pages* provided by the *Permanent Memory Manager*.

Pages are assumed to be of a fixed size, for example between 1 KB to 4 KB, and to contain several records. Therefore, above the *Storage Structures Manager*, every access is to records, while below this level the unit of access is a page.

The unit of cost for data access is a page access (read or write), and we assume that the costs of operations in the main memory on the data in a page are negligible compared with the cost of a page access.

The most important type of file is the *heap file*, which stores records in no particular order, and provides a record at a time interface for accessing, inserting and deleting records.

3.1.1 Record Structure

Each record consists of one or more *attributes* (or *fields*) of an elementary type, such as numbers or character strings, and contains several additional bytes, called *record header*, which are not used to store data attributes, but for record management. These

bytes generally contain information on the length of the record, the number of attributes, whether the record has been deleted, and the name of the file to which it belongs. We will assume that records are not larger than a page and that the values of the attributes are stored according to one of the strategies shown in Figure 3.1.

A fixed-length record

Attribute	Position	Value type	Value
Name	1	char(10)	Rossi
StudentCode	2	char(6)	456
City	3	char(2)	MI
BirthYear	4	int(2)	68

Total number of characters = 20
Attribute values are separated:

a) by position	Rossi␣␣␣␣␣456␣␣␣MI68
b) with a separator	Rossi@456@MI@68
c) with an index*	(1, 6, 9, 11) Rossi456MI68
d) with labels**	(6, 1, Rossi)(4, 2, 456)(3, 3, MI)(3, 4, 68)

* The index, placed at the beginning of each record, indicates the beginning of each attribute value.
** Each attribute value starts with a counter that tells how many characters are used to code the position of the attribute and its value.

Figure 3.1: Representation of attribute values

3.1.2 Page Structure

When a record is stored in the database, it is identified internally by a *record identifier* or *tuple identifier* (RID), which is then used in all data structures as a pointer to the record. The exact nature of a RID can vary from one system to another. An obvious solution is to take its address (*Page number, Beginning of record*) (Figure 3.2a). But this solution is not satisfactory because a record that contains variable-length attributes of type varchar are themselves variable-length strings within a page; so updates to data records can cause growth and shrinkage of these byte strings and may thus require the movement of records within a page, or from one page to another. When this happens all the references to the record in other data structures, most notably for indexing purposes, must be updated.

To avoid this problem, another solution is usually adopted: the RID consists of two parts (*Page number, Slot number*), where the slot number is an index into an array stored at the end of the page, called *slot array*. containing the full byte address of a record (Figure 3.2b). All records are stored contiguously, followed by the available free space.

If an updated record moves within its page, the local address in the array only must change, while the RID does not change. If an updated record cannot be stored in the same page because of lack of space, then it will be stored in another page, and the original record will be replaced by a forwarding pointer (another RID) to the new location. Again, the original RID remains unaltered. A record is split into smaller records stored in different pages only if it is larger than a page.

Each page has a *page header* (HDR) that contains administrative information, such as the number of free bytes in the page, the reference at the beginning of the free

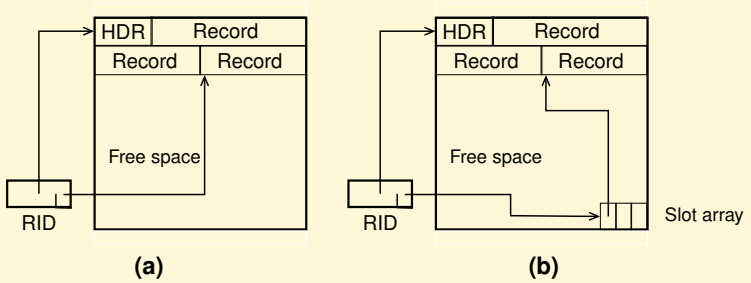


Figure 3.2: Pointers to records

space, and the reference to the next not empty page of the file.

3.1.3 File Pages Management

A collection of records is stored using appropriately the pages of a heap file. To support data updates, the main problem to be addressed is the management of the free space of the pages available, which requires to find a solution to problems related to the following questions:

- Where to store new record?
- How to deal with record updates?
- How to reuse the memory that becomes available after an update or a delete of a record?
- How to compact small fragments of memory in larger units more likely to be reused?

When a record is inserted into a collection, assuming that the records have size smaller than a page, the file manager proceeds as follows:

1. A file page is selected that contains free space for the new record; if the page does not exist, the file is extended with a new page. Let P be the address of the page where the record will be stored.
2. A reference to the beginning of the record is stored in the first free location j of the directory of slots of the page P.
3. The RID (P, j) is assigned to the record.

To implement insertion efficiently, the system uses a table, stored on disk, containing pairs of (*fileName, headerPage*), where the header page is the first page of the file, and the following alternatives are usually considered:

- The heap file pages are organized as two double linked list of pages, those full and those with free space. The two lists are rooted in the header page and contain all the pages in the heap file.
  When the free space of a page is used, or when a free space is created in a full page, the page moves from one list to another.

– In the header is stored a *directory of pages* and each entry contains the pair (page identifier, the amount of free space on the page). If the directory grows and cannot be stored in the header page, it is organized as a linked list. The information about the amount of free space on a page is used to select a page with enough space to store a record to be inserted.

Finally, for reasons of efficiency, the free space existing in different pages is not compacted in one page by moving records. Therefore, it may happen that, due to a lack of available pages, it is not possible to assign a new free page despite the fact that the overall free space in different pages is greater than the total capacity of a page. When this occurs, it is necessary to reorganize the database.

3.2 Cost Model

The most important criteria to evaluate a file organization are the *amount of memory occupied* and the *cost of the basic operations* (record search, insertion, deletion, and modification). The most important operation is the search, because the first step for all operations is to check whether a record exists.

We will also estimate the values of the following parameters, assuming for simplicity that a file R has $N_{\mathsf{rec}}(R)$ records, of equal and constant length L_r, stored in $N_{\mathsf{pag}}(R)$ pages of size D_{pag}:

1. Memory requirements.
2. Operations cost to[1]
   (a) search for a record with a given key value (*equality search*);
   (b) search for records with a key value in a certain range (*range search*);
   (c) insert a new record;
   (d) delete a record;
   (e) update a record.

Due to the characteristics of the permanent memory, the operations cost will be estimated by considering only the operations to read and write a file page, ignoring the cost of operations in temporary memory, which is assumed to be negligible. In current computer the typical times of operations in temporary memory are in fact at least 10 000 times lower than the typical access time to the permanent memory.

For simplicity, the cost of the operations will be expressed in terms of the number of permanent memory accesses, i.e., the number of pages read or written, rather than in terms of execution time. This simplification, however, does not affect the comparison of alternative solutions, which is what matters in selecting the best alternative. Instead, to estimate the cost of operations in terms of execution time, other factors should be considered, such as the way in which a file is stored in permanent memory, the buffer management technique, the implementation techniques of the operating system and the characteristics of the device. When, in some instances, we want to emphasize the magnitude of the execution time, we will make the simplifying assumption that the time to perform an operation is a simple function of the number of access operations:

$$\mathit{ExecutionTime} = \mathit{NoAccesses} \times \mathit{OneAccessAverageTime}$$

1. In estimating the cost of insertion and deletion, for simplicity, we will not consider the cost of updating service information for the management of the file pages.

where *OneAccessAverageTime*, which depends on the type of permanent memory and the size of the file pages, is assumed to be equal to 10 ms.

3.3 Heap Organization

The simplest way to organize data is to store it in file pages in the insertion order rather than sorted by a key value. In this way, the pages allocated to data can be contiguous or can be linked in a list.

The heap organization is the default solution used by DBMSs, and it is adequate for the following cases:

- When the collection of records is small.
- When key search operations are infrequent, and do not require fast answers, or when they are facilitated by the presence of appropriate indexes.

3.3.1 Performance Evaluation

Memory Requirements. Unlike more complex organizations that we will see later, the memory occupied by a heap organization is only that required by the inserted records: $N_{\mathsf{pag}}(R) = N_{\mathsf{rec}}(R) \times L_r / D_{\mathsf{pag}}$.[2]

Equality Search. Assuming that the distribution of the key values is uniform, the average search cost is:

$$C_s = \begin{cases} \left\lceil \frac{N_{\mathsf{pag}}(R)}{2} \right\rceil & \text{if the key exists in the file} \\ N_{\mathsf{pag}}(R) & \text{if the key does not exist in the file} \end{cases}$$

Range Search. The cost is $N_{\mathsf{pag}}(R)$ accesses because all file pages must be read.

Insert. A record is inserted at the end of the file, and the cost is 2.

Delete and Update. The cost is that of a key search plus the cost of writing back a page: $C_s + 1$.

3.4 Sequential Organization

A sequential organization is used for efficient processing of records stored in sequential order, according to the value of a *search-key* k for each record. The disadvantage of this organization is that it is costly to maintain the sequential order when new records are inserted in full pages. For this reason, commercial DBMSs do not usually preserve the sequential order in the case of a page overflow, and offer other more complex organizations, which we will see later, to exploit the benefits of sorted data.

2. Here and hereafter, when we talk about memory requirements we are only considering the number of pages to store the records, always lower than the number of pages in the file, where a portion of the memory is used by service information and the pages may have free space initially left for record insertions.

3.4.1 Performance Evaluation

Memory Requirements. If record insertions are not allowed, the organization requires the same memory as a heap organization. In the case of insertions, additional memory is required to leave space in the pages at the time of loading data.

Equality Search. The cost of a search by a key value is $\lceil N_{\mathsf{pag}}(R)/2 \rceil$, both when the value exists in the file and when the value does not exist. If the data is stored in consecutive pages, then a binary search has the cost $\lceil \lg N_{\mathsf{pag}}(R) \rceil$.

Range Search. A search by the key k in the range $(k_1 \leq k \leq k_2)$, and assuming keys numerical uniformly distributed in the range $(k_{\mathsf{min}}, k_{\mathsf{max}})$, the ratio $s_f = (k_2 - k_1)/(k_{\mathsf{max}} - k_{\mathsf{min}})$, called *selectivity factor*, is an estimate of the fraction of pages occupied by the records which will satisfy the condition, and the cost of the operation is:

$$C_s = \lceil \lg N_{\mathsf{pag}}(R) \rceil + \lceil s_f \times N_{\mathsf{pag}}(R) \rceil - 1$$

where the first term is the cost of the binary search to identify the page containing k_1. The number of pages occupied by the records in the range key is decreased by 1 because the first page has been found with the binary search.

Insert. If the record must be inserted in a page not full, the cost is $C_s + 1$. If all the pages are full, the cost is estimated by assuming that the record must be inserted in the middle of the file, and all subsequent $N_{\mathsf{pag}}(R)/2$ pages must be read and written to move their records as result of a single insertion. The cost is $C_s + N_{\mathsf{pag}}(R) + 1$.

Delete and Update. The cost is $C_s + 1$, if the update does not change the key on which data is sorted.

3.5 Comparison of Costs

Table 3.1 compares costs for heap and sequential organizations in consecutive pages, with C_s as the search cost of a key value present in the file.

Table 3.1: A comparison of heap and sequential organizations

Type	Memory	Equality Search	Range Search	Insert	Delete
Heap	$N_{\mathsf{pag}}(R)$	$\lceil N_{\mathsf{pag}}(R) \,/\, 2 \rceil$	$N_{\mathsf{pag}}(R)$	2	$C_s + 1$
Sequential	$N_{\mathsf{pag}}(R)$	$\lceil \lg N_{\mathsf{pag}}(R) \rceil$	$C_s - 1 + \lceil s_f \times N_{\mathsf{pag}}(R) \rceil$	$C_s + 1 + N_{\mathsf{pag}}(R)$	$C_s + 1$

A heap organization has good performance for insertion operations, but it has bad performances for range queries and for equality search, in particular for the search of a key value not in the file.

A sequential organization has good performance for search operations, but it has bad performance for insertion operations.

As we will see with other organizations, the important thing to remember is that any solution is better than another under certain conditions and therefore the choice of the most appropriate depends on how data is used, and on the costs to minimize.

3.6 External Sorting

A frequent operation in a database system is sorting a collection of records for different reasons:

- To load them in a physical organization.
- To delete the duplicates from the result of a query.
- To perform a join operation with the *merge-join* algorithm, an **ORDER BY**, a **GROUP BY**, etc.

Sorting a file is a different process from sorting a table in the temporary memory, because the number of record is usually too large to be completely stored in the memory available. For this reason, the classic algorithms for sorting tables are not applicable to this problem and an *external sorting* algorithm is used. The evaluation of this algorithm is also different from those used for the temporary memory. Instead of the number of comparisons, only the number I/O operations are considered.

Let $N_{\mathsf{pag}}(R)$ be the number of file pages, and B the buffer pages available. The classical external sorting algorithm, called *merge-sort*, consists of two phases:

1. *The sort phase.* B file pages are read into the buffer, sorted, and written to the disk. This creates $n = \lceil N_{\mathsf{pag}}(R)/B \rceil$ sorted subset of records, called *runs*, stored in separate auxiliary files, numbered from 1 to n. The runs have all the same number of pages, B, except the last.
2. *The merge phase* consists of multiple merge passes. In each merge pass, $Z = B-1$ runs are merged using the remaining buffer page for output. At the end of a merge pass, the number of runs becomes $n = \lceil n/Z \rceil$. A merge pass is repeated until $n > 1$.

The final auxiliary file contains the sorted data.

The parameter Z is called the *merge order*, and $Z + 1$ buffer pages are needed to proceed with a *Z-Merge*.

Example 3.1

Let us show how to sort the file A_0 containing 12 pages, with file and buffer pages capacity of 2 records, $B = 3$ and 2-merge passes (Figure 3.3):

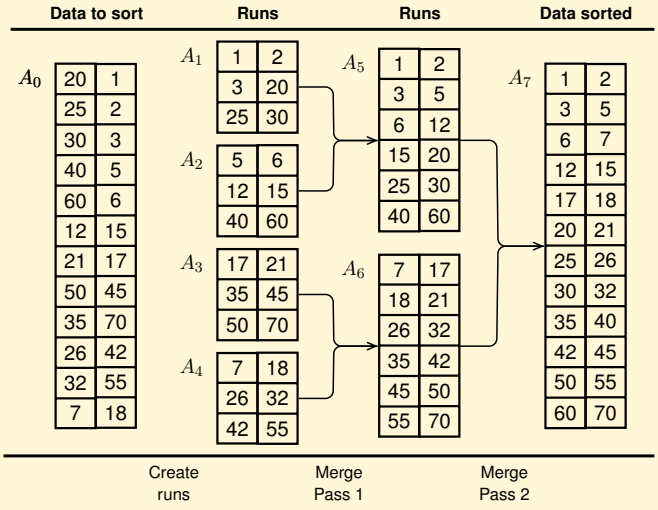


Figure 3.3: External sorting using merge-sort with $Z = 2$

- The initial sort phase creates the runs A_1, A_2, A_3 and A_4.
- The first merge pass creates the runs A_5 and A_6 by merging A_1, A_2 and A_3, A_4.
- The second merge pass creates the sorted data A_7 by merging A_5, A_6.

3.6.1 Performance Evaluation

Suppose that B buffer pages are available. The external sorting cost is evaluated in term of *number of passes*, i.e., the number of times that the $N_{\mathsf{pag}}(R)$ pages are read in and written out. The number of passes is the initial one to produce the sorted runs, plus the number of the *merge passes*. Therefore, the total cost of the merge-sort algorithm in terms of number of pages read and written is:

$$C_{\mathsf{sort}}(R) = \mathrm{SortPhaseCost} + \mathrm{MergePhaseCost}$$

$$C_{\mathsf{sort}}(R) = 2 \times N_{\mathsf{pag}}(R) + 2 \times N_{\mathsf{pag}}(R) \times \mathrm{NoMergePasses}$$

If $N_{\mathsf{pag}}(R) \leq B \times (B-1)$, the data can be sorted with a single merge pass, as it will be assumed in some cases, and so with two passes the cost becomes

$$C_{\mathsf{sort}}(R) = 4 \times N_{\mathsf{pag}}(R)$$

In general, the number of passes required in the merge phase is a function of the number of file pages $N_{\mathsf{pag}}(R)$, the number S of initial runs, and the merge order $Z = B - 1$. S depends on $N_{\mathsf{pag}}(R)$ and the number of buffer pages available to create the initial runs ($S = \lceil N_{\mathsf{pag}}(R)/B \rceil$). After each merge pass, the maximum length of the runs increases of a factor Z, and so their number becomes

$$\lceil S/Z \rceil, \lceil S/Z^2 \rceil, \lceil S/Z^3 \rceil, \ldots$$

The algorithm terminates when a single run is generated, that is for the minimum value of k such that $Z^k \geq S$, quantity that for a certain value of $N_{\mathsf{pag}}(R)$, decreases with the increase of Z. Therefore the number of passes required in the merge phase is: $k = \lceil \log_Z S \rceil$, and the total cost of the merge-sort is:

$$C_{\mathsf{sort}}(R) = 2 \times N_{\mathsf{pag}}(R) + 2 \times N_{\mathsf{pag}}(R) \times \lceil \log_Z S \rceil = 2 \times N_{\mathsf{pag}}(R) \times (1 + \lceil \log_Z S \rceil)$$

Table 3.2 shows some values of the number of merge passes, cost and time to sort a file — expressed in minutes and calculated assuming that the reading or writing a page costs 10 ms — depending on the number of pages, the size of the buffer and the merge order. The quantities shown justify the improvements used in DBMS to reduce the external sorting costs.

For example, a technique has been studied to revise the sort phase in order to increase the length of the runs and thereby reduce the number of number of merge passes. The initial runs become long on the average $2B$, with the method *replacement sort* [Knuth, 1973], and $B \times e$, where $e = 2.718\ldots$, with the method *natural selection* [Frazer and Wong, 1972].

Table 3.2: Total costs of the merge-sort

	B = 3, Z = 2			B = 257, Z = 256		
$N_{\text{pag}}(R)$	Merge Passes	Cost	Time (m)	Merge Passes	Cost	Time (m)
100	6	1400	0.23	0	200	0.03
1000	9	20 000	3.33	1	4000	0.67
10 000	12	260 000	43.33	1	40 000	6.67
100 000	16	3 400 000	566.67	2	600 000	100.00
1 000 000	19	40 000 000	6666.67	2	6 000 000	1000.00

Curiosity. A typical DBMS sort 1 M of 100 byte records in 15 minutes. The best result in 2001 was 3.5 seconds obtained with a SGI system, 12 CPUs, 96 disks and 2 G of RAM. Another criterion sometimes used to evaluate an external sorting algorithm is how much data it sorts in one minute (minute sort). Every year the *Sort Benchmark Home Page* shows the results of the best solution for different evaluation criteria. In 2009 the data sorted in one minute were 500 GB and 1353 GB in 2011, a result obtained by TritonSort, with a system with "52 nodes × (2 Quadcore processors, 24 GB memory, 16×500 GB disks), Cisco Nexus 5096 switch."

Table Organizations in INGRES. As with any DBMS, a table R has initially a heap organization, then, once data has been loaded, it is possible to reorganize the table as sorted, with the command:

MODIFY R **TO HEAPSORT**
ON Attr [**ASC** | **DESC**]{, Attr [**ASC** | **DESC**]}

The command also includes the possibility to leave some free space in the data pages (**FILLFACTOR**). Record insertions in the table that produce overflows do not preserve the sorted order.

3.7 Summary

1. The *Storage Structures Manager* primarily implements a database table of records as a *heap file of pages*, and each page contains one or more records. A record is identified by a RID, the pair (page identifier, position in the page), and it can have fixed-length fields (easier to manage) or variable length fields (more complex to manage, especially in the presence of fields greater than the size of a page).
   In the case of updates, there is the problem of managing the free space of pages available in a file, which requires appropriate solutions to avoid wasting memory and performance degradation.
2. Data organization is a way to store it in a file, which is evaluated in terms of memory occupied and number of file pages to read or write to perform operations.
3. The heap and sequential organizations are used when the number of records is small or the main interest is in minimizing the storage requirement. Otherwise,

other organizations are used, as will be seen in the next chapters.

4. The heap organization is the easiest way to store data in files, but is not suitable for searching a small number of records.
5. The sequential organization stores the data sorted on the value of a search key and has better performance than the heap one for search operations, but the insertion of records in general does not preserve the sorted order.
6. Sorting records on the value of an attribute is the typical operation on a file and is often used in the operations of other organizations. The merge-sort is the most widely-used external sorting algorithm and its performance depends on the number of buffer pages available.

Bibliographic Notes

Heap and sequential organizations are presented in every book cited in the bibliographic notes of Chapter 1. External sorting is described in [Knuth, 1973], [Tharp, 1988].

Exercises

Exercise 3.1 Explain what is meant by *file reorganization* and give an example of file organization that requires it, specifying which operations motivate it.

Exercise 3.2 Discuss the advantages and disadvantages of records with fixed fields vs variable fields, and of records with fixed length vs variable length.

Exercise 3.3 Let $R(K, A, B, \mathit{other})$ a relation with $N_{\mathsf{rec}}(R) = 100\,000$, a key K with integer values in the range (1, 100 000), and the attribute A with integer values uniformly distributed in the range (1, 1000). The size of a record of R is $L_r = 100$ bytes. Suppose R stored with heap organization, with data *unsorted* both respect to K and A, in pages with size $D_{\mathsf{pag}} = 1024$ bytes.

The cost estimate C of executing a query is the number of pages read from or written to the permanent memory to produce the result.

Estimate the cost of the following SQL queries, and consider for each of them the cases that *Attribute* is K or A, and assume that there are always records that satisfy the condition.

1.
```
SELECT   *
FROM     R
WHERE    Attribute = 50;
```

2.
```
SELECT   *
FROM     R
WHERE    Attribute BETWEEN 50 AND 100;
```

3.
```
SELECT     Attribute
FROM       R
WHERE      Attribute = 50
ORDER BY   Attribute;
```

4.
```
SELECT     *
FROM       R
WHERE      Attribute BETWEEN 50 AND 100
ORDER BY   Attribute;
```

5. **SELECT** Attribute
   FROM R
   WHERE Attribute **BETWEEN** 50 **AND** 100
   ORDER BY Attribute;

6. **INSERT INTO** R **VALUES** (...);
7. **DELETE**
   FROM R
   WHERE Attribute = 50;

8. **UPDATE** R
   SET A = 75
   WHERE K **BETWEEN** 50 **AND** 100;

Exercise 3.4 Assuming that a page access requires 10 ms, estimate the execution time of the SQL queries of the previous exercise in the case of a sequential organization with records *sorted* on the key K values.

Exercise 3.5 Consider a file R with 10 000 pages to sort using 3 pages in the buffer, and to write the sorted file to the disk.

1. How many runs are produced in the first pass?
2. How many 2-way merge phases are needed to sort the file?
3. How much time does it take to sort the file if a page access requires 10 ms?
4. How many buffer page B are needed to sort the file with one merge phase?

Exercise 3.6 Consider a file R with $N_{\mathsf{rec}}(R) = 10\,000$ records of 100 bytes stored in pages with size 1 K. Assume that there are $B = 5$ buffer pages to sort the file, and to write the sorted file to the disk.

1. How many runs are produced in the first pass, and how long will each run be?
2. How many passes are needed to sort the file completely?
3. Which is the cost of sorting the file?
4. What is the number of records $N_{\mathsf{rec}}(R)$ of the largest file that can be sorted in just two passes?

Chapter 4

HASHING ORGANIZATIONS

After having seen the simple heap and sequential organizations, in this chapter, and the next, we will consider more complex table organizations based on a key, allowing the search for a record with a given key value with as few accesses as possible to the permanent memory. The chapter is dedicated to the study of a procedural method, also known as *hashing*, proposed since the early fifties to manage data sets in temporary memory. The next chapter will be dedicated instead to organizations based on trees.

4.1 Table Organizations Based on a Key

The goal of a table organization based on a key is to allow the retrieval of a record with a specified key value in as few accesses as possible, 1 being the optimum. To this end, a mapping from the set of keys to the set of records is defined. The mapping can be implemented with a *primary organization* or with a *secondary organization*, defined as follows.

Definition 4.1 *Primary Organization*

A table organization is said to be *primary* if it determines the way the records are physically stored, and therefore how the records can be retrieved; otherwise it is said to be a *secondary organization*.

In the case of a *primary organization* the mapping from a key to the record can be implemented as a *function*, by means of either a *hashing* technique or a tree structure.

In the first case a hash function h is used that maps the key value k to the value $h(k)$. The value $h(k)$ is used as the address of the page in which the record is stored. In the second case a tree structure is used and the record is stored in a leaf node.

Definition 4.2 *Static or Dynamic Primary Organization*

A primary organization is *static* if once created for a known table size, the performance degrades as the table grows because overflow pages must be added, and a reorganization must be performed.
A primary organization is *dynamic* if once created for a known table size, it gradually evolves as records are added or deleted, thus preserving efficiency without the need for reorganization.

In the case of a *secondary organization* the mapping from a key to the record is implemented with the *tabular method*, listing all inputs and outputs, commonly known as an *index*. A *secondary organization* helps answer queries but does not affect the location of the data.

An index in this context has a role similar to that of a book. The pages of a book are ordered, and to find information about a particular subject, the index in the back of the book is used. In a similar way, in the case of a set of records, to find a record with a given key, first the index is probed in order to get the record RID, and then the record is retrieved by means of the RID.

Definition 4.3 *Secondary Organization*

An *index* I on a key K of a set of records R is a sorted table $I(K, RID)$ on K, with $N_{\mathsf{rec}}(I) = N_{\mathsf{rec}}(R)$. An element of the index is a pair (k_i, r_i), where k_i is a key value for a record, and r_i is a reference (RID) to the corresponding record. The records of R are stored with an independent organization.

An index is stored in permanent memory using a primary organization.

In the literature the terminology is not very uniform, and the terms "primary" and "secondary" are usually used to distinguish two types of indexes: "primary" for indexes on the primary key, i.e. "organizations for primary key", and "secondary" is used for indexes on other attributes, i.e. "organizations for secondary key".

In the rest of the chapter we analyze the static and dynamic solutions for the procedural approach, while the tree based approach and indexes will be considered in the next chapter.

4.2 Static Hashing Organization

This is the oldest and simplest method for a primary table organization based on a key. Since here we are only interested in the record keys, we will talk about the *storage and retrieval of keys* rather than the *storage and retrieval of records with a given key value*. Moreover, we assume that records have the same and fixed size, and that the key k has a type *integer*. The N records of a table R are stored in an area, called *primary area*, divided into M *buckets* that may consist of one or several pages. We will assume that the buckets consist of one page with a capacity of c records, and so the primary area is a set of M pages numbered from 0 to $M - 1$.

A record is inserted in a page whose address is obtained by applying a hashing function H to the record key value (Figure 4.1). The ratio $d = N/(M \times c)$ is called the primary area *loading factor*.

Records hashed to the same page are stored in order of insertion. When a new record should be inserted in a page already full, an *overflow* is said to have occurred, and a strategy is required to store the record elsewhere.

The design of a static hashing organization requires the specification of the following parameters:

- The hashing function.
- The overflow management technique.
- The loading factor.
- The page capacity.

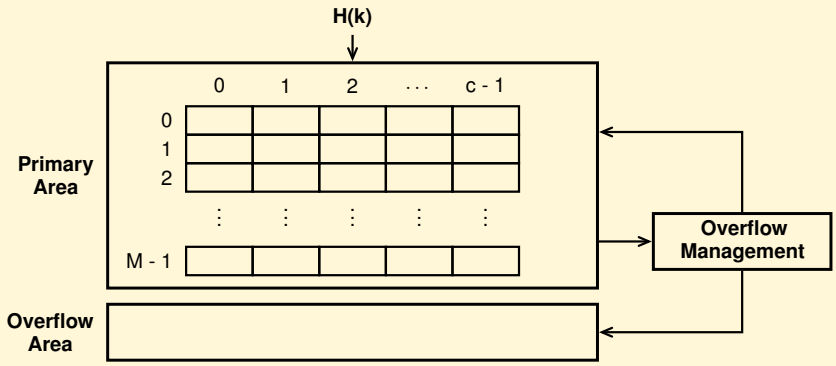


Figure 4.1: Static hashing organization

The Hashing Function. A good hashing function must randomize the assignment of keys over the entire address space. A universally good hashing function does not exist; some experimental results on the performances of different hashing functions have shown that the following simple function works just as well as more complicated ones:

$$H(k) = k \bmod M$$

with M a prime number.

The Overflow Management Technique. In addition to a careful choice of the hashing function, the method used to handle overflows is also very important. The most common techniques used in permanent memories are *open overflow* (or *open addressing*) and *chained overflow* (or *closed addressing*).

Open overflow performs a primary area linear search to find the first available page to insert the overflow record. When the last page has been searched, the process starts back at the first one.

Chained overflow inserts the overflow record in another page of a separate overflow area which is pointed to from the home page. Additional overflow records from the same page are chained through the overflow area.

The performance of this organization will depend on the number of overflows, and this will vary according to the loading factor and the page capacity.

The Loading Factor. In general, lower loading factors and higher page capacities give better performances, but occupy more memory. For a low loading factor (< 0.7) the retrieval requires just 1 access on average. For high loading factors (> 0.8), open addressing deteriorates rapidly, while the chained overflow still performs quite well.

The Page Capacity. The pages capacity c is a very important factor for the performance of a hash organization. Suppose we need 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 = 75%, but the performances are very different. In fact, in the first case the overflows are the 29.6%, and in the second case, while the collisions increase because M is halved, the overflows become the 18.7%, with a reduction of 37%.

Since overflows significantly degrade the performance of hashing organizations in the permanent memory, we may further increase the size of the pages. For example, keeping the load factor d = 0.75 = 75%, but choosing $c = 10$, the performances improve because the overflows are reduced to 4%.

4.2.1 Performance Evaluation

A study of the behavior of the a hash organizations taking into account the characteristics of permanent memories, and in particular the size of the pages, is reported in [Severance and Duhne, 1976]. From this study follows that for page capacity less than 10, it is preferable to give up hash organizations. Since the page size D_{pag} is fixed by the permanent memory manager, their capacity depends on the average length of the record L_r, and therefore this organization is used if $L_r < D_{\mathsf{pag}}/10$.

A static hashing organization has excellent performance as long as there are no overflows to manage: a record retrieval requires 1 page access. Overflows quickly degrade performance and so a reorganization must be performed of the hash structure, creating a new primary area with more space, choosing a new hashing function, and reloading the data.

To reduce the cost of data loading, and to improve then performance, the operation proceeds in two stages: first it stores in an auxiliary file T the pairs (*Record* with key k, $H(k)$), which is sorted on the addresses $H(k)$ generated; then the records in T are stored in the hash structure, by reading and writing each page only once.

The main drawback of a static hashing organization is that it does not support range queries, i.e. the retrieval of all records with a key value which lies within a range of values, such as "find all keys greater than k_1 but less than k_2".

4.3 Dynamic Hashing Organizations

Several dynamic hashing organizations have been proposed to avoid the reorganization which is necessary in static hashing organizations. The proposals can be classified into two categories: those that make use of both a primary area for data pages, and an auxiliary data structure (a kind of index), whose size changes with the primary area size, and those in which only the size of the primary area changes dynamically. In both cases, the hashing function will automatically change when the structure changes dimension, in order to allow the retrieval of a key in about one access.

In the following we present the *virtual hashing* and the *extendible hash* as examples of organizations with auxiliary data structures, as well as the *linear hashing* as example of organization that uses instead only the primary area that is expanded linearly. Reference to other more elaborate methods are given in the bibliographic notes.

4.3.1 Virtual Hashing

Litwin proposed a new type of hashing called *Virtual Hashing* that works as follows [Litwin, 1978]:

1. The data area contains initially M contiguous pages with a capacity of c records. A page is identified by its address, a number between 0 and $M - 1$. M can also be a small number. For example, the author started his experiments with $M = 7$.
2. A bit vector $\mathcal{B}$ is used to indicate with a 1 which page of the data area contains at least a record.

3. Initially a hashing function H_0 is used in order to map each key value k to an address $m = H_0(k)$, between 0 and $M - 1$, where the record with the key k should be stored. If an overflow is generated then
   (a) the data area is doubled, maintaining contiguous the pages;
   (b) the hashing function H_0 is replaced by the new hashing function H_1 that produces page addresses between 0 and $2M - 1$;
   (c) the hashing function H_1 is applied to k and all the records of the original overflowing page m to distribute the records between m and a new page m' in the new half of the table. The records of pages different from m are not considered because they must not change their pages.

This method requires the use of a hashing functions series H_0, H_1, H_2, ..., H_r; in general, H_r produces a page address between 0 and $2^r M - 1$.

The index of the hashing function H is the number of times that the data area has been doubled.

The hashing functions must satisfy the following property for every key value k:

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

In other words, the application of H_{j+1} to a key k gives $H_j(k) = m$ (the new page address generated is the original one), or $m + 2^j M$ (the corresponding page address in the new half of the doubled data area).

The function suggested by Litwin is

$$H_r(k) = k \bmod (2^r \times M)$$

Let k be the key of a record to be retrieved, and r the number of doubling of the data area. The address of the page that contains the desired record, if it exists, is computed with the recursive function in Figure 4.2.

```
function PageSearch(r, k: integer): integer
    begin
        if r < 0
            then write "The key does not exist";
            else if B(H_r(k)) = 1
                then PageSearch := H_r(k)
                else PageSearch := PageSearch(r - 1, k)
    end;
```

Figure 4.2: Search operation

The use of the vector $\mathcal{B}$, stored in the main memory, allows to establish with a single access if the record with a specified key is present in a non-empty page.

Example 4.1
Suppose that we insert the key 3820 into the initial hash structure of Figure 4.3a, with $M = 7$ and $c = 3$. Applying the function $H_r(k) = H_0(3820)$ we get $m = 5$: since $\mathcal{B}(5) = 1$, the new key must be stored in the page with address 5.

This page is full, therefore it is necessary to double the data area. The records in the page with address 5 are distributed between this page and a new page with address $m' = 12$, by means of the function H_1. The vector $\mathcal{B}$ is doubled too and its values are updated, as shown in Figure 4.3b.

From now on the transformation function H_1 will be used to access the hash structure. Let us now assume that we must insert the key 3343, with $H_1(3343) = 11$. The zero bit in the vector $\mathcal{B}$ indicates, however, that this page has not yet been used, and therefore the key transformation $H_0(3343) = 4$ is applied. Being the page 4 full, it is still necessary to use the procedure for the resolution of the overflow, which in this case does not require a doubling of the primary area, because the page 11 already exists in the structure. It is sufficient to set to 1 the bit in the vector $\mathcal{B}$ and then transform with H_1 all the keys of the original page 4 (7830, 1075, 6647) plus the new one 3343, distributing them between the pages 4 and 11.

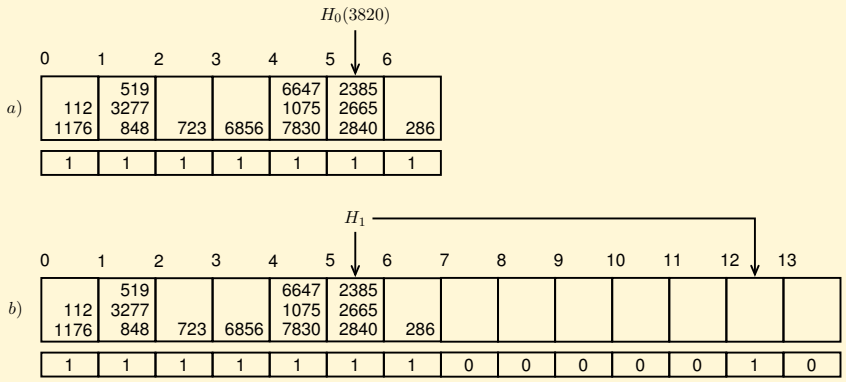


Figure 4.3: Virtual Hashing: *a*) initial structure and *b*) the structure after the first overflow from page 5

Memory Requirements. The memory occupied is that required for the data area and for the binary vector $\mathcal{B}$. The memory is not, however, well used, because of the frequent doublings of the data area. The disadvantage of this solution lies in the fact that the loading factor of the data area fluctuates between 0.45 and 0.9 and is on average 0.67. The advantage of this solution is that each record is retrieved with one access, if the vector $\mathcal{B}$ is stored in the main memory.

Observation. This solution, as it has been presented, is of mainly historical interest, having as main limit the doubling of the data with contiguous pages. However, his exposure is useful for illustrating another solution, the *extendible hashing*, which overcomes the limitations of *virtual hashing* with a different use of the vector $\mathcal{B}$.

4.3.2 Extendible Hashing *

Unlike virtual hashing, *Extendible hashing* uses a set of data pages, and an auxiliary data structure $\mathcal{B}$, called *directory*, an array of pointers to data pages.

Let r be a record with key k. The value produced by the hash function $H(k)$ is a binary value of b-bit, called *hash key* (a typical value for b is 32), which, however, is not used to address a fixed set of pages as in virtual hashing. Instead, pages are

allocated on demand as records are inserted in the file, considering only the initial p bits of b, which are used as an offset into $\mathcal{B}$. The value of p grows and shrinks with the number of pages used by data. The number of $\mathcal{B}$ entries is always a power of 2, that is 2^p. We call p the *directory level*. The value of p is stored in $\mathcal{B}$.

A $\mathcal{B}$ entry is a pointer to a data page containing records with the same first p' bit of their hash key, with $0 \leq p' \leq p$. We call p' the *data page level*, and its value is stored in the data page. The value of p' depends on the evolution of the data pages as result of overflows, as explained below, and it is used to determine membership in a data page.

Let us assume that initially the hash structure is empty, $p = 0$, and consists of a directory with one entry containing a pointer to an empty page of size c (Figure 4.4a).

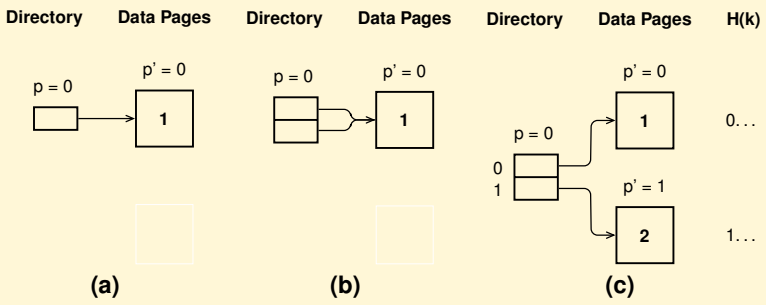


Figure 4.4: Extendible Hashing structure: a) initial; b) doubling of directory after an overflow from page 1, and c) after page 1 split

The first c records inserted are stored in the page. In general, when we attempt to insert the next record into a full page n (in the example $n = 1$), there are two possibilities, depending on the value of p'.

1. If $p' = p$ the operation concerns the data page and the directory $\mathcal{B}$.
   (a) $\mathcal{B}$ is doubled and its level p takes the value $p + 1$ (Figure 4.4b). Let w be the bits of the previous value of p, indexing one of the entries of the previous $\mathcal{B}$. The entries of the doubled $\mathcal{B}$ indexed by both the binary values $w0$ and $w1$ each contains a pointer to the same data page that the w entry used to point to.
   (b) Since now $p' < p$, the next case applies to deal with the data page with the overflow.
2. If $p' < p$ the operation concerns the data page only.
   (a) The data page n is split in two (n and n'), and their levels p' take the value $p' + 1$ (Figure 4.4c).
   (b) The records of page n are distributed over n and n', based on the value of the first $(p' + 1)$ high-order bit of their hash keys: records whose key has 0 in the $(p' + 1)$th bit stay in the old page n, and those with 1 go in the new page n'.
   (c) The pointers in the $\mathcal{B}$ entries are updated so that those that formerly pointed to n now point either to n or to n', depending on their $(p'+1)$th bit (Figure 4.4c).

In general, if $\mathcal{B}$ has several entries with pointers to the same page, then the entries are contiguous and in number of 2^q, for some integer q. This means that a page pointed by 2^q contiguous entries contains all and only those records with hash keys with

the same prefix long exactly $p' = p - q$ bit. For example, Figure 4.5a shows the $\mathcal{B}$ structure after the split of the data page 2 in Figure 4.4c: the entries $\mathcal{B}(00)$ and $\mathcal{B}(01)$ point to the same page that contains records with hash keys prefix 0 $(p' = 1)$. Figure 4.5b shows $\mathcal{B}$ after the split of page 3.

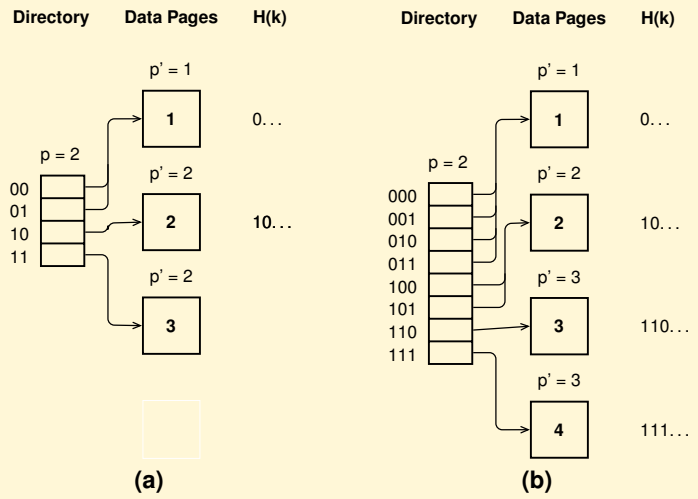


Figure 4.5: Extendible hash structure: a) after an overflow from page 2 and b) after an overflow from page 3

If after a delete operation, the contents of two pages 'close' can be stored in only one of them, then the two pages are merged, the new data page levels is $p' - 1$ and $\mathcal{B}$ is updated. Two pages are said to be 'close' if they are identified by the same value p' and their records differ for the p'th value of their hash key. If the only two pages 'close' with $(p' = p)$ are merged, $\mathcal{B}$ is halved. For example, if after a delete operation in page 4 (Figure 4.5b), the two 'close' pages 3 and 4 are merged, and the structure will become again that shown in Figure 4.5a.

The advantage of this method is that performance does not degrade as the file grows, and the directory $\mathcal{B}$ keeps the space overhead low. The retrieval of a record involves an additional level of indirection since we must first access $\mathcal{B}$, but this extra access has only a minor impact on performance, since most of the directory will be kept in main memory and thus the number of page accesses is usually only slightly higher than one.

4.3.3 Linear Hashing

The basic idea of this method is again to increase the number of data pages as soon as a page overflows; however, the page which is split is not the one that flows over, but the page pointed by the current pointer p, initialized to the first page $(p = 0)$ and incremented by 1 each time a page is split [Litwin, 1980]. Overflow management is necessary because the pages that flow over are not generally split: a page will only be split when the current pointer reaches its address.

Initially, M pages are allocated and the hash function is $H_0(k) = k \bmod M$. When there is an overflow from a page with address $m \geq p$, an overflow chain is maintained for the page m, but a new page is added. The records in page p, and possible overflows from this page, are distributed between the page p and the new page using the hash

function $H_1(k) = k \bmod 2M$, which generates addresses between 0 and $(2M - 1)$. Figure 4.6 shows an example with $M = 7, c = 3$, where two overflows from pages 2 and 3 have already occurred, and so pages 0 and 1 have been split. The arrival of key 3820 generates an overflow from page 5 and page 2 is split.

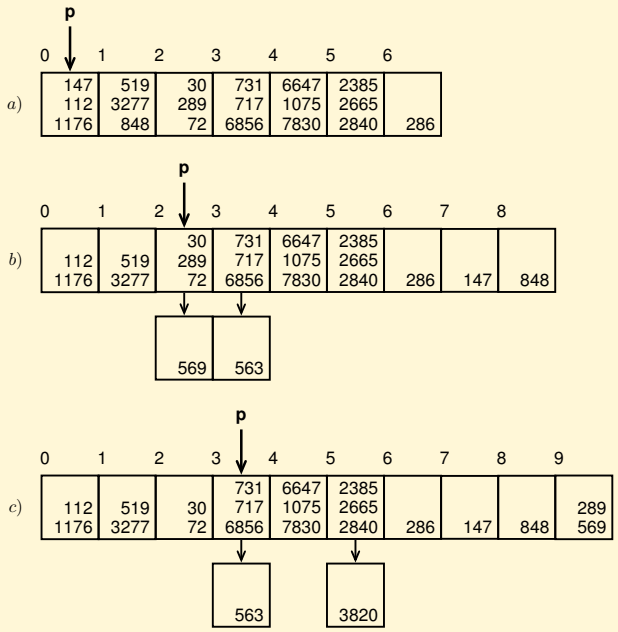


Figure 4.6: Expansion steps in linear hashing

To retrieve a record with key value k, the page address is computed with the following function:

PageAddress(p: int, k: int): int :=
    if $H_0(k) < p$ **then** $H_1(k)$ **else** $H_0(k)$

After M overflows, the memory will be $2M$ pages. Pointer p is set to 0, function H_0 is replaced by H_1, H_1 by $H_2 = k \bmod 2^2M$, and the process continues. In general, after r doublings of the data, the function $H_r(k) = k \bmod 2^rM$ will be used.

Linear hashing has performances similar to those of extendible hashing.

Another interesting example of this kind of solution is the *Spiral Hashing*, proposed by Martin [Martin, 1979]. The name is derived from considering the memory space organized as a spiral rather than as a line. Like linear hashing, spiral hashing requires no index, but it has both a better performance and storage utilization because of the following interesting properties, in contrast to linear hashing: the hashing function distributes the records unevenly over the data pages. The load is high at the beginning of the active address space and tapers off towards the end. Therefore the page that is split is the one that is most likely to overflow.

4.4 Summary

1. Hashing organizations are those that give the best result when a record must be retrieved using the record key: a hash function applied to the key provides the

page address that could hold the record with the specified key. The average search cost is a good approximation of what is obtainable with a perfect transformation, in particular if use is made of a dynamic organization. Another great advantage of these techniques is the simplicity of the implementation.
A static organization requires a reorganization when the amount of data increases in order to avoid performance degradation, whereas this phenomenon is not present in a dynamic organization, which adapts to the number of records present.

2. The most important parameter in the design of a static hashing organization is the load factor of the primary area, which influences both the cost of operations and the memory requirement. With regard to the overflow management techniques, the best performance is obtained with the management of the overflow with separate chaining, which enables a search with an average number of accesses near unity and a memory occupation only a few percentage points (5 – 10%) more than strictly necessary.
3. Another feature of hashing organizations is that the average cost of the operations is generally low, but in the worst case can be considerably higher, and it is not always possible to estimate the cost precisely. A borderline case is given by the organization with static overflow handled by open addressing: in the worst case, the search for a record is equivalent to the cost of a scan of the entire primary area. Another worst case is when in a dynamic organization, after a page split, all the records are assigned again to the same page.
4. Hashing organizations do not support range queries, i.e. the retrieval of all records with a key value which lies within a range of values.

Bibliographic Notes

Static hashing organizations are presented in every book cited in the bibliographic notes of Chapter 1.

The first proposal of a dynamic hashing organization was made for the temporary memory [Knott, 1975], and then the approach was extended to the permanent memory by Litwin [Litwin, 1978, 1980] and Scholl [Scholl, 1981] Mullin [Mullin, 1985]. Larson [Larson, 1982] and Ramamohanarao [Ramamohanarao, 1984] have instead improved linear hashing.

In [Cesarini and Soda, 1991] an interesting dynamic hashing organizations is presented by combining a variant of the spiral hashing with a particular management technique of overflows.

A review of dynamic hashing organizations is presented in [Enbody and Du, 1988].

Exercises

Exercise 4.1 The CREATE TABLE statement of a relational system creates a heap-organized table by default, but the DBA can use the following command to transform a heap organization into a hash primary organization:

MODIFY Table **TO HASH ON** Attribute;

The manual contains the following warning: "Do not modify a table's structure from its default heap structure to a keyed (i.e. hash) structure until the table contains most, if not all, of its data, ..., (otherwise) query processing performance degrade upon adding extra data". Explain what determines the performance degradation.

Exercise 4.2 Let $R(K, A, B, \mathit{other})$ be a relation with an integer primary key K. In this book it has been shown how the relation is stored with a *primary static hashing*

organization. Explain how to modify the operations when the static hashing organization is made using an integer non-key attribute A.

Exercise 4.3 Let $R(K, A, B, \textit{other})$ be a relation with $N_{\mathsf{rec}}(R) = 100\,000$, $L_r = 100$ bytes, and a key K with integer values in the range (1, 100 000). Assume the relation stored with a *primary static hashing organization* using pages with size $D_{\mathsf{pag}} = 1024$ bytes and a loading factor $d = 0,80$.

Estimate the cost of the following SQL queries, assuming that there are always records that satisfy the condition.

1. **SELECT** *
   FROM R;

2. **SELECT** *
   FROM R
   WHERE K = 50;

3. **SELECT** *
   FROM R
   WHERE K **BETWEEN** 50 **AND** 100;

4. **SELECT** *
   FROM R
   WHERE K **BETWEEN** 50 **AND** 100
   ORDER BY K;

Exercise 4.4 Let $R(K, A, B, \textit{other})$ be a relation with $N_{\mathsf{rec}}(R) = 100\,000$, $L_r = 100$ bytes, and a key K with integer values in the range (1, 100 000), and the attribute A with integer values uniformly distributed in the range (1, 1000) and $L_A = 4$. Assume the relation stored using pages with size $D_{\mathsf{pag}} = 1024$ bytes, and the following queries must be executed:

1. Find all R records.
2. Find all R records such that $A = 50$.
3. Find all R records such that $K \geq 50$ and $K < 100$.

Which of the following organizations is preferable to perform each operation?

1. A serial organization.
2. A static hashing organization.

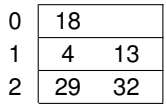
Exercise 4.5 Let $R(K, A, B, \textit{other})$ be a relation with $N_{\mathsf{rec}}(R) = 100\,000$, $L_r = 100$ bytes, and a key K with integer values in the range (1, 100 000), and $L_K = 4$. Assume the relation stored using pages with size $D_{\mathsf{pag}} = 1024$ bytes, and the following queries must be executed:

1. Find all R records.
2. Find all R records such that $K = 50$.
3. Find all R records such that $K \geq 50$ and $K < 100$.
4. Find all R records such that $K \geq 50$ and $K < 55$.
5. Find all K values.

Which of the following organizations is preferable to perform each operation?

1. A sequential organization.
2. A static hashing organization.
3. An unclustered hash index I.

Exercise 4.6 Consider a *linear hashing* organization, with $M = 3$ and each page holding 2 data entries. The following figure shows how the keys $\{4, 18, 13, 29, 32\}$ are stored.



Show how the structure changes by inserting then the following keys in the order (9, 22, 44, 35).

Chapter 5

DYNAMIC TREE-STRUCTURE ORGANIZATIONS

In the previous chapter we saw primary organizations with the procedural approach, highlighting the advantages and disadvantages, in particular, the inability to search by key ranges, which in many cases precludes the use of the techniques illustrated. For this reason research has been directed from the beginning of the sixties towards an alternative approach based on tree structures. The tree structures presented in this chapter are among the most important ones in computer science because of their versatility and performance characteristics, and are commonly used by both operating systems and commercial DBMSs.

5.1 Storing Trees in the Permanent Memory

Balanced binary trees, such as the AVL trees that are used for main memory data structures, are not well suited for permanent memory. First of all, with a binary tree, the search for an element of a set may involve a high number of accesses to the permanent memory. For example, to search through a set with one million elements, organized with a binary tree, without any attention to the way in which the nodes are stored in pages of the permanent memory, requires an average of $\lg 1\,000\,000 = 20$ accesses. The second point is that for large and volatile sets an algorithm to keep a binary tree balanced can be very costly.

A solution to the first problem can be to store the nodes of a binary tree into the pages of the permanent memory properly. Figure 5.1 shows an example with pages that may contain seven nodes. From every page, eight different pages can be accessed, so that the tree behaves as a multiway search tree.

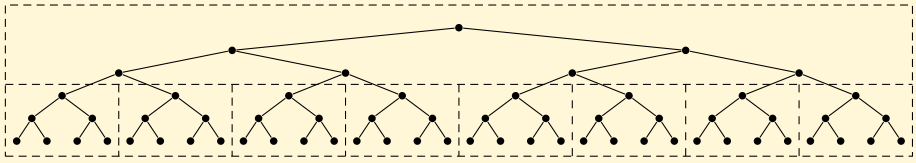


Figure 5.1: Paged binary tree

This layout of the tree reduces the search cost. For example, if the page capacity is of 100 nodes, for the index of the previous example the cost of a search would be at most $\log_{100} 1\,000\,000 = 3$ accesses. Two problems are still to be solved: how to keep the structure balanced in presence of insertions and deletions, and how to keep pages almost full.

A solution to both problems is a particular perfectly balanced *multiway* tree structure, called a B–tree, proposed by Bayer and McCreight in 1972, which ensures, in an elegant way, a minimum occupancy of each page with a simple balancing operation. In the following, we first present the properties of the B–tree and then those of a variant called a B^+–tree, which is the one that is commonly used.

5.2 B–trees

For simplicity, as with hashing organizations, we will talk about the *storage and retrieval of keys* rather than the *storage and retrieval of records with a given key value*, and we denote a record entry with key k as $k*$. Moreover, we assume that records have the same and fixed size, and that keys are integers.

Definition 5.1 A B–tree of *order* m ($m \geq 3$) is an m-way search tree that is either empty or of height $h \geq 1$ and satisfies the following properties:

1. Each node contains at most $m - 1$ keys.
2. Each node, except the root, contains at least $\lceil m/2 \rceil - 1$ keys. The root may contain any number n of keys with $n \leq m - 1$.
3. A node is either a leaf node or has $j + 1$ children, where j is the number of keys of the node.
4. All leaves appear on same level.
5. Each node has the following structure:

$$[p_0, k_1*, p_1, k_2*, p_2, \ldots k_j*, p_j]$$

where:

- The keys are sorted: $k_1 < \ldots < k_j$.
- p_i is a pointer to another node of the tree structure, and is undefined in the leaves.
- Let $K(p_i)$ be the set of keys stored in the subtree pointed by p_i. For each non-leaf node, the following properties hold:
  - $\forall y \in K(p_0), y < k_1$
  - $\forall y \in K(p_i), k_i < y < k_{i+1}, i = 1, \ldots j - 1$
  - $\forall y \in K(p_j), y > k_j$

Definition 5.2 *Height*

The *height* h of a B–tree is the number of nodes in a path from the root to a leaf node.

Figure 5.2 shows an example of a B–tree of order 5, with height 3. Note that if the tree is visited according to the *in-order* traversal, all the keys will be visited in ascending order of their values.

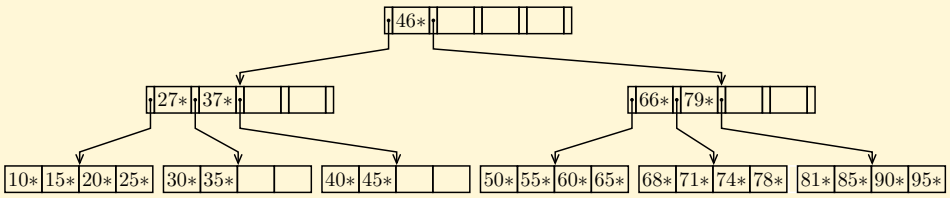


Figure 5.2: Example of a B–tree with $m = 5$

5.2.1 Operations

Search. The search of a key k starts at the root node. If the key is not in the root, and $h > 1$, the search continues as follows:

1. If $k_i < k < k_{i+1}, 1 \leq i \leq m$, then the search continues in the subtree p_i.
2. If $k_m < k$, then the search continues in the subtree p_m.
3. If $k < k_1$, then the search continues in the subtree p_0.

If the key value is not found in a leaf node, the search is unsuccessful, otherwise the search cost is $\leq h$.

Insertion. The insertion of a key k into a B–tree is also quite simple. First, a search is made for the leaf node which should contain the key k. An unsuccessful search determines the leaf node Q_1 where k should be inserted.

If the node Q_1 contains less than $m - 1$ keys, then k is inserted and the operation terminates. Otherwise, if Q_1 is full, it will be split into two nodes, with the first half of the m keys that remain in the old node Q_1, the second half of the keys that go into a new adjacent node Q_2, and the median key, together with the pointer to Q_2, that is inserted into the father node Q of Q_1, repeating the insertion operation in this node. This splitting and moving up process may continue if necessary up to the root, and if this must be split, a new root node will be created and this increases the height of the B–tree by one.

Note that the growth is at the *top* of the tree, and this is an intrinsic characteristic of a B–tree to ensure the important properties that it always have all the leaves at the same level, and each node different from the root is at least 50% full.

Alternatively, if an adjacent brother node of Q_1 is not full, the insertion can be performed without splitting Q_1 by applying a *rotation* technique, as explained for the deletion operation.

Example 5.1
Let us show the effect of the insertion of the key 70 in the B–tree represented in Figure 5.2.

1. The key 70 must be inserted in the node Q_1.

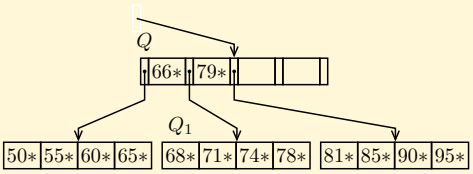


Figure 5.3: Insertion of key 70 – Step 1

2. The node Q_1 is full, and it is split into two nodes Q_1 and Q_2, and the median key 71, together with the pointer to Q_2, is inserted into the node Q.

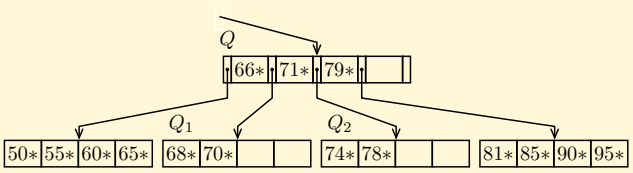


Figure 5.4: Insertion of key 70 – Step 2

Deletion. Key deletion is slightly more complicated. If a key from a non-leaf node is deleted, it will be replaced by the next following key, which is in a leaf node, and therefore the effect of a key deletion is always on a leaf. Furthermore, after a deletion if the leaf node p has less than $\lceil m/2 \rceil - 1$ keys, it has to be regrouped with an adjacent brother node in order to respect the definition of B–tree, using one of the following techniques: *merging* or *rotation*.

The node p is *merged* with one of its adjacent brother nodes which contains $\lceil m/2 \rceil - 1$ keys operating in a way that is exactly the inverse to the process of division.

Merging is illustrated by Figure 5.4. If key 70 is deleted from node Q_1, it becomes underfull and it is merged with the brother to the right Q_2. The key 71 separating the two nodes in the ancestor Q is no longer necessary and it too is added to the single remaining leaf Q_1, so the tree will become that shown in Figure 5.3.

The elimination of 71 from Q can cause a further underflow by requiring the merging of Q with one of its adjacent brothers. In such a case, the process is applied recursively and terminates upon encountering a node that does not need be merged or if the root node is used. If the root node contains a single key, as a result of the merging, it becomes empty, and is removed. The result is that the B–tree shrinks from the top. Thus the deletion process reverses the effects of the insertion process.

When the *merging* of the node p with one of its adjacent brothers is not possible, then the *rotation* technique is applied.

Rotation is illustrated by Figure 5.5a. If key 70 is deleted from Q_2, it becomes underfull and a rotation is performed to borrow the maximum key 65 from the brother to the left Q_1. The key 65 is moved in the ancestor node Q and replace the key 66 which is moved in Q_2 as the new smallest key value. The tree will become that shown in Figure 5.5b.

Data Loading. The initial B–tree structure depends on the order in which the keys are loaded. For example, the B–tree in Figure 5.2 is the result of loading the keys in the following order:

10, 15, 30, 27, 35, 40, 45, 37, 20, 50, 55, 46, 71, 66, 74, 85, 90, 79, 78, 95, 25, 81, 68, 60, 65.

If the keys to load are sorted, the result is a B–tree with the leaves filled at 50%, except the last one, as shown in Figure 5.6 for the keys:

50, 55, 66, 68, 70, 71, 72, 73, 79, 81, 85, 90, 95.

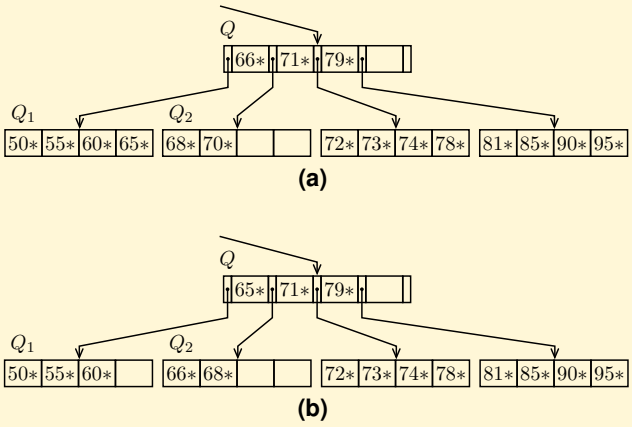


Figure 5.5: A rotation example

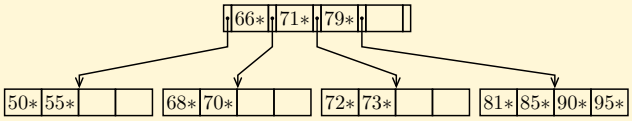


Figure 5.6: The resulting B–tree from loading a sorted set of keys

One way to improve memory usage is to load the keys sorted, and to use the rotation technique: instead of splitting a full node different from the first, the rotation technique is applied with the brother node to the left until it does not fill completely. For the keys of the previous example, the tree in Figure 5.7 is obtained.

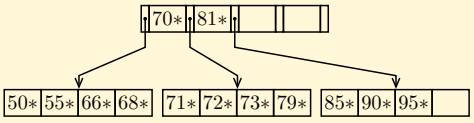


Figure 5.7: The resulting B–tree from loading a sorted set of keys with the rotation technique

5.3 Performance Evaluation

Let us evaluate the costs of the operations expressed in terms of the number of nodes to read and write, assuming that the memory buffer can hold $h+1$ nodes: In this way the nodes involved in the individual operations are transferred into the buffer only once.

B–tree Height. As with any tree, the cost of operations depend on the height of the tree. The following theorem establishes the important relation between N, m and the height of a B–tree:

Theorem 5.1

In a B–tree of order m, with height h and with $N \geq 1$ keys, the following relation holds

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

Proof

A B–tree of order m with height h, has the minimum (b_{min}) or the maximum (b_{max}) number of nodes when they are filled with the minimum or maximum number of keys:

$$\begin{aligned} b_{\mathsf{min}} &= 1 + 2 + 2\lceil m/2 \rceil + 2\lceil m/2 \rceil^2 + \cdots + 2\lceil m/2 \rceil^{h-2} \\ &= 1 + 2\frac{\lceil m/2 \rceil^{h-1} - 1}{\lceil m/2 \rceil - 1} \\ b_{\mathsf{max}} &= 1 + m + m^2 + \cdots + m^{h-1} \\ &= \frac{m^h - 1}{m - 1} \end{aligned}$$

The minimum (N_{min}) and the maximum (N_{max}) number of keys that can appear in a structure of order m with height h are:

$$\begin{aligned} N_{\mathsf{min}} &= \textit{root} + (\textit{min number of keys per node}) \times (b_{\mathsf{min}} - 1) \\ &= 1 + (\lceil m/2 \rceil - 1) \times (b_{\mathsf{min}} - 1) \\ &= 2\lceil m/2 \rceil^{h-1} - 1 \\ N_{\mathsf{max}} &= (\textit{max number of keys per node}) \times b_{\mathsf{max}} \\ &= (m-1) \times b_{\mathsf{max}} \\ &= m^h - 1 \end{aligned}$$

Therefore the following relation holds:

$$2\lceil m/2 \rceil^{h-1} - 1 \leq N \leq m^h - 1$$

and passing to logarithms, the thesis follows. ■

Table 5.1 shows the minimum and maximum height of a B–tree with the specified values of N and m, assuming records of 100 bytes and pointers of 4 bytes. Note as with large pages the height is low also for large sets of records.

Table 5.1: Minimum (h_{min}) and maximum (h_{max}) height of a B–tree, by the size of the pages (D_{pag})

		N							
		1000		10 000		100 000		1 000 000	
D_{pag}	m	h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}
512	5	4.3	6.7	5.7	8.8	7.2	10.8	8.6	12.9
1024	10	3.0	4.9	4.0	6.3	5.0	7.7	6.0	9.2
2048	20	2.3	3.7	3.1	4.7	3.8	5.7	4.6	6.7
4096	40	1.9	3.1	2.5	3.8	3.1	4.6	3.7	5.4

Memory Requirements. Let us assume that each node is 70% full on an average. The number of nodes b of a B–tree of order m to store N keys is

$$b = (m'^h - 1)/(m' - 1) = N/(m' - 1)$$

where $m' = 0.7m$ and $h = \log_{m'}(N + 1)$.

Equality Search. The cost of a search is $1 \leq C_s \leq h$ reads.

Range Search. B–trees are very good for equality searches, but not to retrieve data sequentially or for range searches, because they require multiple tree traversals. This is the reason why a variation of this tree structure is usually used. Let us consider the tree in Figure 5.8. To retrieve all records with the keys in increasing order, the tree is visited in the *in-order* (*symmetric*) traversal and so the nodes are visited as follows: 1, 2, 4, to find the deepest node to the left with the minimum key, and then upward and downward for the sequential search, 2, 5, 2, 6, 1, 3, 7, 3, 8, 3, 9, 1.

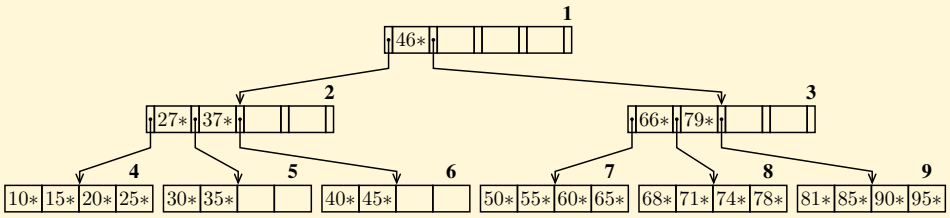


Figure 5.8: Example of a B–tree with $m = 5$

Insertion. An insertion is made in a leaf node. If the node is not full, the new key is inserted keeping the node's keys sorted. The cost is h reads and 1 write.

The worst case is when the node and all its parent must be split. The cost is h reads and $2h + 1$ writes.

Deletion. The cost of the operation is estimated by considering three cases.

1. If the key is in a leaf, and the merging and rotation operations are not required, the cost is h reads and 1 write.

2. If the key is in a node, and the merging and rotation operations are not required, the cost is h reads and 2 writes.
3. The worst case is when for all the nodes of the path from the root to the node, except for the first two, the merging operation is required, and for the child of the root a rotation operation is required. The cost is $2h - 1$ reads and $h + 1$ writes.

5.4 B^+–trees

A B^+–tree is a well known B–tree variant to enhance search performance especially for range queries. In a B^+–tree *all the records*, denoted as $k*$, are stored sorted in the leaf nodes, organized into a doubly linked list. Only the set of the *highest keys*, denoted as k, in each leaf node, except the last one, is stored in the non-leaf nodes, organized as a B–tree (Figure 5.9).

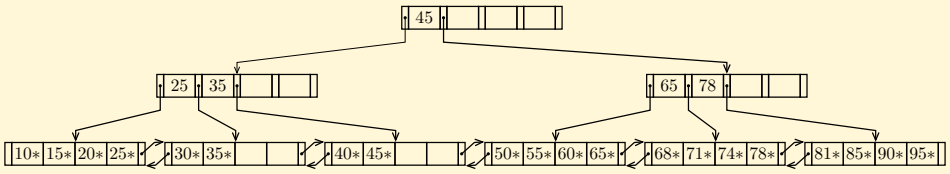


Figure 5.9: A B^+–tree

This organization is also called *index-organized file*, *index sequential file* or *clustered indexed file* to emphasize that the records are stored in the leaves, like a sequential organization, and the set of the other nodes of the B^+–tree are like a *sparse* index with an entry for the highest key of the records in each leaf. In the following, we will call it *index sequential organization*.

Note that records are part of the tree structure stored in one file, therefore, to read all records in sorted order, the tree structure *must* be used to locate the first data page. Moreover, since the organization is a primary one, there can be at most one index sequential organization for a table.

A B^+–tree, also called B–tree in some textbooks, is different from a B–tree for the following reason:

1. A key search requires always the same number of accesses equal to the B^+–tree height.
2. A B^+–tree is usually *shallower* than a B–tree with the same data, and so a key search is faster, because the records are stored in the leaf nodes, and only the keys are stored in the *non-leaf* nodes of the tree.
   Moreover, since all records are in the leaf nodes, a sequential scan of data or a range key search are faster.
3. When a leaf node F is split into F_1 and F_2, a *copy* of the highest key in F_1 is inserted in the father node of F, while when an internal node I is split the *median key* in I is *moved* in the father node, as in a B–tree.
4. When a record with the key k_i is deleted, the k_i* is deleted in the leaf F, and if k_i is used in a father node because it was the highest key in F, it is not necessary to replace it with the new highest key in F. Only when the deletion of a record causes the number of the records in F to fall below the minimum, it is necessary to reorganize the tree.

Index sequential organization in DBMSs. In INGRES a table R is stored initially with a heap organization, and then it is possible to select an index sequential organization, static (**ISAM**) or dynamic (**BTREE**), with the command:

MODIFY R **TO** (**ISAM** | **BTREE**) **UNIQUE ON** Attr{, Attr};

In Oracle the index sequential organization, called IOT (*index organized table*), is used when a table with a primary key is created with the clause **ORGANIZED INDEX**:

CREATE TABLE R(Pk Type **PRIMARY KEY**, ...) **ORGANIZED INDEX**;

In SQL Server the index sequential organization is used when a **CLUSTERED INDEX** is defined on the primary key of a table.

CREATE TABLE R(Pk Type **PRIMARY KEY**, ...) ;
CREATE CLUSTERED INDEX Rclust **ON** R(Pk);

The index sequential organization is one of the oldest types of organizations used in file and database systems. This organization was called *Virtual Storage Access Method* (VSAM) when it was adopted for IBM file systems, and is called with different names in commercial DBMSs (see the box).

Static Tree-Structure Organization

A B^+–tree organization can be used to implement a static index structure rather than a dynamic one. The tree structure is fixed at loading time, and insertions into full leaf nodes are treated as page overflows of a static hashing organization. This solution is simple to implement, but has the usual inconveniences of a static organization.

This organization was called *Index Sequential Access Method* (ISAM), and it was used initially for IBM file systems, and by some DBMSs such as Ingres.

5.5 Index Organization

To support fast retrieval of records in a table using different keys, two types of indexes can be used, in accordance with the table organization:

1. If the table is stored with a *heap organization*, an index is defined for each key, and the elements of the indexes are pairs (k_i, rid_i), where k_i is a key value for a record, and rid_i is the record identifier.

   Some database systems also permit one of the indexes on a relation to be declared to be *clustered*. In DB2 this type of index is created with the command **CREATE INDEX** Name **ON** Table(Attributes) **CLUSTER** (Figure 5.10).

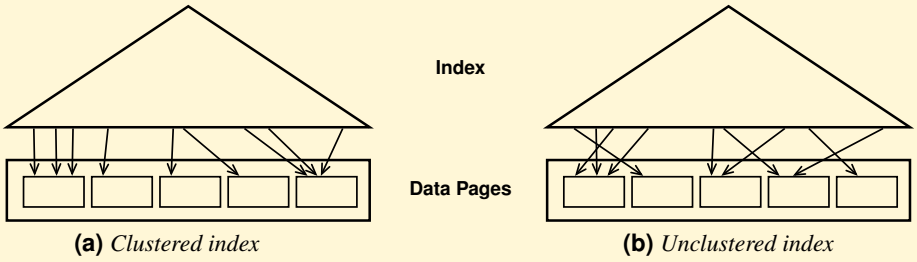


Figure 5.10: Types of indexes

Definition 5.3 *DB2 Clustered Index*

A *clustered index* on a key K of a table is created by first sorting the table on the values of the index key K. If new records are inserted into the table after the clustered index is created, the efficiency of the index decreases because the table records are no longer in the physical order of the index key. In order to overcome this problem, when a clustered index is created it is possible to specify that a small fraction of each data page is left empty for future insertions, and then the clustered index should be recreated from time to time.

A clustered index is particularly useful for a range key search because it requires fewer page accesses, as it will be shown in the following section.

Note that the term *clustered index* is used with different meanings, i.e. *index sequential primary organization*, as in SQL Server, or *clustered index secondary organization*, as in DB2.

2. If the table is stored with a *dynamic primary organization* using the "primary key", indexes are defined for the other keys, and the element of the indexes are pairs (k_i, pk_i), where k_i is a key value for a record, and pk_i is the primary key value of the corresponding record. If the primary organization is *static*, the elements of the indexes are pairs (k_i, rid_i), as in the previous case.

Since an index can be viewed as a table, if it is large, it is stored usually using a B^+–tree primary organization.

5.5.1 Range Search

An index is useful for a range search if the interval is not too large, otherwise it is better to proceed with a scan of the data pages to find records that satisfy the condition, as shown by the following analysis.

Unclustered Index. Let R be a table with key K and $N_{\mathsf{rec}}(R)$ records, stored in $N_{\mathsf{pag}}(R)$ pages, and $N_{\mathsf{leaf}}(\mathsf{Idx})$ be the number of leaves of an unclustered B^+–tree index Idx on K

The selectivity factor s_f of the condition $(\psi = v_1 \leq k \leq v_2)$ is an estimate of the fraction of records which will satisfy the condition. With numerical keys and uniform distribution of the values, $s_f(\psi)$ is estimated as

$$s_f(\psi) = \frac{v_2 - v_1}{k_{\max} - k_{\min}}$$

To find a record R using the index, first the RID of the record matching the condition is retrieved from a leaf of the index, and then the record of R is retrieved with one access. Let us assume that the cost of an index access is estimated with the number of leaves to visit $\lceil s_f(\psi) \times N_{\text{leaf}}(\text{Idx}) \rceil$, and the number of RID of the records that satisfy the condition is

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

Since the records are not sorted on the index key values, the number of pages to read is E_{rec}, and therefore the search cost using the index is

$$C_s = \lceil s_f(\psi) \times N_{\text{leaf}}(\text{Idx}) \rceil + \lceil s_f(\psi) \times N_{\text{rec}}(R) \rceil = \lceil s_f(\psi) \times (N_{\text{leaf}}(\text{Idx}) + N_{\text{rec}}(R)) \rceil$$

The search of the records with a table scan has the cost $N_{\text{pag}}(R)$, therefore the index is useful if

$$\lceil s_f(\psi) \times (N_{\text{leaf}}(\text{Idx}) + N_{\text{rec}}(R)) \rceil < N_{\text{pag}}(R)$$

Example 5.2
Let R be a table with key K and 500 000 records, stored in $N_{\text{pag}}(R) = 60\,113$ pages. Let us estimate for which value of $s_f(\psi)$ an unclustered B^+–tree index on K, with $N_{\text{leaf}}(\text{Idx}) = 6012$, is useful to retrieve the records with the keys within a certain range:

$$\lceil s_f(\psi) \times (6012 + 500\,000) \rceil < 60\,113$$

the inequality holds for $s_f < 0.12$, that is for very selective conditions.

In this case the use of the index has also the advantage of returning the records sorted according to the key, while to obtain the same result with a table scan the cost of sorting E_{rec} records should also be taken into account.

Clustered Index. The records are always retrieved using the index since, although it has been constructed from sorted records, in the case of subsequent insertions it is not certain that the records are still sorted (i.e. the records are *almost* sorted). Therefore, the cost of a search is estimated as the sum of the cost of accessing the leaves of the index, ignoring the height of the index, and the cost of accessing the data pages.

When the records are sorted on the values of the key, the clustered index is almost always convenient. In fact, the number of data pages to visit to find E_{rec} records is estimated as

$$\lceil f_s(\psi) \times N_{\text{pag}}(R) \rceil$$

and the overall cost of the search with the use of the index becomes

$$C_s = \lceil f_s(\psi) \times (N_{\text{leaf}}(\text{Idx}) + N_{\text{pag}}(R)) \rceil$$

For the search operation the index is advantageous with respect to a table scan if

$$\lceil f_s(\psi) \times (N_{\mathsf{leaf}}(\mathsf{Idx}) + N_{\mathsf{pag}}(R)) \rceil < N_{\mathsf{pag}}(R)$$

With the data of the previous example, the index is advantageous when $f_s(\psi) < 0.9$, that is also with non-selective conditions.

Indexes with Variable Length Keys

If the keys have string values with a very variable length, and such as to prohibit the reduction to the case of fixed length equal to the maximum possible, there are several possible solutions to reduce the space occupied by the keys and then increase the number of node's children in the non-leaf nodes of a B^+–tree.

The idea is to reduce the key length in the non-leaf nodes by truncating the keys to the fewest number of characters needed to distinguish them from each other. Example of solutions are the *prefix* B^+–tree, *trie*, and the *String* B–tree. Interested readers may refer to the bibliographic notes at the end of this chapter for specific proposals.

5.6 Summary

1. The B–tree is a fundamental search tree, perfectly balanced, for storing a set of records in the permanent memory that must be accessed both sequentially and directly for range key search. It has been described as *ubiquitous* because of its utility.

   What does the B stand for? The tree name has never been explained. Certainly, the B does *not* stand for *binary*. The B could stand for *balanced*, for the Bayer name of the first author, or for the name Boeing Corporation, for which the authors were working at the time.
2. The B^+–tree, a variant of the B–tree which stores all the records in the leaves and constructs a B–tree on the maximum key of each leaf, is the most widely used structure in relational DBMSs for both the primary organization of tables (*index sequential organization*) and index secondary organization.

Bibliographic Notes

Tree organizations for tables of records or indexes are treated in every book cited in the bibliographic notes of Chapter 1.

Bayer and McCreight present the B–tree in [Bayer and Creight, 1972] and in [Comer, 1979; Chu and Knott, 1989] is made a review of many variations of this data structure. A detailed analysis of the performance of B–trees and derivates is given in [Chu and Knott, 1989; Rosenberg and Snyder, 1981].

The *String* B–tree has been proposed in [Ferragina and Grossi, 1995, 1999], and its experimental analysis has been presented in [Ferragina and Grossi, 1996], together with comparisons with alternative structures.

Exercises

Exercise 5.1 The **CREATE TABLE** statement of a relational system creates a heap-organized table by default, but provides the DBA the following command to transform a heap organization into a tree-structure organization:

MODIFY Table **TO ISAM ON** Attribute;

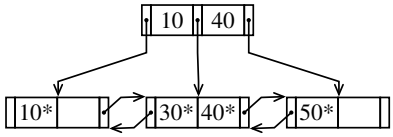
The manual contains the following warning: “Do not modify a table’s structure from its default heap structure to a keyed (i.e. ISAM) structure until the table contains most, if not all, of its data, ..., (otherwise) query processing performance degrade upon adding extra data”. Explain what determines the performance degradation.

Exercise 5.2 Answer the following questions about index and tree organizations:

– What is the difference between an *index secondary organization* and *index sequential organization*?
– What is the difference between a *clustered index* and an *unclustered index*? If an index contains data records as ‘data entries’ can it be unclustered?

Exercise 5.3 Show the result of entering the records with keys in the order (1, 2, 3, 4, 5) to an initially empty B^+–tree of order $m = 3$. In case of overflow, split the node and do not re-distribute keys to neighbors. Is it possible to enter the records with keys in a different order to have a tree of less height?

Exercise 5.4 Show how the following B^+–tree changes after the insertion of the record with key 25.



Exercise 5.5 Consider a DBMS with the following characteristics: a) file pages with size 2048 bytes, b) pointers of 12 bytes, c) the page header of 56 bytes. A secondary index is defined on a key of 8 bytes. Compute the maximum number of records that can be indexed with

1. A three levels B–tree.
2. A three levels B^+–tree. For simplicity, assume that the leaf nodes are organized into a singly linked list.

Exercise 5.6 Consider a secondary index on a primary key of a table with N records. The index is stored with a B^+–tree of order m. What is the minimum number of nodes to visit to search a record with a given key value?

Exercise 5.7 Discuss the advantages and disadvantages of a B–tree and a *static hashing* primary organizations.

Exercise 5.8 Let $R(K, A, B, \mathit{other})$ be a relation with $N_{\mathsf{rec}}(R) = 100\,000$, $L_r = 100$ bytes, a key K with integer values in the range (1, 100 000) and $L_K = 4$. Suppose R stored with heap organization in pages with size $D_{\mathsf{pag}} = 1024$ bytes and a loading factor $f_r = 0,8$, and an index exists on the key K stored as B^+–tree,

Estimate the cost of the following SQL queries, assuming that there are always records that satisfy the **WHERE** condition.

1. **SELECT** *
   FROM R;

2. **SELECT** *
   FROM R
   WHERE K = 50;

3. **SELECT** *
   FROM R
   WHERE K **BETWEEN** 50 **AND** 100;

4. **SELECT** *
   FROM R
   WHERE K **BETWEEN** 50 **AND** 100
   ORDER BY K;

Exercise 5.9 Let $R(A, B, C, D, E)$ be a relation with key A, $N_{\mathsf{pag}}(R) = 10\,000$, $N_{\mathsf{rec}}(R) = 100\,000$ and $D_{\mathsf{pag}} = 500$ bytes. The values of all attributes are strings with length 10 bytes. Consider the query

SELECT A, B
FROM R
ORDER BY A;

1. Estimate the query execution cost without indexes.
2. Estimate the query execution cost with a clustered index on A stored as B^+–tree with $N_{\mathsf{leaf}} = 3500$.

Chapter 6

NON-KEY ATTRIBUTE ORGANIZATIONS

The previous two chapters have described organizations to retrieve records of a table with a specified key value in as few accesses as possible. Other important organizations are the ones to retrieve records of a table that satisfy a query which involves non-key attributes, i.e. attributes that do not uniquely identify a record. Non-key attributes are also called *secondary keys* by some authors, but the term is not standard and other authors use it with a different meaning. In the following, for the sake of brevity, sometimes we will just call them *attributes* when the context does not create ambiguity. In this chapter, after a definition of the problem and the type of queries considered, the main organizations will be presented to speed up the search for records in a table that satisfy a condition on one or more non-key attributes.

6.1 Non-Key Attribute Search

The records to retrieve are specified with search conditions of the following types:

1. An *equality search*, which specifies a value v for an attribute A_i ($A_i = v_i$).
2. A *range search*, which specifies a range of values for an attribute A_i ($v_1 \leq A_i \leq v_2$).
3. A *boolean search*, which consists of the previous search types combined with the operators AND, OR and NOT.

These three types of searches do not exhaust all the possibilities, but they are sufficient to show how queries can be very complex and require data organizations different from those seen so far to generate the answer quickly. In this chapter, we will only consider the fundamental solutions for searches of type (1) and (2), or for those of type (3) with conjunctions of simple conditions. The general case will be considered in Chapter 11.

With a primary organization, queries on non-key attributes can only be answered with a scan of the data. With large collections of records and queries satisfied by small subsets (as a general guideline, less than 15% of the records), if the response time is an important requirement, this approach is not worthwhile and the cost and management of an *index* that makes it possible to speed up the search of the records that match the query is justified. Figure 6.1 shows an index on the attribute Quantitiy of Sales, assuming for simplicity that the RIDs are integers.

Sales

RID	Date	Product	City	Quantity
1	20090102	P1	Lucca	2
2	20090102	P2	Carrara	8
3	20090103	P3	Firenze	5
4	20090103	P1	Arezzo	10
5	20090103	P1	Pisa	1
6	20090103	P4	Pisa	8
7	20090103	P2	Massa	5
8	20090104	P2	Massa	2
9	20090105	P4	Massa	2
10	20090103	P4	Livorno	5

Index

Quantity	RID
1	5
2	1
2	8
2	9
5	3
5	7
5	10
8	2
8	6
10	4

Figure 6.1: An *index* on Quantity

Indexes are non-exclusive. Therefore they can be created for any non-key attributes, regardless of the primary organization used to store the table records. An index can be defined also on multiple attributes (*multi-attribute* or *composite index*).

An excessive number of indexes can be harmful to the overall performance. The attributes to be indexed must be selected carefully, and this problem has been extensively studied by a number of researchers.

A typical implementation of an index is the *inverted index* organization described in the next section.

6.2 Inverted Indexes

Definition 6.1

An *inverted index* I on a non-key attribute K of a table R is a sorted collection of entries of the form $(k_i, n, p_1, p_2, \ldots, p_n)$, where each value k_i of K is followed by the number of records n containing that value and the *sorted* RID list of these records (*rid-list*).

Figure 6.2 shows an inverted index on the attribute Quantitiy of Sales.

Sales

RID	Date	Product	City	Quantity
1	20090102	P1	Lucca	2
2	20090102	P2	Carrara	8
3	20090103	P3	Firenze	5
4	20090103	P1	Arezzo	10
5	20090103	P1	Pisa	1
6	20090103	P4	Pisa	8
7	20090103	P2	Massa	5
8	20090104	P2	Massa	2
9	20090105	P4	Massa	2
10	20090103	P4	Livorno	5

Inverted index

Quantity	n	RID list
1	1	5
2	3	1, 8, 9
5	3	3, 7, 10
8	2	2, 6
10	1	4

Figure 6.2: An *inverted index* on Quantity

Although an inverted index requires elements of variable length and, in the case of record updates, a management of the sorted rid-lists, it has a wide use for the following reasons:

– The data file is accessed only to find the records that match the query.
– The result of queries such as "how many records have the index key that satisfy a condition?", or "search the index key values of a set of records that satisfy a condition?", can be found using the index only.
– It allows complete independence of the index organization from the strategy used for data storage. Therefore, the data must not be reorganized at any addition or deletion of an inverted index.

6.2.1 Performance Evaluation

The performance is evaluated in terms of:

– The amount of extra memory needed, not counting the memory needed to store the data file.
– Cost of the search for records with specified values for the indexed attributes, and of the update operations.

The costs will be expressed as a function of the following parameters stored in the database system catalog:

$N_{\mathsf{rec}}(R)$	the number of records of R.
$N_{\mathsf{pag}}(R)$	the number of pages occupied by R.
L_R	the number of bytes to represent the value of $N_{\mathsf{rec}}(R)$.
$N_I(R)$	the number of indexes on R.
L_I	the average number of bytes to represent a key value of the index I.
$N_{\mathsf{key}}(I)$	the number of distinct keys in the index I.
$N_{\mathsf{leaf}}(I)$	the number of leaf nodes in the index I.
L_{RID}	the number of bytes to represent the RID of a record.

Memory Requirements. For simplicity, let us assume that the organization of the indexes, which contain elements of variable length, does not require more memory than strictly necessary.

$$\begin{aligned} \mathcal{M} &= \textit{Index memory} \\ &= \sum_{i=1}^{N_I(R)} N_{\mathsf{key}}(I_i)(L_{I_i} + L_R) + N_I(R) \times N_{\mathsf{rec}}(R) \times L_{\mathsf{RID}} \\ &\approx N_I(R) \times N_{\mathsf{rec}}(R) \times L_{\mathsf{RID}} \end{aligned}$$

The memory required by an inverted index is therefore mainly due to the RIDs stored in the index.

Equality Search. The operation cost can be estimated easily under the following simplifying assumptions:

– The values of an attribute are uniformly distributed across its active domain, and the attributes are considered independent.
– The records are uniformly distributed in the pages of R.
– An index is stored in a B^+–tree with the sorted rid-lists in the leaf nodes.

The operation cost is:

$$C_s = C_I + C_D$$

where C_I is the cost of accessing the index pages to find the RIDs of the records that satisfy the condition, while C_D is the cost of accessing the data pages containing the records. The cost C_I is usually approximated to the cost of accessing the leaf nodes, ignoring the cost of the visit of the path from the root to a leaf node:

$$C_I = \lceil s_f(\psi) \times N_{\mathsf{leaf}}(I) \rceil = \left\lceil \frac{N_{\mathsf{leaf}}(I)}{N_{\mathsf{key}}(I)} \right\rceil$$

Let us consider two cases for C_D, depending on whether the data is sorted or not on the index key.

If the index is unclustered, it is necessary to have an estimate of the number of records E_{rec} satisfying the query condition. For a simple condition $(A_i = v_i)$, E_{rec} is equal to the average length of a rid-list estimated as:

$$E_{\mathsf{rec}} = \left\lceil \frac{N_{\mathsf{rec}}(R)}{N_{\mathsf{key}}(I)} \right\rceil$$

The following formula is used to estimate C_D:

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

where the function $\Phi(k, n)$ is an estimate of the number of pages, in a file of n pages, that contain at least one of the k records to be retrieved using a *sorted* rid-list.

An approximate evaluation of the function $\Phi(k, n)$, usually used in the literature, has been proposed by Cardenas, by assuming pages to have infinite capacity [Cardenas, 1975]:

$$\Phi(k : int, n : int) : int = \left\lceil n \left(1 - \left(1 - \frac{1}{n} \right)^k \right) \right\rceil$$

The formula is justified by the following considerations:

$1/n$	is the probability that a page contains one of the k records.
$(1 - 1/n)$	is the probability that a page does not contain one of the k records.
$(1 - 1/n)^k$	is the probability that a page does not contain any of the k records.
$(1 - (1 - 1/n)^k)$	is the probability that a page contains at least one of the k records.
$n(1 - (1 - 1/n)^k)$	is an estimate of the number of pages that contain at least one of the k records.

A page that contains more than one of the k records to retrieve is read only once because the rid-list is sorted, so $n(1 - (1 - 1/n)^k)$ is also an estimate of the number page accesses.

The function is approximately linear in k when $k \ll n$, while it is close to n for k large and, therefore, $\Phi(k, n) \leq \min(k, n)$, so the number of pages accesses is always less than n. The shape of the function Φ, for a fixed $n = 100$, is shown in Figure 6.3.

The Cardenas' formula has the advantage of a low evaluation cost but appreciably underestimates the cost of Φ in the case of pages with $c < 10$, and has been revised

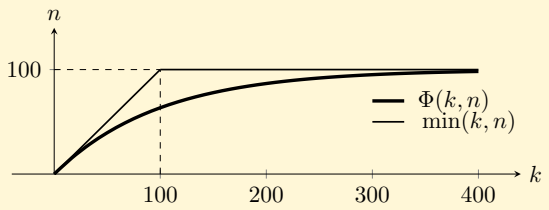


Figure 6.3: Shape of the function Φ

by several researchers. For $c \geq 10$, the error involved in the Cardenas' approximation is practically negligible.
The data access cost C_D in the case of a clustered index, assuming to retrieve the records using a sorted rid-list, is estimated as:

$$C_D = \left\lceil \frac{1}{N_{\mathsf{key}}(I)} \times N_{\mathsf{pag}}(R) \right\rceil$$

Range Search. Let us assume that to retrieve the records that satisfy the condition $v_1 \leq A_i \leq v_2$, an equality search operation is performed for each A_i value in the range. The cost is:

$$C_s = C_I + C_D$$

The cost of accessing the index is due to the visit of the leaves that contain the rid-lists, and it is estimated as:

$$C_I = \lceil s_f(\psi) \times N_{\mathsf{leaf}}(I) \rceil$$

where $s_f(\psi) = (v_2 - v_1)/(\max(A_i) - \min(A_i))$ is the selectivity factor of the condition $v_1 \leq A_i \leq v_2$.

The number of data page accesses C_D is estimated as:

C_D = *NoIndexKeyValues* $\times$ *NoPageAccessesForRidList*

where *NoIndexKeyValues* $= \lceil s_f(\psi) \times N_{\mathsf{key}}(I) \rceil$, while *NoPageAccessesForRidList* depends on the fact that data are sorted or not on the index key.

If the index is unclustered

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

where $\lceil N_{\mathsf{rec}}(R)/N_{\mathsf{key}}(I) \rceil$ is the average length of the rid-lists.

If the index is clustered

$$C_D = \lceil s_f(\psi) \times N_{\mathsf{key}}(I) \rceil \times \left\lceil \frac{1}{N_{\mathsf{key}}(I)} \times N_{\mathsf{pag}}(R) \right\rceil = \lceil s_f(\psi) \times N_{\mathsf{pag}}(R) \rceil$$

Multi-attribute Search. Let us assume that an index exists on each each attribute used in k simple conditions of a conjunction. The cost of accessing the index to find the rid-list of the records that satisfy the conjunctive condition is estimated as

When to create an index An index allows the DBMS to locate a small subset of records in a table more quickly and thereby speeds up response to user queries, but requires memory and increases the cost of updating the attributes on which it is defined. The general problem of choosing the indexes for a database is complex and the reader may refer to the bibliographic notes for specific proposals. In the simplest case the choice of indexes can be made by keeping in mind the following guidelines that are usually found in the manuals of commercial systems:

- Define indexes on attributes with N_{key} high and frequently used to retrieve less than 15% of the records.
- Define indexes on foreign keys to facilitate the implementation of join operations.
- Define more than four indexes for a relation only if the updating operations are rare.

In commercial systems an index is automatically created to enforce a primary key constraint.

$$C_I = \sum_{j=1}^{k} \lceil s_f(\psi_j) \times N_{\mathsf{leaf}}(I_j) \rceil$$

where $s_f(A_i = v_i) = 1/N_{\mathsf{key}}(I_i)$.

The number of records E_{rec} that satisfy the conjunctive condition is estimated as:

$$E_{\mathsf{rec}} = \left\lceil N_{\mathsf{rec}}(R) \prod_{j=1}^{k} s_f(\psi_j) \right\rceil$$

and therefore the number of page accesses is

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

Insertion and Deletion. The $N_I(R)$ indexes on a table R must be updated whenever a record is either inserted into or deleted from R. The operation requires $N_I(R)$ reads and writes of the inverted indexes to update the rid-lists.

6.3 Bitmap indexes

A bitmap is an alternative method of representing a rid-list of an index. Each index element has a bit vector instead of a rid-list.

Definition 6.2

A *bitmap index* I on a non-key attribute K of a table R, with N records, is a sorted collection of entries of the form (k_i, B), where each values k_i of K is followed by a sequence of N bits, where the jth bit is set to 1 if the record jth has the value k_i for the attribute K. All other bits of the bitmap B are set to 0.

Sales		
RID	...	Quantity
1	...	2
2	...	8
3	...	5
4	...	10
5	...	1
6	...	8
7	...	5
8	...	2
9	...	2
10	...	5

Bitmap index				
1	2	5	8	10
0	1	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
1	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	0	0	0
0	0	1	0	0

Figure 6.4: A *bitmap index* on Quantity

Figure 6.4 shows the *bitmap* index for the example of Figure 6.2.

Using bitmaps might seem a huge waste of space, however bitmaps are easily compressed, so this is not a major issue. With efficient hardware support for bitmap operations (AND, OR, XOR, NOT), a bitmap index is better suited to answer queries such as "how many sales of product P1 have been made in Pisa", since the answer is found by counting the 1's of one bit-wise AND of two bitmaps.

Example 6.1
It is interesting to compare the memory occupied by an inverted index and a bitmap index defined on the same attribute, stored with B^+–tree, considering only the memory for the leaves, supposed completely full.
The number of leaves of an inverted index is:

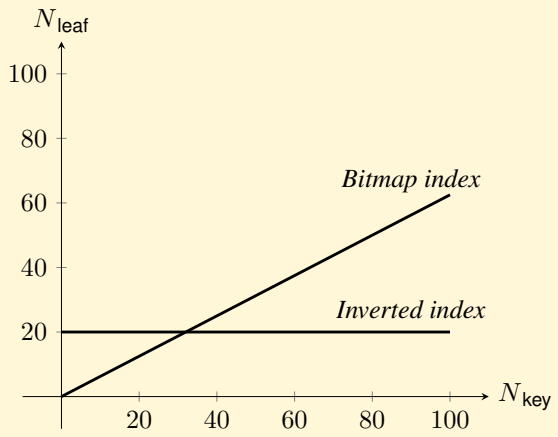
$$N_{\text{leaf}} = (N_{\text{key}} \times L_{\text{k}} + N_{\text{rec}} \times L_{\text{RID}})/D_{\text{pag}} \approx (N_{\text{rec}} \times L_{\text{RID}})/D_{\text{pag}}$$

The number of leaves of a bitmap index is:

$$N_{\text{leaf}} = (N_{\text{key}} \times L_{\text{k}} + N_{\text{key}} \times N_{\text{rec}}/8)/D_{\text{pag}} \approx N_{\text{key}} \times N_{\text{rec}}/(D_{\text{pag}} \times 8)$$

where D_{pag} is the leaves page size in bytes, L_{k} is a value attribute size in bytes and L_{RID} is the RID size in bytes.

For the approximations made, the number of leaves of a B^+–tree index does not depend on N_{key}, while the number of leaves of a bitmap index increases linearly with N_{key}.



The two values are equal for $N_{\text{key}} = 8 \times L_{\text{RID}}$ (for $L_{\text{RID}} = 4$ bytes, $N_{\text{key}} = 32$), for lower values the B^+–tree index takes more memory, while for higher values a bitmap index takes more memory.

An interesting aspect is that although the binary vectors are generally very long, and with very selective attributes (high values of N_{key})) become *scattered*, i.e. a large number of bits will be zero, they can be easily stored in a compressed form to reduce the memory requirements. For example, the Oracle system, which uses such techniques, suggests using them if $N_{key} < N_{rec}/2$.

The bitmap indexes are used when the data is never updated, as happens with *decision support databases* (*Data Warehouse*), because when data change, the operations of modifying the index become complex, particularly when they are compressed.

Index creation in DB2 and Oracle. The following command creates an inverted or bitmap index on a relation R, stored with a heap organization:

CREATE [**UNIQUE** | **BITMAP**] **INDEX** Name **ON** R
(Attr [**ASC** | **DESC**] {, Attr [**ASC** | **DESC**]})

The command also provides the possibility of specifying that the values of the index attributes are unique, that a free space must be left in the index nodes (**PCTFREE**), that the nodes have a certain minimum percentage of filling (**MINPCUSED**), that the leaves of the index are linked by a bidirectional list (**ALLOW REVERSE SCANS** in DB2 only), and finally that in the index must be stored the values of both the key attribute and those of other attributes (**INCLUDE**, in DB2 only), to allow the optimizer to generate more query plans that use indexes only, as we will see later.

An index can be specified as clustered (e.g., **CLUSTER** in DB2).The records of the table on which the index is defined are sorted only at index creation time, but not after overflows from data pages. The command **REORGANIZE** is used to reorganize the clustered index.

6.4 Multi-attribute Index

Let us see how to use inverted indexes to speed up the search of records that satisfy a conjunction of equality or range conditions on a subset of k attributes A_1, A_2, ..., A_k. A more general solutions will be presented in the next chapter.

A query with a condition that uses all the k attributes is called *exact match query*, otherwise is called *partial match query*.

Let $R(A_1, A_2, A_3)$ be a relation with $N_{\mathsf{rec}}(R)$ records. The attributes A_1, A_2, A_3 have n_1, n_2 and n_3 distinct values.

Three inverted indexes on each attribute have $(n_1+n_2+n_3)$ elements, and the total number of RIDs is $3\times N_{\mathsf{rec}}(R)$. To find an exact match query result three rid-lists have to be intersected.

An alternative solution to speed up the search consists in building a *multi-attribute* (*composite*) index on A_1, A_2, A_3, with $(n_1 \times n_2 \times n_3)$ elements, one for each combination of the values of the attributes A_i, with the rid-lists of the records that have those three values in the three attributes. The total number of RIDs is now $N_{\mathsf{rec}}(R)$. This solution is sufficient to find the rid-lists of records that satisfy an exact match query.

For a partial match query, this type of index cannot be always used, as in the case of a condition that uses one attribute different from A_1, or both the attributes A_2, A_3. If, instead, the query uses only the first two attributes A_1, A_2, the index can be used by making the union of n_3 disjoint lists, associated with consecutive elements of the index, which becomes $(n_3 \times n_2)$ if the query uses only A_1.

To ensure the property that for all queries that use any combination of the three attributes A_1, A_2, A_3 it is sufficient to merge the rid-lists of consecutive elements, then indexes are needed for the three combinations

$$(A_1A_2A_3), (A_2A_3A_1), (A_3A_1A_2)$$

Similarly, for four attributes A_1, A_2, A_3, A_4 the indexes are needed for the following six combinations

$$(A_1A_2A_3A_4), (A_2A_3A_4A_1), (A_3A_4A_1A_2), (A_4A_1A_2A_3), (A_2A_4A_1A_3), (A_3A_1A_4A_2)$$

In general, for n attributes and $t = \lceil n/2 \rceil$, to ensure that for any combination of i attributes, with $1 \leq i \leq n$, an index exists to execute the query, their number is $\binom{n}{t}$ [Knuth, 1973].

6.5 Summary

1. An index is the standard data structure used by all DBMSs to speed up the search for records in a table that satisfy a condition on one or more non-key attributes.
2. An index can be added or removed without any effects on the organization of the relations.
3. An index can be implemented as an inverted or bitmap index.

Bibliographic Notes

Index organizations are presented in every book cited in the bibliographic notes of Chapter 1.

Algorithms to choose indexes for a relational database have been proposed by many authors. For an introduction to the problem see [Albano, 1992] and for specific proposals see [Finkelstein et al., 1988; Chaudhuri and Narasayya, 1997].

Exercises

Exercise 6.1 To speed up the search for records in a table with an equality predicate on a non-key attribute A, and selectivity factor f_s, is preferable:

1. A sequential organization on a key attribute.
2. A static hash organization on a key attribute.
3. An inverted index on A.

Briefly justify the answer and give an estimate of query execution cost in all three cases.

Exercise 6.2 Consider a relation R with N_{rec} records stored in N_{pag} of a heap file, and an inverted index on the attribute A with N_{key} integer values in the range A_{min} and A_{max}. Show two different execution plans to evaluate a non-key range query with condition $k_1 \leq A \leq k_2$, and their estimated cost. Explain in which case one is better than the other.

Exercise 6.3 Consider the relation $R(A, B, C, D, E)$ with the key A, and each attribute a string 10 characters long. Assume that $N_{pag}(R) = 10\,000$, $N_{\mathsf{rec}} = 100\,000$ and $D_{\mathsf{pag}} = 500$. Consider the following query:

```
SELECT    A, B
FROM      R
ORDER BY  A;
```

a. Estimate the cost of a plan without the use of indexes.
b. Estimate the cost of a plan with the use of a clustered index on B stored with a B^+–tree with $N_{\mathsf{leaf}} = 2500$.
c. Estimate the cost of a plan with the use of a clustered index on A stored with a B^+–tree with $N_{\mathsf{leaf}} = 2500$.
d. Estimate the cost of a plan with the use of a clustered index on A, B stored with a B^+–tree with $N_{\mathsf{leaf}} = 5000$.

Exercise 6.4 Which of the following SQL queries execution takes less advantage from the presence of a multi-attribute index on R with A as the first attribute and B as the second attribute?

1. **SELECT** $*$ **FROM** R **WHERE** A = 10;
2. **SELECT** $*$ **FROM** R **WHERE** B = 20;
3. **SELECT** $*$ **FROM** R **WHERE** A $<$ B;
4. **SELECT** $*$ **FROM** R **WHERE** A $<$ C;
5. **SELECT** $*$ **FROM** R **WHERE** C $<$ 100 **AND** A = 10.

Exercise 6.5 Discuss the advantages and disadvantages of bitmap indexes.

Exercise 6.6 Consider a relation R with an unclustred index on the numerical non-key attribute B. Explain whether to find all records with $B > 50$ is always less costly to use the index.

Chapter 7

MULTIDIMENSIONAL DATA ORGANIZATIONS

Multidimensional or *spatial* data is used to represent geometric objects and their position in a multidimensional space. Examples of systems that use this type of data are: Geographical Information Systems (GIS), Computer Aided Design and Manufacturing Systems (CAD/CAM) and Multimedia Database Systems. The organizations seen in the previous chapters are not suitable for efficient handling of these types of data. This chapter first describes typical kinds of multidimensional data commonly used in practice, and then presents organization techniques supporting efficient evaluation of basic queries.

7.1 Types of Data and Queries

Let us consider multidimensional data representing points or regions in a k-dimensional space.

The data that represents points is encountered both in applications that deal with geographic data objects and in applications where the information to be managed can be interpreted as spatial points for the purpose of a search. For example, the k attributes of the records of a relation can be interpreted as coordinates of a k-dimensional space and a table of N_{rec} record as N_{rec} spatial points.

Example 7.1
Consider a set of 8 records with two attributes A_1 and A_2 of type integer, which represent the latitude and longitude of cities, whose name will be used to denote the corresponding record. It is assumed that the A_1 and A_2 values are normalized in the range 0 to 100 (Figure 7.1).

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

Figure 7.1: Data on cities locations

A two-dimensional representation of these points, as shown in Figure 7.2, allows a simple geometric interpretation of exact match queries.

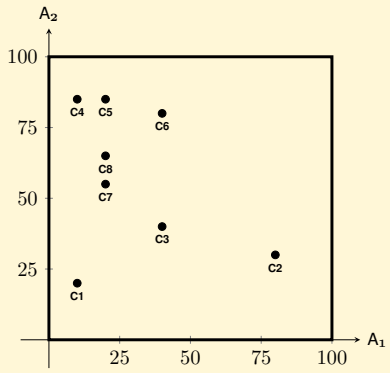


Figure 7.2: Multidimensional representation of data on cities

The problems that will be considered are (a) how to partition the space in regions that contain records that can be stored in a page, (b) how to quickly find the region containing the points in a specified rectangular area. Another interesting query is that to find *nearest neighbors*, e.g. to find a point nearest to a given one, and specific solutions have been developed. This topic is outside the scope of this book. Interested readers may refer to the bibliographic notes at the end of this chapter for specific proposals.

Let us see with an example how we can proceed to divide the data space into non-overlapping partitions of variable size. The information on the partitions (points present in them and their location) is then usually managed with trees of different kinds in order to facilitate search operations.

Example 7.2
Suppose that pages have a capacity 2 and we want to load the data on cities. After the insertion of "C1" and "C2", the insertion of "C3" requires distributing the data into two pages. To proceed let us divide the data space into non-overlapping partitions according to the first coordinate by choosing a value of separation d for A_1: points with coordinate $A_1 \leq d$ are inserted into a page, those with a higher value are inserted into another one. The separator d can be half the range size (subdivision guided by the size of the range) or the median

value of the coordinates of the points present in the region (subdivision guided by the values of the attribute). Suppose we choose the first policy (Figure 7.3a).

When there is a new overflow from a page during data loading, we proceed with another split of the partition, but changing the reference coordinate, with the logic that the splitting dimension alternates, since there are only two dimensions. In general with n dimensions, the splitting dimension cycles. Therefore, after a partition splits along axis A_i then the next one will be along the axis A_{i+1}, and when $i = k$, A_{i+1} becomes A_1 again. Figure 7.3b shows the situation after the insertion of "C4". Then "C5" is inserted, but when "C6" is inserted, a new split is made along A_1 (Figure 7.3c). Then "C7" is inserted, and a new split is made along A_2, and finally once "C8" is inserted we obtain the situation in Figure 7.3d.

Later we will see examples of tree structures to manage the information on the division of the data space into non-overlapping partitions.

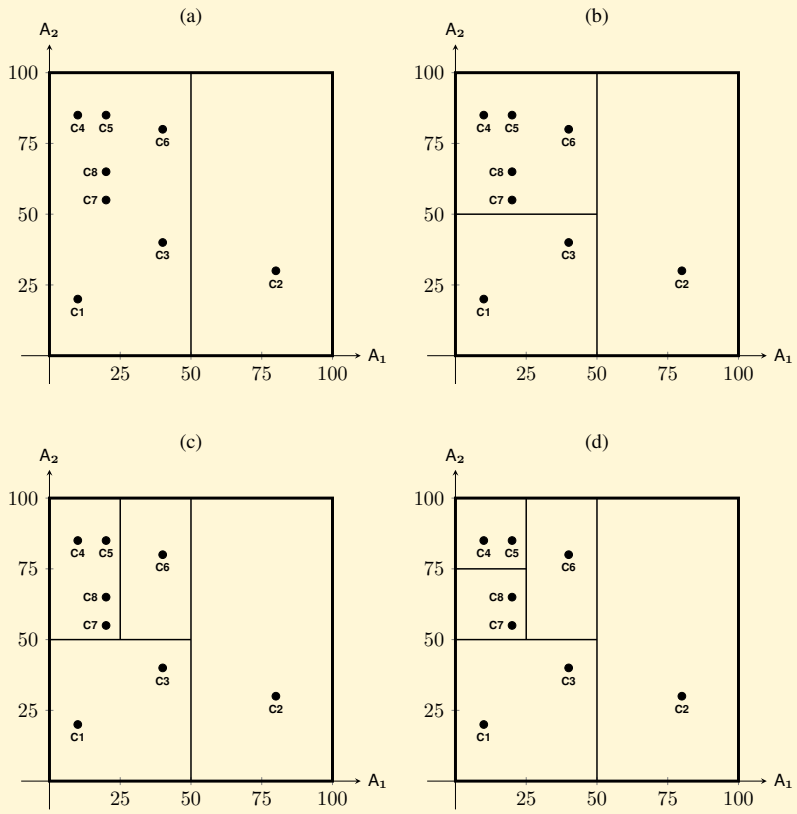


Figure 7.3: Division of the data space into non-overlapping partitions

Let us show with an example why *multi-attribute indexes* do not support queries on *multidimensional data* very well.

Example 7.3
Let us consider a multi-attribute index on the data in Figure 7.2 and how the points are stored in the leaf nodes of a B^+–tree with capacity 2 (Figure 7.4).

The dotted lines indicate the linear order in which points are stored in a B^+–tree and the boxes how points are stored in a multidimensional index.

Let us compare the behavior of the two solutions using the following queries:

1. $A_1 < 25$: the B^+–tree index performs very well. As we will see, the multidimensional index handles such a query quite well too.
2. $A_2 < 25$: the B^+–tree index is of no use, while the multidimensional index handles this query just as well as the previous query.
3. $A_1 < 25 \wedge A_2 < 25$: both solutions perform well.
4. *Nearest neighbor queries*: the multidimensional index is ideal for this kind of queries while B^+–tree is not because it linearizes the 2-dimensional space by sorting entries first by A_1 and then by A_2, and therefore in a page there is no nearest neighbor points, as happens in the partitions of the multidimensional index.

A1	A2	RID
10	20	...
10	85	...
20	55	...
20	65	...
20	85	...
40	40	...
40	80	...
80	30	...

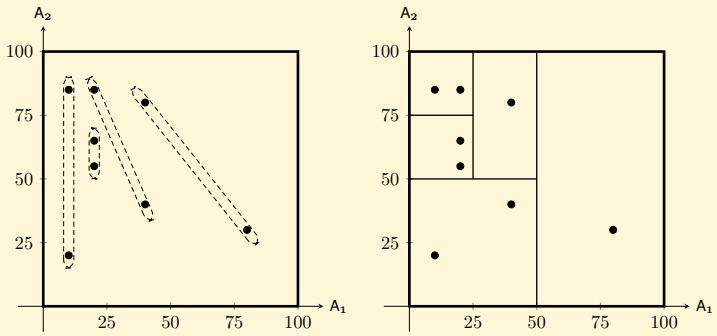


Figure 7.4: Clustering of a multi-attribute index entries in a B^+–tree vs. multidimensional index

Starting from the seventies, many organizations for multidimensional data have been proposed, most of them to deal with point data or rectangles. The attention to the points data management is due to the fact that this kind of data is the basic one to deal with. The rectangles instead are important because they are often used to approximate other geometric shapes such as polygons and irregular shapes.

In the following we present two examples of organizations for multidimensional data: G–trees for point data and B^*–tree for rectangle data.

7.2 G–trees

G–trees combine the ideas of data space partitioning and of B^+–trees in an original way: data space is divided into non-overlapping partitions of variable size identified by an appropriate code, then a total ordering is defined for partition codes, and they are stored in a B^+–tree [Kumar, 1994]. In the following, for simplicity, we consider only the two-dimensional case, and it is assumed that data pages may contain 2 points.

A partition code is a binary string constructed as follows (*partition tree*, Figure 7.5b):

– The initial region is identified by the empty string.
– With the first split along X-axis, the two partitions produced are identified with the strings "0" and "1". The points with $0 < x \leq 50$ belong to the partition "0" and those with $50 < x \leq 100$ belong to the partition "1" (splitting by interval size).
– When a partition of the previous step is split along the Y-axis, the new partition codes become "00" an "01", an so on.
– In general, when a partition R with the code S is split, the subpartition with values less than the half interval has the code S"0" and that with values greater has the code S"1".

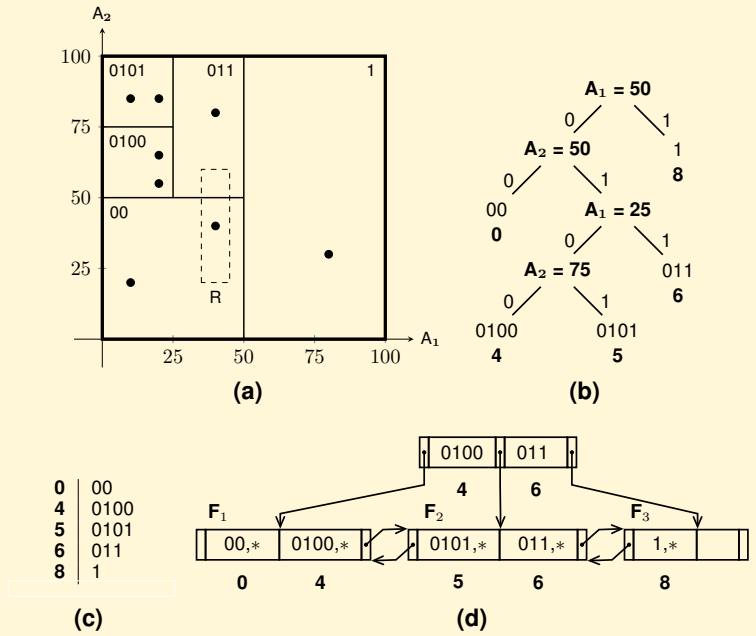


Figure 7.5: An example of partition coding and of a G–tree

Let us see some important properties of the partition coding:

– If a partition R has the code S, the code of the partition from which R has been created with one split is S without the last bit.
– The length of the code of a partition R is the number of split made to get R from the initial space.
– Let *RegionOf*(S) be a function that maps the code S of a partition R in the coordinates of the lower left and upper right vertices of the partition. For example, *RegionOf*("00") = $\{(0,0),(50,50)\}$ and *RegionOf*("011") = $\{(25,50),(50,100)\}$.
– A total ordering for the partition codes is defined as follows: $S_1 < S_2$ if S_1 is a prefix of S_2, or the first most significant bit of S_1 is less than the correspondent one of S_2, or S_1 and S_2 have the same first n bits, and the $(n+1)$-th bit of S_1 is less than the correspondent one of S_2.

The sorted partition codes are stored in a B^+–tree, called G–tree (Figure 7.5d). In each element (S, p) of a leaf node (represented in the figure as $S*$), S is a partition code without subpartitions, with data stored in the page p. In each non-leaf node of the form

$$(p_0, S_1, p_1, S_2, p_2, \ldots, S_m, p_m)$$

the usual relations among the elements of a node in a B^+–tree hold, where the keys are the codes S_i.

Let us see how to proceed for point or range search. In order to facilitate the interpretation of the use that will be made of the binary string of a partition code, and of the operations on G–tree, it is useful to associate integer encodings to the partition as follows.

Let M be the maximum number of splits made. Each partition code less than M bits long is padded with trailing zeros and translated into decimal (Figure 7.5c). The decimal form of the partition codes are shown in the partition tree and in the G–tree. It is important to note that (a) the integer encoding of partitions, as well as the partition tree, are not part of the G–tree, and (b) they must change if the value of M changes because of insertion or deletion of points that change the G–tree structure.

Point Search. Let M be the maximum length of the partition numbers in the G–tree. The search of a point P with coordinates (x, y) proceeds as follows:

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

Example 7.4
Let us search the point $P = (30, 60)$ in the G–tree of Figure 7.5d with $M = 4$.
From the partition tree it is found that it could be in the region $S_P = 011$, i.e. the region 6, that is in the leaf F_2 of the G–tree.

Spatial Range Search. A spatial range search looks for the points P_i with coordinates (x_i, y_i) such that $x_1 \leq x_i \leq x_2$ and $y_1 \leq y_i \leq y_2$, e.g. that are in the query region $R = \{(x_1, y_1), (x_2, y_2))\}$, identified by the coordinates of vertices in the lower left and upper right (Figure 7.5a). The query result is found as follows:

1. The G–tree is searched for the leaf node F_h of the partition containing the lower left vertex (x_1, y_1) of R.
2. The G–tree is searched for the leaf node F_h of the partition containing the upper right vertex (x_2, y_2) of R.
3. For each leaf from F_h to F_k (in the B^+–tree the leaf nodes are sorted) the elements S are searched such that R_S = *RegionOf*(S) overlaps with the query region R. If R_S is fully contained in R, then all points in the associated page satisfy the query, otherwise each point in it must be examined and checked.

Example 7.5
Let us search the points in the region

$$R = \{(35, 20), (45, 60)\}$$

Using the integer encoding of partitions and the partition tree, the result is that the lower left vertex $(30, 20)$ is in the partition 0 and the upper right vertex $(40, 60)$ is in the partition 6. In the G–tree the partitions 0, 4, 5 and 6 are examined in order to check if they overlap with R.

For example, partition 0 with the code "00" corresponds to the region

$$R_S = \{(0, 0), (50, 50)\}$$

that overlaps with R. Therefore the associated page is retrieved to check which of its points are in the query result. Partitions 4 and 5 do not overlap with R, while partition 6 overlaps and the points in the associated page must be examined.

Point Insertion. Let M be the maximum length of the partition codes in the G–tree. The insertion of a point P with coordinates (x, y) proceeds as follows:

1. The G–tree is searched for the leaf node F of the partition R_P that should contain it. Let S_P be the code of R_P.
2. If R_P is not full, insert P, otherwise R_P is split in R_{P_1} and R_{P_2}, with codes $S_{P_1} = S_P$"0" and $S_{P_2} = S_P$"1". If the new strings have a length greater than M, M takes the value $M + 1$, and the integer encoding of the partitions in Figure 7.6c are changed.
3. The points in R_P and P are distributed in R_{P_1} and R_{P_2} (the encoding of each point is calculated as described in Step 1 of the point search algorithm and, if it ends with "0", the point is inserted in R_{P_1}, otherwise in R_{P_2}).
4. The element (S_P, p_{R_P}) in the leaf F is replaced by $(S_{P_1}, p_{R_{P_1}})$ and $(S_{P_2}, p_{R_{P_2}})$. If there is an overflow, the operation proceeds as in B^+–trees.

For example, the insertion of the point $P_1 = (70, 65)$ is in the partition 8 with code "1" and the associated page has space to contain it. Instead, the insertion of the point $P_2 = (8, 65)$ is in the partition 4 with code "0100", which has no space to contain it and a split is required and the result is shown in Figure 7.6, with the new integer encoding of the partitions.

Point Deletion. The deletion of a point P proceeds as follows:

1. Let F be the leaf node with the partition R_P containing P, S_P the partition code of R_P, and S' the partition code of R' obtained from R_P with a split and therefore different from S_P for the last bit only. S' in the G–tree only if it has not yet been split.
2. P is deleted from R_P and then two cases are considered:
   (a) R' *has been split*: if the partition R_P becomes empty, then S_P is deleted from the G–tree, otherwise the operation terminates.
   (b) R' *has not been split*: if the two partition cannot be merged because the number of their points is greater than the page capacity, the operation terminates.

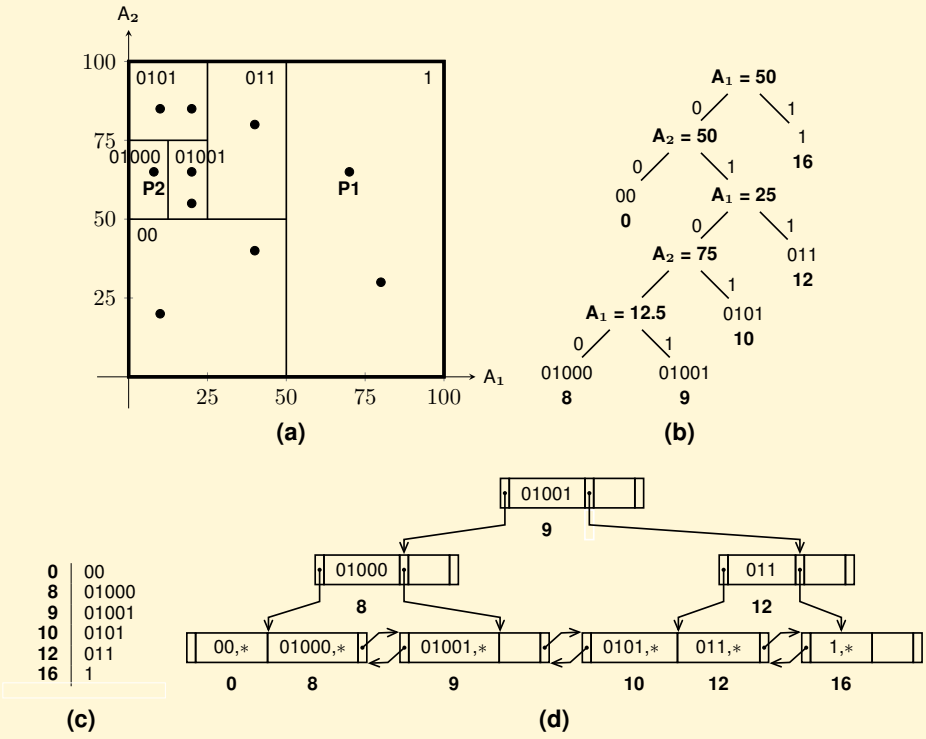


Figure 7.6: Point insertion in a partition to split

Otherwise the two partition are merged, S_P and S' are deleted from the tree, and a new binary string S'', obtained by removing the last bits from S_P, is inserted.

For example, if the points P_1 and P_2 are deleted from the G–tree in Figure 7.6, we return to the situation of Figure 7.5.

7.3 B*–trees *

A B^*–tree, a variant of the **R–tree**, is a dynamic tree structure perfectly balanced as a B^+–tree, for the retrieval of multidimensional data according to its spatial position.

For simplicity, we will consider the two-dimensional case only, and the data consists of *rectangles* described by the coordinates of the bottom left and the top right vertices. For objects of different shapes, the rectangle is the minimum one that contains them.

Terminal nodes of a B^*–tree contain elements of the form (R_i, O_i), where R_i is the rectangular data and O_i is a reference to the data nodes. The nonterminal nodes contain elements of type (R_i, p_i), where p_i is a reference to the root of a subtree, and R_i is the minimum bounding rectangle containing all rectangles associated with the child nodes. For simplicity, in what follows we denote O_i as $*$, and we will call *data region* a rectangular data, and *region* a minimum bounding rectangle.

Let M be the maximum number of items that can be stored in a node and m the minimum number of items allowed in a node, the practice suggests to put $m = 0,4M$. A B^*–tree satisfies the following properties:

– The root node has at least two children unless it is the only node in the tree.
– Each node has a number of elements between m and M unless it is the root node.
– All leaf nodes are at the same level.

Example 7.6
Let us consider the rectangles in Figure 7.7a. The rectangles with the edges in bold are *data regions*, while the others are *regions*.

Figure 7.7b shows a B^*–tree with $M = 3$ and $m = 2$ for the data of Figure 7.7a.

Note that the regions R22 and R23 contain the data region R6, but R23 is the only parent region of R6 (Figure 7.7a). The choice of the parent region will be discussed later.

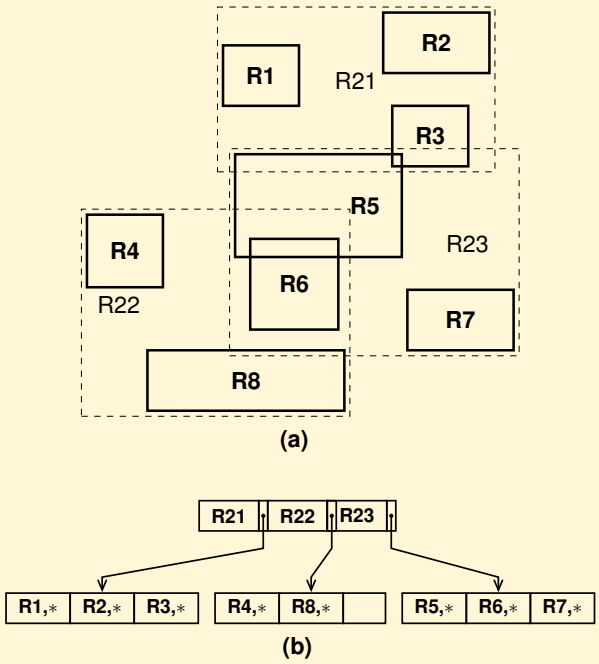


Figure 7.7: An example of R^*–tree

There are some substantial differences between the B^*–tree and B^+–trees:

– The elements of the nodes of a B^+–tree are sorted, while on those of a B^*–tree there is no sort order.
– The regions associated with different elements of the same level may overlap in a B^*–tree, unlike the intervals associated with the elements of a B^+–tree that do not overlap.

We will see later how these differences have an impact on the way we operate on a B^*–tree.

The main operation on $\mathbf{B}^*$–trees, is to find all data regions that overlap to a given rectangular region R.

Search Overlapping Data Regions. The search of the data regions overlapping with R proceeds as follows. The root is visited in order to look for the elements (R_i, p_i) with R_i that overlaps with R. For each element (R_i, p_i) found, the search proceeds in a similar way in the subtrees rooted at p_i. When a leaf node is reached, the data regions R_i in the search result are those with R_i that overlaps with R.

Example 7.7
Let R be the rectangular region with dotted border in Figure 7.8.

During the search of overlapping data regions, in the root is found that R21 and R23 overlap R. Among the descendants of R21, R3 overlaps R, while among the descendants of R23 both R5 and R7 overlap R. Therefore the search result is R3, R5, and R7.

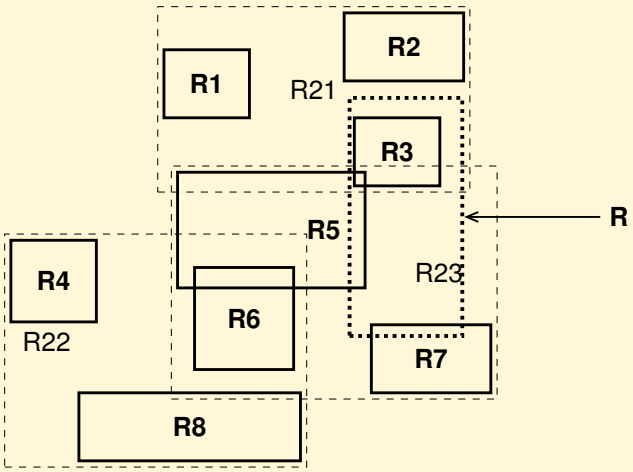


Figure 7.8: An example of search for overlapping data regions

The operations that change the tree, such as insertions and deletions of objects, are more complex and for a detailed description of them see the bibliographic notes.

Insertion. Let S be a new data region to insert in the tree. The operation is similar to inserting a key in a B^+–tree since S is stored into a leaf node, and if there is an overflow, the node will be split into two nodes. In the worst case the division can propagate to the parent node up to the root. But there is a fundamental difference between the operations in the two types of trees.

In B^+–trees there is a single node in which to insert a new element, because the values of the keys in the nodes at the same level partition the domain of the key. In the case of $\mathbf{B}^*$–trees, since the regions in different internal nodes at the same level may overlap, the new data region S may overlaps with more of them, and so it could be inserted in more leaf nodes with a different parent node.

The choice of the region in an internal node can be made according to the degree of overlap with S. For example, one can choose the one that needs the smallest increase in the area to contain S. After having selected the leaf node N where to insert S, if an overflow does not occur, the region is recalculated and its value propagates in the parent node. Otherwise, two cases are considered:

1. *Forced Reinsert*. If this is the first overflow from a leaf node, it is not split; instead p of the $(M+1)$ entries are removed from the node and reinserted in the tree. Ex-

periments suggest that p should be about 30% of the maximal number of entries per node, among those with the center at a greater distance from the center of the region. This way of proceeding can avoid splitting the node, because the data regions are placed in other nodes, and constitutes a form of dynamic reorganization of the tree.

2. *Node Splitting*. After the first overflow, the $(M+1)$ elements are divided between two nodes, and two new elements are inserted in the parent node, and the process goes on to propagate the effects. The problem that arises is how to divide the elements of a node because different criteria can be used. In $\mathbf{B}^*$–trees, the subdivision that produces regions with minimum total perimeter is selected.

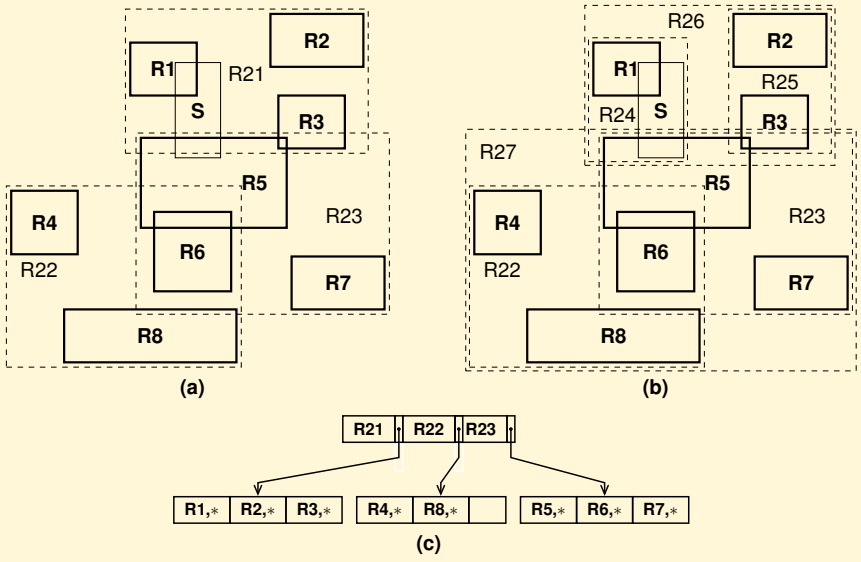
Example 7.8

Let us consider the Figure 7.7a, and assume to insert the data region S (Figure 7.9a). The process starts with the root node of the B^*–tree in Figure 7.9c. Let the region R21 be selected for the insertion.

Following the associated pointer, the leaf node to the left is considered, and since this node does not have enough space to contain S an overflow occurs. Being the first, we proceed with the reinsertion of R1. The region R21 is updated (not shown in the figure) and the reinsertion takes place in the same leaf node, causing another overflow and then a subdivision. Suppose that we get {R1, S} and {R2, R3}.

Let R24 and R25 be the regions containing (R1, S) and (R2, R3) (Figure 7.9b). Consequently, two new elements have to be inserted into the root node to replace R21 (Figure 7.9c).

Since in the root node there is not enough space to contain four elements, there is an overflow. In the root the reinsertion it is not applied, but a subdivision is made. The result is that the old root node is replaced by two new nodes, one containing (R24, R25) and the other (R22, R23). Let R26 be the region containing (R24, R25) and R27 be the region containing (R22, R23). A new root is added with elements R26 and R27 (Figure 7.9d).



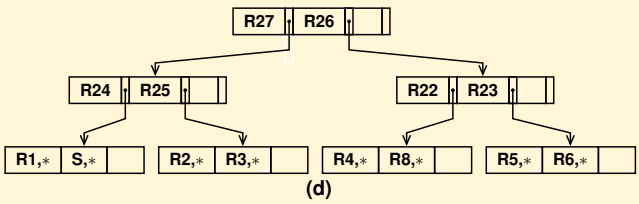


Figure 7.9: Example of a rectangle insertion

Deletion. To delete a data region from a B^*–tree, if the leaf node does not become underfull, it is checked whether the region to which it belongs can be reduced in size. If so, the adjustment is propagated upwards. Otherwise, instead of proceeding as in B^+–trees, it is preferred to remove the tree node, update the parent nodes and reinsert all its elements.

7.4 Summary

1. Multidimensional data is used to represent geometric objects such as points, segments, rectangles, polygons, and their position in a multidimensional space. They require appropriate storage structures to allow searches for points, intervals, for the nearest neighbor and partial match queries. Many solutions have been proposed and in this chapter we discussed only two of them: G–trees and $\mathbf{B}^*$–trees.
2. Multidimensional data is encountered in relational DBMSs, for a multi-attribute search by interpreting the tuples of a relation as points in a space with attributes that correspond to the dimensions, and in systems for text or image retrieval described by a set of features.
3. Structures for multidimensional data are usually based on the idea of recursively partitioning the space into regions of decreasing size that contain at most the amount of data stored in a page. To facilitate search operations, information on the regions are organized using various types of trees, where the nodes represent regions and their children a partition of the region into smaller ones.
4. G–trees are an example of balanced trees to represent points of a k-dimensional space. They combine the ideas of data space partitioning and of B^+–trees: the data space is partitioned into regions of varying dimensions identified by an appropriate code on which there is a total ordering and the set of codes is organized as a B^+–tree.
5. The $\mathbf{B}^*$–trees are the most popular structure used by commercial DBMS extensions dedicated to multidimensional data.

Bibliographic Notes

Numerous solutions have been proposed to treat multidimensional data. For an extensive review, which includes other structures not considered in this chapter, see [Gaede and Günther, 1998] and the book [Samet, 1990].

The G–tree is presented in [Kumar, 1994], the **R–tree** was proposet in [Guttman, 1984] and B^*–tree in [Beckmann et al., 1990]. For a survey of all these structures see also [Yu and Meng, 1998].

Other interesting solutions for multidimensional data are the ***grid file*** [Nievergelt et al., 1984] and **UB–tree** [Ramsak et al., 2000].

Exercises

Exercise 7.1 Answer the following questions briefly:

- Explain the difference between a multi-attribute index stored in a B^+–tree and a multidimensional organization.
- Describe some examples of spatial queries.
- Describe some examples of nearest neighbor queries.

Exercise 7.2 Show the G–tree in Figure 7.5 after inserting the point $(30, 30)$.

Exercise 7.3 Show the G–tree in Figure 7.5 after the deletion of the point $(80, 30)$.

Exercise 7.4 Consider the B^*–tree shown in Figure 7.9d, and answer the following questions.

1. Give an example of search of a data region that requires a visit of both the R26 and R27 subtrees.
2. Give an example of a data region contained in both R26 and R27, and is inserted into R27.

Chapter 8

ACCESS METHODS MANAGEMENT

Once the managers of the permanent memory, buffer and storage structures have been implemented, with the *Access Methods Manager* there is a first version of a relational *Storage Engine*, although it lacks transactions and concurrency. The *Access Methods Manager* provides to the *Relational Engine* the operators used by its modules to execute the commands for the definition and use of databases.

8.1 The Storage Engine

As a programming language transforms a computer into an *abstract machine* whose characteristics and functionalities are mainly determined by those of the programming language, so will a language to define and use databases transforms a computer, and in particular its file management system, into an *abstract database machine*, called the *database management system*, whose characteristics and functionalities will depend mainly on those of the adopted data model.

An abstract database machine is normally divided into two parts: an *abstract machine for the logical data model*, called the ***Relational Engine***, and an *abstract machine for the physical data model*, called the ***Storage Engine***.

The ***Relational Engine*** includes modules to support the execution of SQL commands, by interacting with the ***Storage Engine***, which includes modules to execute the operations on data stored in the permanent memory.

Normally the DBMS storage engine is not accessible to the user, who will interact with the relational engine. An example of a system in which this structure is clearly shown is *System R*, a relational DBMS prototype developed at the IBM scientific center in San Josè, from which DB2 was then produced. This system has a relational engine called a *Relational Data System* (RDS) and a storage engine called a *Relational Storage System* (RSS).

While the interface of the relational engine depends on the data model features, the interface of the storage engine depends on the data structures used in permanent memory. Although in principle it is possible to isolate a set of data structures to define a storage engine suitable to the functionality of a engine for any of the data models, in systems in use this does not occur because each of them adopt solutions for a specific data model.

To give a better idea of the interface of a storage engine, we will consider the case

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

- Operators to create databases.
- Operators to start and to end a transaction:
  - *beginTransaction*: *null* → *TransId*
    to specify the beginning of a transaction. The operator returns the system generated transaction identifier.
  - *commit*: *TransId* → *null*
    to specify the successful termination of a transaction.
  - *abort*: *TransId* → *null*
    to specify the unsuccessful termination of a transaction with the request to abort it.
- Operators on heap files and indexes.
- Operators about *access methods* available for each relation. Access methods are ways of retrieving records from a table and consist of either a heap file scan or a direct access using their RID obtained by an index scan with a search condition.

These operators are used by the relational engine to implement the system functionality. For example, the query optimizer translates a SQL query into a *physical query plan*, in which each node is the algorithm that uses the storage engine operators to evaluate the corresponding relational algebra operator.

Transactions and physical query plans will be discussed in forthcoming chapters.

8.2 Operators on Databases

The JRS storage engine stores a database, files and indexes, in a folder with the following operators:

- *createDB*: *Path* × *BdName* × *TransId* → *null*
  creates a database in the path specified.
- *createHeapFile*: *Path* × *BdName* × *heapFileName* × *TransId* → *null*
  creates a heap file in the database in the path specified.
- *createIndex*: *Path* × *BdName* × *IndexName* × *heapFileName* × *Attribute* × *Ord* × *Unique* × *TransId* → *null*
  creates an index on a relation attribute, a key if Unique = true. The index is sorted by default ascending, if *Ord* is not specified as desc. An index can be built on multiple attributes, and several indexes can be defined on a relation.

A database, an index or a heap file are deleted with the operators:

- *dropDB*: *Path* × *BdName* × *TransId* → *null*
- *dropIndex*: *Path* × *BdName* × *IndexName* × *TransId* → *null*
- *dropHeapFile*: *Path* × *BdName* × *heapFileName* × *TransId* → *null*

8.3 Operators on Heap Files

A database table is stored in a heap file on which there are operators to insert, delete, retrieve or update records with a specified RID, or to get the number of pages used and the number of the records.

- *HF_insertRecord*: *Record* → *RID*
- *HF_deleteRecord*: *RID* → *null*
- *HF_getRecord*: *RID* → *Record*
- *HF_updateRecord*: *RID* × *FieldNum* × *NewField* → *null*
- *HF_getNPage*: *null* → *int*
  HF_getNRec: *null* → *int*

A table is a set of records where each record contains the same number of fields and can have a variable-length. As we will see later, a heap file supports a *scan* operation to step through all the records in the file one at a time.

8.4 Operators on Indexes

An index is a set of records of type Entry organized as B^+–tree. An Entry has two attributes Value and RID. The Value is the search key and the RID is the identifier of the record with the search key value.

The operators available on indexes are those to insert or delete elements, or to get data about the B^+–tree used to store them, such as the number of leaves, minimum and maximum search key Value.

- *I_insertEntry*: *Value* × *RID* → *null*
- *I_deleteEntry*: *Value* × *RID* → *null*
- *I_getNkey*: *null* → *int*
  I_getNleaf: *null* → *int*
- *I_getMin*: *null* → *Value*
  I_getMax: *null* → *Value*

An index provides a way to efficiently retrieve all records that satisfy a condition on the search key, through operators that we will see later.

8.5 Access Method Operators

The ***Access Methods Manager*** provides the operators to transfer data between permanent memory and main memory in order to answer a query on a database. Permanent data are organized as collections of records, stored in heap files, and indexes are optional auxiliary data structures associated with a collection of records. An index consists of a collection of records of the form (key value, RID), where *key value* is a value for the search key of the index, and RID is the identifier of a record in the relation being indexed. Any number of indexes can be defined on a relation, and a search key can be multi-attribute. The operators provided by the *Access Methods Manager* are used to implement the operators of physical query plans generated by the query optimizer.

Records of a heap file or of an index are accessed by scans. A heap file scan operator simply reads each record one after the other, while an index scan provides a way to efficiently retrieve the RID of a heap file records with a search by key values in a given range.

The records of a heap file can be retrieved by a serial scan or directly using their RID obtained by an index scan with a search condition.

A heap file or index scan operation is implemented as an *iterator*, also called *cursor*, which is an object with methods that allow a consumer of the operation to get the result one record at a time. The iterator is created with the function open, and has the

methods (a) isDone, to know if there are records to returns, (b) getCurrent, to return a record, (c) next, to find the next record, and (d) close to end the operations.

When a heap file iterator is created, it is possible to specify the RID of the first record to return. Instead, when an index iterator is created a key range is specified.

Once an iterator C is opened, a scan is made with the following type of program:

```
while not C.isDone() {
    Value := C.getCurrent();
    ...;
    C.next()
};
```

Heap File Scan Operators.

HFS_open: *HeapFile* $\rightarrow$ *ScanHeapFile*
HFS_open: *HeapFile* $\times$ *RID* $\rightarrow$ *ScanHeapFile*

The *ScanHeapFile* iterator methods are:

- *HFS_isDone*: *null* $\rightarrow$ *boolean*
- *HFS_next*: *null* $\rightarrow$ *null*
- *HFS_getCurrent*: *null* $\rightarrow$ *RID*
- *HFS_reset*: *null* $\rightarrow$ *null*
- *HFS_close*: *null* $\rightarrow$ *null*

Index Scan Operators.

IS_open: *Index* $\times$ *FirstValue* $\times$ *LastValue* $\rightarrow$ *ScanIndex*

The *ScanIndex* iterator methods are::

- *IS_isDone*: *null* $\rightarrow$ *boolean*
- *IS_next*: *null* $\rightarrow$ *null*
- *IS_getCurrent*: *null* $\rightarrow$ *Entry*
- *IS_reset*: *null* $\rightarrow$ *null*
- *IS_close*: *null* $\rightarrow$ *null*

8.6 Examples of Query Execution Plans

Let us see some examples of programs that use access methods of the storage engine to execute simple SQL queries. The programs show the nature of possible query execution plans that *might* be generated by the query optimizer of the relational engine. However, as we will see in Chapters 11 and 12, the query optimizer does not translate a SQL query in a query execution plan of this type, but in a *physical plan*, an algorithm to execute a query given as a tree of *physical operators*. These operators use those of the access methods to implement a particular algorithm to execute, or to contribute to the execution, of a relational algebra operator.

Example 8.1
Let us consider the relation Students with attributes Name, StudentNo, Address and City, and the query:

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

Assuming the relation is stored in a file with the same name, the structure of a possible program to execute the query is:

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

Example 8.2
Assuming that on Students there is an index IdxCity on the attribute City. A more efficient program to retrieve the Pisa students' name is:

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

8.7 Summary

1. A DBMS Storage Engine provides a set of operators used by the Query Manager to generate a query execution plan. The main abstractions provided are: creation of databases, heap files, indexes, and to start and to end transactions.
2. The operators available on heap files are those to retrieve records using their RID.
3. The operators available on indexes are those to retrieve their elements using a search key value.
4. An access method is a way for retrieving records from a table and consists of either a file scan (i.e. a complete retrieval of all records) or an index scan with a matching selection condition. An access method is implemented as an *iterator*, which is an object with methods that allow a consumer of the operation to get the result one record at a time.

Bibliographic Notes

For examples of more detailed descriptions of the storage engine interface of relational systems, see the documentation of systems MINIBASE [Ramakrishnan and Gehrke, 2003], WiSS [Chou et al., 1985], and the book [Gray and Reuter, 1997].

Chapter 9

TRANSACTION AND RECOVERY MANAGEMENT

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

9.1 Transactions

A database is a model of some aspect of the world to serve an explicit purpose in providing information about that aspect of the world being modeled. In general, there are rules defined in the database schema, called ***static integrity constraints***, that must be satisfied by all database states, and others, called ***dynamic integrity constraints*** that restrict allowable state transitions.

Definition 9.1 *Consistent State*

A *consistent state* of the database is a state in which all integrity constraints are satisfied.

When an event occurs in the real world that changes the modeled reality state, to ensure that the database state changes in a corresponding way, a mechanism to properly group operations on the database into atomic units of work, called *transactions* is necessary.

We assume that a transaction is *correct*, that is, it causes the database to change from one consistent state to another consistent state when performed alone. The DBMS must guarantee that this property holds even if there is a system failure and when the transaction is performed concurrently with other transactions, by avoiding interferences that affect their correctness.

The following example shows why the transactions mechanism is necessary to protect a database from failures.

Example 9.1
Consider an application program to manage flight reservations for an airline company. To reserve a flight from the city C_1 to C_3 with a connecting flight at C_2, two different reservations must be made: one for the flight from C_1 to C_2, and another for the flight from C_2 to C_3. Let us assume that one of the integrity constraints is that the sum of the available and reserved seats on a flight does not exceed the number of seats on the flight. Therefore, the following actions should be undertaken:

1. Decrease by one the available seats on the flight from C_1 to C_2.
2. Increase by one the reserved seats on the flight from C_1 to C_2.
3. Decrease by one the available seats on the flight from C_2 to C_3.
4. Increase by one the reserved seats on the flight from C_2 to C_3.

In the time interval between steps 2 and 3, although the database is in a consistent state, if the application is interrupted due to some kind of failure, the state of the database would be incorrect, since such a state would not be in accordance with the user's intentions. In fact the application should reserve both flights, or none of them, to model a correct situation.

Let us present the concept of transaction both from the programmer's point of view and from the system's point of view.

9.1.1 Transactions from the Programmer's Point of View

A DBMS provides a programming abstraction called a ***transaction***, which groups together a set of instructions that read and write data. In this context the term transaction is used instead for the *execution* of a user program. Different executions of the same program produce different transactions.

Definition 9.2 A transaction is a sequence of operations on the database and on temporary data, with the following properties:

Atomicity Only transactions terminated normally (*committed transactions*) change the database; if a transaction execution is interrupted because of a failure (*aborted transaction*), the state of the database must remain unchanged as if no operation of the interrupted transaction had occurred.

Isolation When a transaction is executed concurrently with others, the final effect must be the same as if it was executed alone.

Durability The effects of committed transactions on the database survive system and media failures, i.e. commitment is an irrevocable act.

The acronym ACID is frequently used to refer to the following four properties of transactions: *Atomicity*, *Consistency*, *Isolation*, and *Durability*. Among these properties, atomicity, durability and isolation are provided by a DBMS. Consistency cannot be ensured by the system when the integrity constraints are not declared in the schema. However, assuming that each transaction program maintains the consistency of the database, the concurrent execution of the transactions by a DBMS also maintain consistency due to the isolation property.

The isolation property is sometime called the ***serializability*** property: when a transaction is executed concurrently with others, the final effect must be the same as a *serial* execution of committed transactions, i.e. the DBMS behaves as if it executes the transactions one at a time.

The DBMS module that guarantees the properties of atomicity and durability, in order to protect the database from different kinds of failures, is called the ***Transaction and Recovery Manager***. The isolation property, on the other hand, is guaranteed by the ***Concurrency Manager***, which will be discussed in the next chapter.

When studying transactions, we abstract from the specific language in which they are written. The same considerations apply when a programming language with database operators is used, as well as when a transaction is a set of SQL statements terminated by a COMMIT or ROLLBACK command.

Example 9.2
Suppose we have a database with two relations

CheckingAccounts(Number, Balance)
SavingAccounts(Number, Balance)

Figure 9.1 shows a program to transfer a money amount from a saving account to a checking account. The ROLLBACK command signals to the DBMS that the transaction must be undone, while the COMMIT command signals to the DBMS to finalize the changes.

9.1.2 Transactions from the DBMS's Point of View

Although a transaction performs many operations on the data retrieved by the database, a DBMS only "sees" the read and write operations on its data. A write operation updates a page in the buffer, but does not cause an immediate transfer of the page to the permanent memory, as we will show later. For this reason, if for some kind of failure the content of the buffer is lost, the updates might not have been written to the database. To correct this situation, the DBMS will have to take special preventive measures.

Definition 9.3

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

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

The transaction operations are not necessarily part of the user's code: *beginTransaction* and *commit* could be issued automatically at the beginning and the end of the program, while *abort* could be automatically generated by the concurrency manager.

```
program MoneyTransfer;
var
exec sql begin declare section
    xAmount, Amount, xBalance: integer;
    Number, FromAccount, ToAccount: array [1..6] of char;
exec sql end declare section

begin
exec sql connect "UserId" identified by "Password";
{Input data is read}
writeln('Write Amount, Withdrawals Account, Deposit Account');
read(Amount, FromAccount, ToAccount);
exec sql
    select Balance   into :xBalance
    from SavingAccounts
    where Number= :FromAccount;
if xBalance < Amount
    then
        begin
        writeln('Insufficient Balance'); rollback;
        end
    else
        begin
            exec sql
                update SavingAccounts
                set Balance = :xBalance – :Amount
                where Number = :FromAccount;
            exec sql
                update CheckingAccounts
                set Balance = :xBalance + :Amount
                where Number = :ToAccount;
        end;
if sqlcode = 0  then commit  else rollback
end;
end {program}.
```

Figure 9.1: An example of transaction

The execution of the *commit* operation does not guarantee the successful termination of the transaction, because it is possible that the transaction updates cannot be written to the permanent memory, and therefore it will be aborted, as we will see in the following. Figure 9.2 shows the different states of a transaction execution.

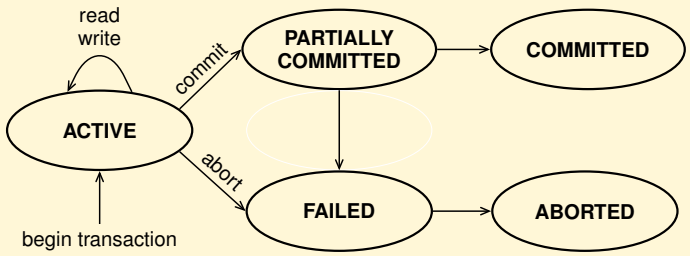


Figure 9.2: State transition diagram for transaction execution

Definition 9.4 *Transaction State*

- A transaction enters into the *active* state immediately after it starts execution, where it stays while it is executing.
- A transaction moves to the *partially committed* state when it ends.
- A transaction moves to the *committed* state if it has been processed successfully and all its updates on the database have been made durable.
- A transaction moves to the *failed* state if it cannot be committed or it has been interrupted after a transaction failure while in the *active* state.
- A transaction moves to the *aborted* state if it has been interrupted and all its updates on the database have been undone.

A transaction is said to have *terminated* if it has either committed or aborted.

9.2 Types of Failures

A centralized database can become inconsistent because of the following types of failures: *transaction failure*, *system failure* or *media failure*. We assume that the occurrence of a failure is always detected, and this causes:

- the immediate interruption of a transaction or of the whole system, depending on the type of failure;
- the execution of specific *recovery procedures* to ensure that the database only contains the updates produced by committed transactions.

Definition 9.5 *Transaction Failure*

A *transaction failure* is an interruption of a transaction which does not damage the content of either the buffer or the permanent memory.

A transaction can be interrupted (i.e., it can fail) because (a) the program has been coded in such a way that if certain conditions are detected then an abort must be issued, (b) the DBMS detects a violation by the transaction of some integrity constraint or access right, or (c) the concurrency manager decided to abort the transaction since it was involved in a deadlock.

Definition 9.6 *System Failure*

A *system failure* is an interruption (*crash*) of the system (either the DBMS or the OS) in which the content of the buffer is lost, but the content of the permanent memory remains intact.

When a system crash occurs, the DBMS is restarted, automatically or by an operator.

Definition 9.7 *Media Failure*

A *media failure* (*distaster* or *catastrophe*), is an interruption of the DBMS in which the content of the permanent memory is corrupted or lost.

When a media failure occurs, the recovery manager uses a backup to restore the database.

9.3 Database System Model

A view of the components of a centralized DBMS, already presented in general terms, is shown again in Figure 9.3 with an emphasis on some aspects specific to transaction management.

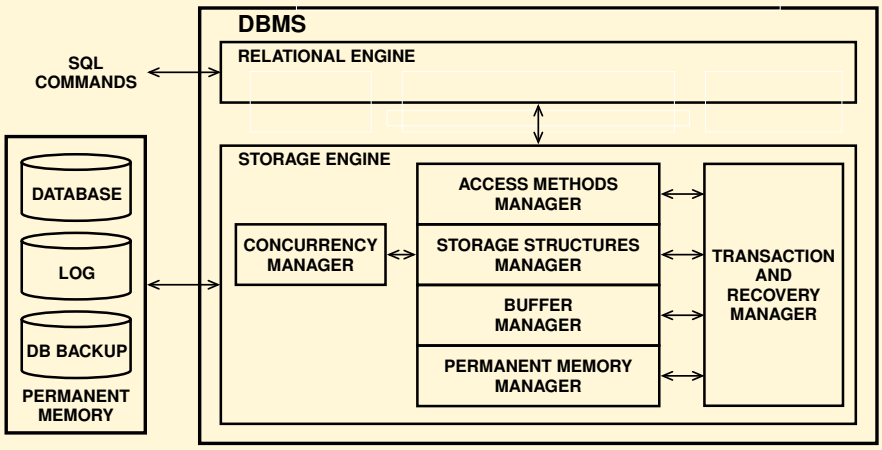


Figure 9.3: Model of a DBMS

To guarantee the atomicity and durability properties of transactions, the permanent memory consists of three main components: the database, and a set of auxiliary data (*Log* and *DB Backup*) used by the recovery procedure in the case of failures. The database, log and backup are usually stored on distinct physical devices. Moreover, since the log files pages are managed with a different algorithm from that used for the database pages, the systems use different buffers for these two types of pages.

To simplify the description of the system, we will suppose that transactions update pages, and not records, which are usually smaller than pages, as it happen in real situations.

The *Transaction and Recovery Manager* performs the following tasks: (a) the execution of a read, write, commit and abort operation on behalf of transactions; (b) the management of the log; (c) the execution of a *restart* command after a system failure, that guarantees that the database only contains the updates of the successfully terminated transactions.

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

1. The database is just a set of pages.
2. Each update operation affects a single page.
3. The operation of transferring a page from the buffer to the permanent memory is an atomic operation.
4. If different transactions are concurrently in execution, they read and write different pages. We will discuss in the next chapter the case of conflicts between transactions accessing the same pages.

Next section presents data protection, then Section 9.5 presents classes of recovery algorithms, Section 9.6 presents the recovery manager operators to deal with transaction and system failures, and Section 9.7 presents the recovery manager algorithms for system and media failures.

9.4 Data Protection

We have already discussed how the database can be in an incorrect state due to a transaction, system or media failure. The different techniques that are used in these situations share the common principle of redundancy: to protect a database the DBMS maintains some redundant information during normal execution of transactions so that in the case of a failure it can reconstruct the most recent database state before the occurrence of the failure.

Definition 9.8 *Recovery*

The recovery is the process that restores the database to the consistent state that existed before the failure.

We now discuss the main additional information that make possible the recovery of a database.

Database Backup. DBMSs provide facilities for periodically making a backup copy of the database (*database dump*) onto some form of tertiary storage.

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

For each transaction T_i, the following information is written to the log:

- When the transaction starts, the record (*begin*, T_i).
- When the transaction commits, the record (*commit*, T_i).
- When the transaction aborts, the record (*abort*, T_i).
- When the transaction modifies the page P_j the record (W, T_i, P_j, BI, AI), where BI is the old value of the page (*before image*) and AI is the new version of the page (*after image*)[1].

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

Figure 9.4: The operations of a transaction and the corresponding records in the log

Each log record is identified through the so called LSN (*Log Sequence Number*), that is assigned in a strictly increasing order. A LSN could be, for instance, a serial number of the position of the first character of the record in the file.

Figure 9.4 shows a simple example of the information written to the log by the operations of a transaction, assuming for simplicity that A and B are pages containing just an integer value.

The exact content of the log depends on the algorithms of the transaction manager, which we will discuss later. For instance, in some cases it is not necessary to write both the before and the after image of a modified page.

In general a log is stored in a file buffered for efficiency reasons. The log has pages on its own which are only written to the permanent memory when they become full in the buffer, and not when a single record is written to the page. For this reason, the transaction manager sometime forces the writing of a log page to the permanent memory, to prevent data loss in the case of system failure. For simplicity, we assume for now that *the log is not buffered and each record is immediately written to the permanent memory*.

Undo and Redo Algorithms. A database update changes a page in the buffer, and only after some time the page may be written back to the permanent memory. Recovery algorithms differ in the time when the system transfers the pages updated by a transaction to the permanent memory.

We say that a recovery algorithm requires an *undo* if an update of some uncommitted transaction is stored in the database. Should a transaction or a system failure occur, the recovery algorithm must *undo* the updates by copying the before-image of the page from the log to the database (Figure 9.5a).

We say that a recovery algorithm requires *redo* if a transaction is committed before all of its updates are stored in the database. Should a system failure occur after the transaction commits but before the updates are stored in the database, the recovery algorithm must *redo* the updates by copying the after-image of the page from the log to the database (Figure 9.5b).

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

1. In real systems the records written to the log are more compact than the new or old version of the page.

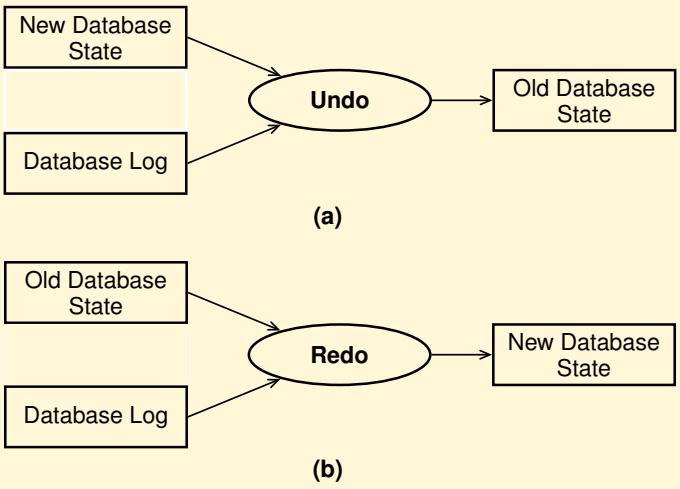


Figure 9.5: Undo and Redo algorithms

Checkpoint. To reduce the work performed by a recovery procedure in the case of system failure, another information is written to the log, the so called ***checkpoint*** (*CKP*) event.
In the literature different methods of performing and recording checkpoints have been discussed. Here we show the two methods most commonly used.

The first one is based on the hypothesis that a checkpoint should mark a state in which the log is completely aligned with a correct state of the database (*commit-consistent checkpoint*). In this case, after either a constant period of time, or a constant number of records written to the log, the system performs the following steps.

Definition 9.9 *Commit-consistent checkpoint*

When the checkpoint procedure starts, the following actions are performed:

1. The activation of new transactions is suspended.
2. The systems waits for the completion of all active transactions.
3. All pages present in the buffer which have been modified are written to the permanent memory and the relevant records are written to the log (*flush operation*).
4. The *CKP* record is written to the log.
5. A pointer to the CKP record is stored in a special file, called ***restart file***.
6. The system allows the activation of new transactions.

In the third step, the transferring of the modified pages to the permanent memory is forced so that all the transactions terminated before the checkpoint have their updates made durable in the database, and must not be redo in the case of system failures.

This strategy is simple to implement but not efficient, because of the steps (1) and (2). This discourages a frequent execution of the checkpoint procedure, and consequently increases the work that must be done in the case of restart. The problem of finding the optimal checkpoint interval can be solved with a balance between the cost of the checkpoint and that of the recovery, but this is a "difficult" problem, and usually the following, more complex but more efficient checkpoint method is preferred, called ***buffer-consistent checkpoint – Version 1***.

Definition 9.10 *Buffer-consistent checkpoint – Version 1*

When the checkpoint procedure starts, the following actions are performed:

1. The activation of new transactions is suspended.
2. The execution of database operation of the active transactions is suspended.
3. All pages present in the buffer which have been modified are written to the permanent memory and the relevant records are written to the log (*flush operation*).
4. The *CKP* record, containing the list of the identifiers of the active transactions, is written to the log.
5. A pointer to the CKP record is stored in a special file, called ***restart file***.
6. The system allows the activation of new transactions and continues the execution of the active transactions.

This method does not need the waiting for the termination of active transactions. However, it still presents the problem of the buffer flush operation, which, in the case of a big buffer, is a costly operation. For this reason other methods have been studied to reduce the checkpoint execution time. One of this is the ARIES algorithm, that avoids (a) the suspension of the activation of new transactions and the execution of active transactions, and (b) the buffer flush operation.

9.5 Recovery Algorithms

The *Recovery managers* for the transactions management differ in the way they combine the *undo* and *redo* algorithms to recover the last consistent state of a database from a system failure. *Undo* and *redo* algorithms can be combined in four different ways and each combination defines a new recovery algorithm:

Undo-Redo	requires both *undo* and *redo*.
Undo-NoRedo	requires *undo* but not *redo*.
NoUndo-Redo	requires *redo* but not *undo*.
NoUndo-NoRedo	requires neither *undo* nor *redo*.

9.5.1 Use of the Undo Algorithm

The use of the *undo* algorithm depends on the policy used to write pages updated by an *active transaction* to the database. There are two possibilities:

1. *Deferred update*
2. *Immediate update*.

These two policies are called ***NoUndo-Undo*** (*Fix-NoFix*, *NoSteal-Steal*).

Definition 9.11

The *deferred update* policy requires that updated pages cannot be written to the database before the transaction has committed.

To avoid that pages updated by a transaction are written to the database, they are "pinned" in the buffer until the end of the transaction.

An algorithm for the execution of the transaction with the *deferred update* policy is of the *NoUndo* type: when a transaction or system failure occurs it is not necessary to undo its updates since the database has not been changed.

Definition 9.12

The *immediate update* policy allows that updated pages can be written to the database before the transaction has committed.

To allow the buffer manager to write an updated page to the database before the transaction has committed, the page is marked as "dirty" and its pin is removed.

An algorithm that adopts the immediate update policy is certainly of the *Undo* type: if a transaction or system failure occurs the updates on the database must be undone.

To undo the updates of a transaction the following rule must be observed.

Definition 9.13 *Undo Rule* (*Write Ahead Log, WAL*)

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

This rule enables a transaction to be undone in the case of abort by using the before-images from the log.

Because of the hypothesis that a write to the log is immediately transferred to the permanent memory, to guarantee such rule it is sufficient to "unpin" the updated page after the write to the log.

9.5.2 Use of the Redo Algorithm

The use of the *redo* algorithm depends on the policy used to *commit* a transaction. There are two possibilities:

1. *Deferred commit*.
2. *Immediate commit*.

These two policies are called ***NoRedo-Redo*** (*NoFlush-Flush*, *NoForce-Force*).

Definition 9.14

The *deferred commit* policy requires that all updated pages are written to the database before the *commit* record has been written to the log.

A transaction that implements the *deferred commit* policy is of the *NoRedo* type, since it is not necessary to redo all the updates on the database in the case of failure. With this solution, however, the buffer manager is forced to transfer to the permanent memory all the pages updated by the transaction before the *commit* operation.

Definition 9.15

The *immediate commit* policy allows the *commit* record to be written to the log before all updated pages have been written to the database.

An algorithm that implements the *immediate commit* policy is of the *Redo* type, since it is necessary to redo all the updates in the case of a system failure. On the other hand the buffer manager is free to transfer the unpinned pages to the permanent memory when it considers appropriate.

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

Definition 9.16 *Redo Rule* or *Commit Rule*

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

If this rule is not followed, a system failure shortly after the transaction commit will lose the last after-images, making it impossible to redo the transaction.

9.5.3 No use of the Undo and Redo Algorithms

A *NoUndo* algorithm requires that all the updates of a transaction *must be* in the database *after* the transaction has committed. Conversely a *NoRedo* algorithm requires that all the updates of a transaction *must be* in the database *before* the transaction has committed. Therefore, a *NoUndo-NoRedo* algorithm requires that all the updates of a transaction *must be* in the database neither *before* nor *after* the transaction has committed. Thus the only possibility is that the commit operation *atomically* writes all the updates of a transaction to the database and mark the transaction as committed. This is feasible using the so called ***shadow pages*** technique proposed in [Lorie, 1977]. The basic idea is shown in Figure 9.6.

The implementation is based on a permanent memory index that maps each page identifier to the physical address where the page is stored (*Page Table*).

Moreover, there exists at a fixed address of the permanent memory the *database descriptor*, a record that contains a pointer to the *Page Table* (Figure 9.6a).

When a transaction starts, a copy of the *Page Table* is created in the permanent memory (*New Page Table*), and it is used by the transaction.

When a transaction updates for the first time a page, for instance page 1, the following actions happen (Figure 9.6b):

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

All the subsequent read and write operations on that page performed by the transaction will operate on the current page.

When the transaction reaches the *commit* point, the system should substitute all the shadow pages with an atomic action, otherwise if a failure would happen during the execution of the commit operation, the database would be left in an incorrect state, since with this approach there is no log to record the information needed for the recovery. This atomic action is implemented by executing the following steps:

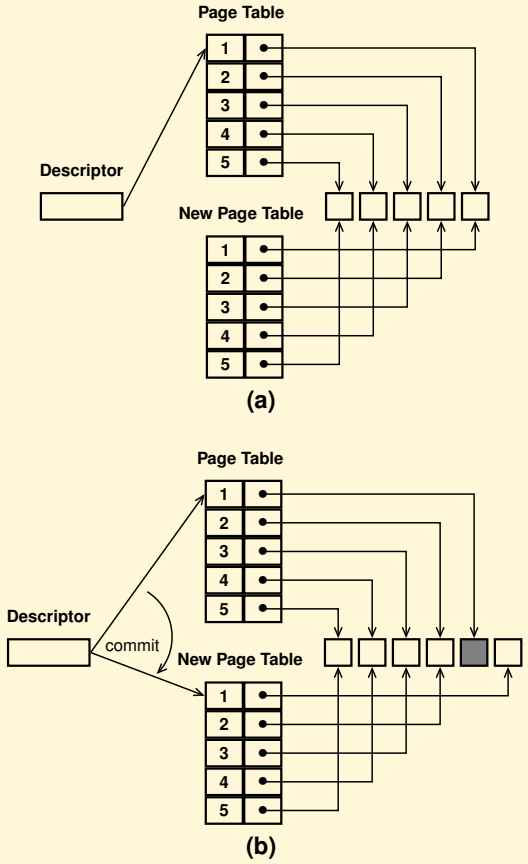


Figure 9.6: The shadow pages technique

1. The pages updated in the buffer are written to the permanent memory, while the *Page Table* (that references the shadow pages) is unchanged.
2. The descriptor of the database is updated with an atomic operation, by replacing the pointer to the *Page Table* with that to the *New Page Table*, that becomes the new *Page Table*.

In the case of failures, the *New Page Table* is ignored and the database descriptor continues to reference the initial *Page Table*, which is unchanged.

This technique has several shortcomings:

– The pages occupied by records of the same table are scattered in the permanent memory (storage fragmentation).
– There is the need of a *garbage collection* algorithm to put the shadow pages back in the list of free pages.
– Some kind of log must still be maintained to deal with disasters.
– This method becomes complex if the system manages concurrent transactions.

9.6 Recovery Manager Operations

In this section three simplified implementations of the following operators of the *Recovery Manager* are presented. They are executed atomically and differ for the policy used to deal with the page updates and the termination of transactions.

beginTransaction ( )	marks the beginning of a transaction and returns its identifier T, automatically generated.
read (T, P)	reads the page P into the buffer for transaction T.
write (T, P, V)	updates the page P in the buffer with the new version V for transaction T.
commit (T)	for the successful termination of transaction T.
abort (T)	for the unsuccessful termination of transaction T.
restart ( )	bring the database in its last consistent state after a system failure.

To implement these operations, the following operators of the buffer manager and log manager will be used:

– Buffer manager

getAndPinPage (P)	reads the page P from permanent memory into buffer, if is not already present, pins it, and returns the address of the buffer page.
unpinPage (P)	removes the pin from the page P.
updatePage (P, V)	modifies the page P with V and sets its dirty bit.
flushPage (P)	forces the transfer of page P to the permanent memory, if dirty.
setNoDirty (P)	sets page P in the buffer as not dirty.
undoPage (P, V)	reads the page P from permanent memory into buffer, if not already present, modifies it with V, removes the pin and sets its dirty bit.

– Log manager

append (LogRec)	appends at the end of log a record of one of the following types:[2] (*begin*, T), (*commit*, T), (*abort*, T) and, for update operations, (W, T, P, BI, AI), (W, T, P, BI) or (W, T, P, AI), with BI and AI the values of the *before* and *after image* of P.

For simplicity, we ignore the **read** (T, P) operation, identical in all the recovery algorithms: the operation calls **getAndPinPage** (P), and when it has done reading the page, it calls **unpinPage** (P).

9.6.1 Undo-NoRedo Algorithm

In this class of algorithms, redoing a transaction to recover from a system failure is never necessary. Thus only the *before-image* must be written to the log, to deal with transaction and system failures.

A database page is updated before the *commit* record is written to the log (Figure 9.7).

2. The first value is a string that represents the type of operation.

The operation *restart* is efficient, but the execution of transactions is hindered by the fact that at the *commit* point all the pages updated must be written to the database. Moreover, transaction failures require the undo of its updates, so this type of algorithm is generally used with pessimistic concurrency control algorithm that rarely aborts transactions.

```
beginTransaction()
   T := newTransactionIde();
   Log.append(begin, T);
   return(T).
write(T, P, V)
   Buffer.getAndPinPage(P);
   BI := page P;
   Log.append(W, T, P, BI);
   Buffer.updatePage(P, V);
   Buffer.unpinPage(P).
commit(T)
   for each P ∈ Buffer updated by T do
      Buffer.flushPage(P);
   Log.append(commit, T).
abort(T)
   Log.append(abort, T);
   for each LogRec ∈ Log with P updated by T do
      Buffer.undoPage(P, LogRec.BI).
restart()
   for each (begin, T_i) ∈ Log do
      if (commit, T_i) ∉ Log
      then undo (T_i).
```

Figure 9.7: Operators for the *Undo-NoRedo* algorithm

9.6.2 NoUndo-Redo Algorithm

In this class of algorithms, undoing a transaction to recover from a system failure is never necessary. Thus only the *after-image* must be written to the log, to deal with system failures.

The pages updated by a transaction are written to the database when the transaction has committed, but after the writing of the *commit* record to the log (Figure 9.8).

The recovery algorithm performs efficiently with transaction failures, since it is sufficient to ignore the pages updated in the buffer, and for this reason it is preferred in conjunction with an *optimistic* concurrency control algorithm, which aborts transactions in case of conflict, described in the next chapter.

```
beginTransaction()
   T := newTransactionIde();
   Log.append(begin, T);
   return(T).
write(T, P, V)
   Buffer.getAndPinPage(P);
   AI := V;
   Log.append(W, T, P, AI );
   Buffer.updatePage(P, V).
commit(T)
   Log.append(commit, T);
   for each P ∈ Buffer updated by T do
      Buffer.unpinPage(P).
abort(T)
   Log.append(abort, T);
   for each P ∈ Buffer updated by T do
      { Buffer.setNoDirty(P);
        Buffer.unpinPage(P). }
restart()
   for each (begin, T_i) ∈ Log do
      if (commit, T_i) ∈ Log
      then redo (T_i).
```

Figure 9.8: Operators for the *NoUndo-Redo* algorithm

9.6.3 Undo-Redo Algorithm

This class of algorithms require both undo and redo, and is the most complicated one. Thus both the *after-image* and the *before-image* must be written to the log to

deal with transaction and system failures.

The pages updated by a transaction may be written to the database before, during, or after commit (Figure 9.9). The decision is left to the buffer manager.

The recovery algorithm is more costly that in the previous cases, but this solution is usually preferred by the commercial DBMS, since it privileges the normal execution of transactions. Systems that use this algorithm are, for instance, DB2, Oracle, SQL Server.

```
beginTransaction()
   T := newTransactionIde();
   Log.append(begin, T);
   return(T).
write(T, P, V)
   Buffer.getAndPinPage(P);
   BI := page P; AI := V;
   Log.append(W, T, P, BI, AI);
   Buffer.updatePage(P, V);
   Buffer.unpinPage(P).
commit(T)
   Log.append(commit, T).
abort(T)
   Log.append(abort, T);
   for each LogRec ∈ Log with P updated by T do
      Buffer.undoPage(P, LogRec.BI).
restart()
   for each (begin, T_i) ∈ Log do
      if (commit, T_i) ∈ Log
      then redo (T_i) else undo (T_i).
```

Figure 9.9: Operators for the *Undo-Redo* algorithm

From the above discussion it is easy to imagine how the quality of an algorithm depends on many factors. In particular, it has been shown in [Agrawal and Witt, 1985] how it also depends on the techniques adopted by concurrency control. The results of experiments on the performances of various combinations of techniques for concurrency control and recovery for centralized systems in terms of average duration of transactions are presented. From the analysis of the results it has been shown that the most appropriate combination is the use of the *Undo-Redo* algorithm and the *strict two-phase locking* concurrency control, described in the next chapter.

In the rest of the chapter we will only consider an *Undo-Redo* algorithm and an introduction on the subject of recovery from system and media failures is presented.

9.7 Recovery from System and Media Failures

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

- Bring the database in its committed state with respect to the execution up to the to the system failure.
- Restart the normal system operations.

Let us describe a simple version of an algorithm for the first task, a process in two phases called ***rollback*** and ***rollforward***. In the following the checkpoint is assumed of type *buffer-consistent – Version 1*, and that the *rollback* phase is performed before the *rollforward* one.

In the *rollback* phase the log is read from the end to the beginning (a) to undo, if necessary, the updates of the non terminated transactions and (b) to find the set of the identifiers of the transactions which are terminated successfully in order to redo their operations. In particular, the following actions are performed until the first checkpoint record is found, to construct two sets, initially both empty: L_r (*set of transactions to be redone*), L_u (*set of transactions to be undone*).

– If a record is (*commit*, T_i), T_i is added to L_r.
– If a record is an update of a transaction T_i, if $T_i \notin L_r$, the operation is undone and T_i is added to L_u.
– If a record is (*begin*, T_i), T_i is removed from L_u.
– If a record is (*CKP*, L), for each $T_i \in L$, if $T_i \notin L_r$, then T_i is added to L_u. If L_u is not empty, the *rollback* phase continues after the checkpoint record until L_u is emptied.

In the *rollforward* phase, the log is read onward from the first record after the checkpoint to redo all the operations of the transaction T_i successfully terminated, that is $T_i \in L_r$. When a record is (*commit*, T_i), T_i is removed from L_r and this phase terminates when L_r is emptied.

Example 9.3
Figure 9.10 shows the different states of a databases caused by transactions managed with the *Undo-Redo* algorithm. S_1 is a state during a checkpoint that is not correct due to the presence of active transactions (time t_{ck}). A system failure at time t_f leaves the database in a non correct state S_2.

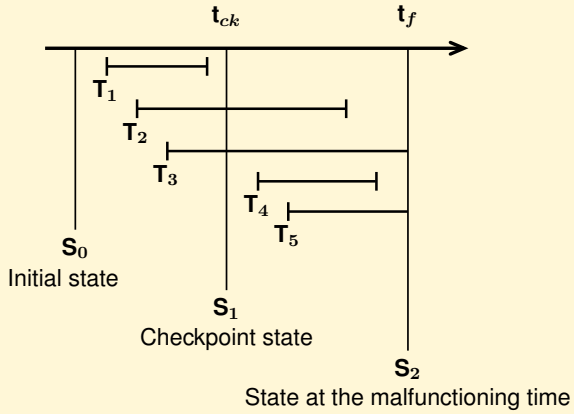


Figure 9.10: States of a database

During the restart, the actions of transaction T_3 and T_5 must be undone, while those of T_2 and T_4 must be redone since, even if T_2 and T_4 have reached the *commit* point before the failure (in the log there are the records (*commit*, T_2) and (*commit*, T_4)), it is not guaranteed that their updates had been effectively transferred to the database.

Suppose that the log contains the following information at the time of system failure (LSN is the *Log Sequence Number*):[3]

Log					
LSN	**Record**	**LSN**	**Record**	**LSN**	**Record**
1	(*begin*, 1)	6	(*begin*, 3)	12	(*begin*, 5)
2	$(W, 1, A, 50, 20)$	7	$(W, 3, A, 20, 30)$	13	$(W, 5, E, 50, 30)$
3	(*begin*, 2)	8	(*CKP*, $\{2, 3\}$)	14	(*commit*, 2)
4	$(W, 2, B, 10, 20)$	9	$(W, 2, C, 5, 10)$	15	$(W, 3, B, 20, 30)$
5	(*commit*, 1)	10	(*begin*, 4)	16	(*commit*, 4)
		11	$(W, 4, D, 5, 30)$		

Figure 9.11 and Figure 9.12 show the actions to be performed in the *rollback*

and *rollforward* phases of the *restart* procedure.

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

Figure 9.11: Actions during the *rollback* phase

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

Figure 9.12: Actions in the *rollforward* phase

In the case of *media failure*, that is of the loss of the database but not of the log, a *cold restart* is performed through the following steps:

- The most recent database backup is reloaded.
- A *rollback* phase is performed on the log, until the DUMP record, that marks the time in which the normal operations are restarted after the backup, to find the identifiers of the transaction successfully terminated.
- A *rollforward* phase is performed to update the copy of the database, by redoing the updates of the transactions terminated with success from the time of the backup to that of the failure.

9.8 The ARIES Algorithm *

ARIES (*Algorithm to Recovery and Isolation Exploiting Semantics*) is a general recovery algorithm of the type *Undo-Redo* to reduce logging overhead, and minimize recovery time from transaction, system and media failures. ARIES has been designed for the IBM DB2 system and it is used by other commercial systems. Its complete description is outside the scope of this text, but it is well presented in the literature on the subject and has been used for the transaction management in the JRS system, with a Java implementation.

ARIES generalizes the solutions previously described to take into account the following important aspects.

Write in the log only the description of the operation. Until now, for simplicity, we have supposed that a page update requires the writing of the entire page and the storing of its before and after images in the log file. However, transactions actually update records of the database and in the log only a description of the operations is written (*operation logging* or *logical logging*) together with the information necessary to perform an undo or a redo of the update. For instance, if a field of a record is modified, it is necessary to write only the old and new version of that field, in addition to the description of the operation.

Store a part of the log in the buffer. Until now we assumed that each record of the log is immediately written to the permanent memory. However, for reasons of performance, the log pages contain more than one record, and a part of temporary memory is reserved as a *log buffer*. Each page of the log remains in the buffer until it is full. This of course does not prevent the writing of the page to the permanent memory, even if not full, after an explicit request.

Since a record of the log can stay around in the buffer for some time, before its transfer to the permanent memory, a system failure causes the loss of all the records present in the log buffer. For this reason, it is necessary to take appropriate action so that the undo and redo rules previously discussed are obeyed.

A new type of checkpoint. As discussed before, storing a checkpoint in the log is equivalent to making a snapshot of the state of the DBMS. The checkpoint can be taken in an asynchronous manner, by avoiding in this way the interruption of the normal activity of the system, which can continue to execute transaction, and without forcing the writing of the dirty pages to the permanent memory. A checkpoint of this type is called a ***fuzzy checkpoint***, and requires the use of two tables: the *transaction table*, which contains information about the state and the operations of the transactions, and the *dirty pages table*, in which each element contains information on the pages modified in the buffer during normal processing. In this way, at any time it is possible to know the list of the dirty pages in the buffer, and perform a checkpoint as follows.

Definition 9.17 *Fuzzy checkpoint.*

The checkpoint procedure performs the following actions:

1. Write a *beginCKP* record, to signal the beginning of the checkpoint procedure.
2. Build a *endCKP* record, by inserting in it the content of the transaction table and the dirty pages table.
3. When the *endCKP* record is written to the permanent memory, the LSN of the *beginCKP* record is stored in a file used in the restart procedure, called the ***master record***.

9.9 Summary

1. A transaction is a sequential program constituted by a set of read and write operations to the database, and a set of operations in temporary memory, which the

DBMS executes by guaranteeing the following properties: *Atomicity*, *Isolation* and *Durability*.

2. The *transaction and recovery manager* controls the execution of the transactions and guarantees their properties by writing the history of the updates performed on the database to a *log*, and providing appropriate algorithms to undo the updates, in the case of transaction failures, and redo all of them, in the case of system failures.
3. The recovery managers differ according to the way in which they use the *redo* and *undo* algorithms.
4. The *checkpoint* is another item of information stored in the log with the aim of reducing the part of the log that must be examined during the restart phase after a system failure. Different methods have been studied to store the information relevant to the checkpoint in order to reduce the restarting time.
5. ARIES is the most popular *undo-redo* algorithm for the management of transactions.

Bibliographic Notes

The main reference for the topics covered in this chapter is the classical book [Bernstein et al., 1987], that is downloadable for free from the Phil Bernstein's home page. The books cited in Chapter 1 and [Gray and Reuter, 1997] devote ample space to transaction management. ARIES is described in [Mohan et al., 1992; Mohan, 1999] and in the books [Ramakrishnan and Gehrke, 2003; Silberschatz et al., 2010].

Exercises

Exercise 9.1 Define the concepts of transaction, of transaction failure and system failure. Describe the algorithm *NoUndo-Redo*.

Exercise 9.2 Consider a DBMS that uses the recovery algorithm *Undo-Redo*. Which of the following statements are true? Briefly justify your answers.

(a1) All the updates of a transaction must be transferred to the database *before* the successful termination of the transaction (i.e. before the commit record is written to the log).

(a2) All the updates of a transaction must be transferred to the database *after* the successful termination of the transaction (i.e. after the commit record is written to the log).

(a3) The updates of a transaction may be transferred to the database *before* or *after* the successful termination of the transaction (i.e. before or after the commit record is written to the log).

(b1) The updates of a transaction must be transferred to the database *before* their before-images have been previously written to the log in the persistent memory.

(b2) The updates of a transaction must be transferred to the database *after* their before-images have been previously written to the log in the permanent memory.

(b3) The updates of a transaction may be transferred to the database *before* or *after* their before-images have been previously written to the log in the permanent memory.

Exercise 9.3 Describe the *NoUndo-Redo* algorithm and how the commit and the abort are implemented.

Exercise 9.4 Consider the following log records, assuming that A, B, C and D are the pages with integer values. Assume the log entries are in the format (W, Trid, Variable, Old value, New value).

```
(BEGIN T1)
(W, T1, A, 20, 50)
      (BEGIN T2)
      (W, T2, B, 20, 10)
      (COMMIT T2)
(CKP, {T1})
(W, T1, C, 10, 5)
      (BEGIN T4)
      (W, T4, D, 30, 5)
(COMMIT T1)
SYSTEM FAILURE
```

Suppose that the transactions are managed with the *Undo-Redo* algorithm, and the checkpoint with the *Buffer-consistent checkpoint – Version 1*. Show the actions made with the system *restart*.

Exercise 9.5 Consider the following log records from the start of the run of a database system, and suppose that the transactions are managed with the *Undo-Redo* algorithm, and the checkpoint with the *Buffer-consistent checkpoint – Version 1*.

```
 1)  (BEGIN T1)
 2)  (W, T1, A, 25, 50)
 3)  (W, T1, B, 25, 250)
 4)        (BEGIN T2)
 5)  (W, T1, A, 50, 75)
 6)        (W, T2, C, 25, 55)
 7)  (COMMIT T1)
 8)              (BEGIN T3)
 9)              (W, T3, E, 25, 65)
10)        (W, T2, D, 25, 35)
11)  (CKP {T2,T3})
12)        (W, T2, C, 55, 45)
13)        (COMMIT T2)
14)        (BEGIN T4)
15)        (W, T4, F, 25, 120)
16)              (COMMIT T3)
17)        (W, T4, F, 120, 150)
18)        (COMMIT T4)
```

Assume the log entries are in the format (W, Trid, Variable, Old value, New value). What is the value of the data items A, B, C, D, E on F on disk after the *restart* if the system crashes

1. just before line 10 is written to disk.
2. just before line 13 is written to disk.
3. just before line 14 is written to disk.
4. just before line 18 is written to disk.
5. just after line 18 is written to disk.

Chapter 10

CONCURRENCY MANAGEMENT

When transactions are executed concurrently, their operations on the database are interleaved, i.e. the operations of a transaction may be performed between those of others. This can cause interference that leaves the database in an inconsistent state. The ***Concurrency Manager*** is the system module that ensures the execution of concurrent transactions without interference during database access.

10.1 Introduction

The classic example of interference during the execution of concurrent transactions is that which causes the loss of updates.

Let us assume that John and Jane have a joint savings account and both go to different tellers. The current balance is 500€. Jane wishes to add 400€ to the account. John wishes to withdraw 100€. Let us assume that the events happen in the order in which they are shown in Figure 10.1, in which $r_i[x]$ (or $w_i[x]$) means that transaction i reads (or writes) the database element x. At the end of these operations the saving account contains 900€, instead of 800€. Although both transactions are completed properly, the effect of T_2 is canceled when T_1 ends, and this certainly is not a correct way to allow more than one person to use the same account.

T_1	T_2
begin	
$r_1[x]$	
	begin
	$r_2[x]$
	$x := x - 100$
$x := x + 400$	
	$w_2[x]$
$w_1[x]$	
	commit
commit	

Figure 10.1: Execution of transactions with loss of updates

In this example, the error is caused by the interleaved execution of operations belonging to different transactions. To avoid this and other problems, the concurrent execution of transactions must be controlled, and in particular the way that their database operations are interleaved.

We assume that each transaction is ***consistent***, i.e. when it executes in isolation, it is guaranteed to map consistent states of the database to consistent states. A simple way to avoid interference among concurrent transactions is to execute them one after another. i.e. to allow only a *serial execution* of them.

Definition 10.1 *Serial Execution*

An execution of a set of transactions $T = \{T_1, \ldots, T_n\}$ is *serial* if, for every pair of transactions T_i and T_j, all the operations of T_i are executed before any of the operations of T_j or vice versa.

If the initial database state is consistent and reflects a real-world state, then the serial execution of a set of consistent transactions preserves these properties. Unfortunately, serial executions are impractical from a performance perspective. In general, since concurrency means interleaved executions of transactions, it is sufficient that the system guarantees that the resulting execution will have the same effect as serial ones. Such executions are called ***serializable***.

Definition 10.2 *Serializable Execution*

An execution of a set of transactions is *serializable* if it has the same effect on the database as some serial execution of the transactions that commit.

The aborted transactions are not considere because it is assumed the transactions are atomic and then the updates of aborted transactions are undone.

Since the serial executions are correct and each serializable execution has the same effect of a serial one, also serializable executions are correct. The execution shown in Figure 10.1 is not serializable. In fact, both serial executions (T_1, T_2) and (T_2, T_1) produce the same effect, which is different from that of the execution in the example, where in particular is lost the update of T_2.

The DBMS module that controls the concurrent execution of transactions is called the ***Concurrency Manager*** or ***Scheduler***. There are many scheduling algorithms to obtain serializability.

In the following sections, we will present some elements of the *theory of serializability*, a mathematical tool that allows us to prove whether a given scheduler is correct. The theory uses a structure called ***history*** (or ***schedule***) to represent the chronological order in which the operations of a set of concurrent transactions are executed, and defines the properties that a history has to meet to be serializable [Bernstein et al., 1987].

10.2 Histories

From the point of view of a DBMS a transaction T_i starts with a *begin*, then continues with a (possible empty) partially ordered sequence of read ($r_i[x]$) and write ($w_i[x]$) operations on the database, and terminates either with an *abort* (a_i) or a *commit* (c_i) operation. A transaction that terminates with an abort does not update the database (*atomicity*). If a transaction commits, then all its updates are stored persistently (*durability*).

To simplify the presentation, we will make the following additional assumptions:

– The database is a fixed set of independent records that can only be read or updated, i.e. we do not consider several collections of record and operations of insertion or deletion of records (therefore, the write operation $w_i[x]$ means the update of x). We will consider the general case in Section 10.6.
– A transaction reads and updates a specific record at most once. The results that will be shown do not depend on this assumption and can be extended easily to the general case, by indicating the operations of T_i with $r_{ij}[x]$ (or $w_{ij}[x]$), instead of $r_i[x]$ (or $w_i[x]$), and revising the definitions that will be given later.

Example 10.1
The transaction defined by the program

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

is seen by the DBMS as the sequence of operations: $r_1[x], r_1[y], w_1[x], c_1$. The operation index identifies an execution of the program and differs from those of other programs.

To treat formally the concurrent execution of a set of transactions $\{T_1, T_2, \ldots, T_n\}$, the concept of ***history*** is used. A history indicates the relative order of execution of operations belonging to $T_1, T_2, \ldots, T_n$.

Definition 10.3 *History*

Let $T = \{T_1, T_2, \ldots, T_n\}$ a set of transaction. A *history* H on T is an ordered set of operations such that:

1. the operation of H are those of $T_1, T_2, \ldots, T_n$;
2. H preserves the ordering between the operations belonging to the same transaction.

Intuitively, the history H represents the actual or potential execution order of the operations of the transactions $T_1, T_2, \ldots, T_n$. For example, Figure 10.2 shows a possible history H_1 of the set $\{T_1, T_2, T_3\}$. Figure 10.3 shows the same history H_1, with the operations ordered from top to bottom; depending on the complexity of the history we will use this notation or that of Figure 10.2.

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

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

Figure 10.2: An example of history involving $\{T_1, T_2, T_3\}$

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

Figure 10.3: Another representation of the history H_1 in Figure 10.2

10.3 Serializable History

Before characterizing a *serializable history*, that is, the histories that represent serializable executions, it is necessary to define under which conditions two histories are equivalent.

For equivalent histories we could mean that they produce the same effects on the database, namely that the values written to the database by the committed transactions are equal. Since a DBMS knows nothing of the computations made by a transaction in temporary memory, a weaker notion of equivalence which takes into account only the order of operations in *conflict* made on the database is preferred.

Definition 10.4 *Operations in conflict*

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

The concept of operations in conflict allows to characterize three types of abnormal situations that may arise during the concurrent execution of transactions.

1. *Dirty Reads* (*Write-Read Conflict*). Consider the following piece of a history: $H = \ldots w_1[x], r_2[x], \ldots$ The transaction T_2 reads x updated by T_1 which has not yet committed. This type of read, called *dirty read*, can give rise to executions not serializable. Consider the history of Figure 10.4 where T_1 transfers 1000€ from account A to account B, when both have an initial balance of 2000€, while T_2 increases by 10% each account. The execution result is different from that one produced by a serial execution of the two transactions.

T_1	T_2
$r_1[A]$	
$A := A - 1000$	
$w_1[A]$	
	$r_2[A]$
	$A := A + 100$
	$w_2[A]$
	$r_2[B]$
	$B := B + 200$
	$w_2[B]$
	commit
$r_1[B]$	
$B := B + 1000$	
$w_1[B]$	
commit	

Figure 10.4: Example of a dirty read

2. *Unrepeatable Read* (*Read-Write Conflict*). Consider the following piece of a history: $H = \ldots r_1[x], w_2[x], r_1[x], \ldots$ Transaction T_2 writes a data x previously read by the transaction T_1, still active, which then rereads it getting a different value even if in meanwhile T_1 has not updated it. This effect obviously can not be achieved by any serial execution of the two transactions.
3. *Lost Update* (*Write-Write Conflict*). Consider the following piece of a history: $H = \ldots w_1[x], w_2[x] \ldots$ The two transactions, while attempting to modify a data item x, both have read the item's old value before either of them writes the item's new value. No serial execution of the two transactions would lead to the same result, as we have seen before (Figure 10.1).

Note that the presence of these conflicts is not a sufficient condition to bring the database to an incorrect state, but it is better to avoid them due to the risk of having non-serializable executions.

Definition 10.5 *Histories c-equivalent*

Two histories H_1 and H_2 are *c-equivalent* (*conflict-equivalent*), that is *equivalent with respect to operations in conflict*, if

1. they are defined on the same set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ and have the same operations;
2. they have the same order of operations in conflict of transactions terminated normally, i.e. for each pair of operations in conflict $p_i \in T_i, q_j \in T_j$ such that $a_i, a_j \notin H_1$, p_i *precedes* q_j in H_1 if and only if p_i *precedes* q_j in H_2.

The condition (1) requires that the histories H_1 and H_2 contain the same set of operations to be comparable, the condition (2) requires that H_1 and H_2 have the same order of the operations in conflict of committed transactions. In particular, it follows that, for any given x in H_1,

1. if T_i executes $r_i[x]$ after T_j executes $w_j[x]$, the same is true in H_2 (in both history each read operation reads the value produced by the same write operation) and

2. if T_i executes $w_i[x]$ before that T_j executes $w_j[x]$, the same is true in H_2 (in both schedule the last write operation are the same).

This definition of c-equivalence of histories is motivated by the fact that the result of concurrent execution of $T_1, T_2, \ldots, T_n$ depends only on the order of execution of operations in conflict. In fact, two operations not in conflict, for example two read operations, have the same effect on the database regardless of their order of execution; while the effect on the database of two conflicting operations depends on their order of execution.

A history can be transformed into a c-equivalent one using the following property.

Definition 10.6 *Commutative Property*

Two database operations o_i and o_j commute if, for all initial database states, they (1) return the same results and (2) leave the database in the same final state, when executed in either the sequence o_i, o_j or o_j, o_i.

Example 10.2

Let us consider the history in Figure 10.3, a c-equivalent history can be obtained by a series of simple interchanges of commutative non-conflicting operations of different transactions. For example, if we interchange $r_1[x]$ and $r_2[x]$ we have another c-equivalent history. Continuing with this process we can move the operations $w_2[y]$ and c_2 before the operations of T_3 and T_1 obtaining the c-equivalent history of Figure 10.5.

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

Figure 10.5: History H_2 c-equivalent to history H_1 of Figure 10.3

The history H_3 in Figure 10.6 is not c-equivalent to the history H_1 in Figure 10.3 because the operations in conflict $w_1[y]$ and $w_2[y]$ are in a different order and the cannot be interchanged.

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

Figure 10.6: History H_3 non-c-equivalent to the history H_1 in Figure 10.3

Other types of equivalence between history have been proposed. The reader may refer to the texts listed in the references for further study. Although the equivalence with respect to operations in conflict may be more restrictive than others, hence allowing less concurrency, it is the most interesting from a computational point of view and has been very successful in DBMSs implementation.

A *serial* history represents an execution in which there is no interleaving of the operations of different transactions. Each transaction is completely executed before the next one starts.

We can now define a stronger condition that is sufficient to ensure that a history is serializable, called ***c-serializability*** (***conflict-serializability***), adopted by commercial systems.

Definition 10.7 *Conflict Serializability*

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

Let us consider the three transactions in Figure 10.2. A serial history is T_1, T_2, T_3. A history non-serial but c-serializable is H_2 in Figure 10.5. In fact, the operations $w_1[y]$ and c_1 can be moved before those of T_3 obtaining the serial history T_2, T_1, T_3 in Figure 10.7. A history both non-serial and *non-c-serializable* is H_3 in Figure 10.6.

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

Figure 10.7: Serial history c-equivalent to the history H_2 in Figure 10.5

Because of the assumption that the data can only be read or updated, but not inserted or deleted, the following property holds:

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

For example, the history in Figure 10.8 is equivalent to the serial one T_1, T_2, T_3, because in both cases T_1 reads the same data item, and the final value of x is that written by T_3, but the history is not c-equivalent to the serial one T_1, T_2, T_3 because the write operations in T_1 and T_2 are in a different order.

Serialization Graph

Although it is possible to examine a history H and decide whether or not it is *c-serializable* using the commutativity and reordering of operations, there is another simpler way to proceed based on the analysis of a particular graph derived from H, called ***serialization graph***.

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

Figure 10.8: History serializable but non-c-serializable

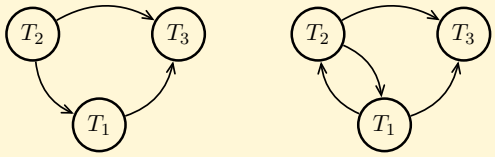
Definition 10.8 *Serialization Graph*

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

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

We say that two transactions T_i and T_j conflicts if $T_i \rightarrow T_j$ appears in $SG(H)$.

The serialization graphs of the histories H_2 and H_3 in Figure 10.5 and and 10.6 are shown in Figure 10.9.



(a) $GS(H_2)$ **(b)** $GS(H_3)$

Figure 10.9: Serialization graphs of H_2 and H_3

Theorem 10.1 *C-Serializability Theorem*

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

Proof

($\Rightarrow$) Assume that H is c-serializable and c-equivalent to a serial history H^s. If there is an arc $T_i \rightarrow T_j$ in $SG(H)$, there is a operation in T_i that appears

before and conflicts with some operation in T_j in H. Since H is c-equivalent to H^s, T_i strictly precedes T_j in H^s. Suppose that in $SG(H)$ exists a cycle $T_1 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$, it follows the absurd that in H^s every transaction T_1, ..., T_k strictly precede itself. Therefore, cycles cannot exist in $SG(H)$.

($\Leftarrow$) Assume that there are m transactions $T_1, \ldots, T_m$ in H. If $SG(H)$ is acyclic, there is a topological order of the transactions in the graph as follows.[1] Since the graph is acyclic, there is at least a node T_i without incoming arcs. Then T_i is added to the serial history H^s and removed from the graph. Then the process is repeated on the graph as long as exists a node. By construction, no arc in the graph is directed toward a transaction already in H^s and so the order in which the transactions are listed is serial. ■

In Figure 10.9, the graph $SG(H_2)$ only is acyclic and then H_2 only is c-serializable. A serial schedule where transactions appear in an order consistent with arcs of $SG(H_2)$ is T_2, T_1, T_3, as was seen in Figure 10.7, which is obtained by a topological order of $SG(H_2)$.

10.4 Serializability with Locking

From the analysis of the serialization graph it can be verified a posteriori if a history is c-serializable. Histories and serialization graphs, however, are abstract concepts, and during the execution of a set of transactions, the serialization graph is not constructed. Conversely, the c-serializability theorem is used to prove that the scheduling algorithm for the concurrency control used by a scheduler is correct, i.e. that all histories representing executions that could be produced by it are *c-serializable*.

10.4.1 Strict Two-Phase Locking

There are many scheduling algorithms to obtain serializability; a very simple one is the ***strict two-phase locking*** protocol (***Strict 2PL***), very popular in commercial systems.

The idea behind locking is intuitively simple. Each data item used by a transaction has a lock associated with it, a *read* (*shared, S*) or a *write* (*exclusive, X*) lock, and *Strict 2PL* protocol follows two rules (Figure 10.10):

1. If a transaction wants to *read* (respectively, *write*) a data item, it first request a *shared* (respectively, *exclusive*) lock on the data item.

   Before a transaction T_i can access a data item, the scheduler first examines the lock associated with the data item. If no other transaction holds the lock, then the data item is locked. If, however, another transaction T_j holds a lock in *conflict*, then T_i must wait until T_j releases it.
2. All locks held by a transaction T_i are *released together* when T_i commits or aborts.

The following notation, called a ***compatibility matrix***, is used for describing lock-granting policies: a row correspond to a lock that is already held on an element x by

1. A *topological order* of an acyclic directed graph G is any order of the nodes of G such that for every arc $a \rightarrow b$, node a precedes node b in the topological order. We can find a topological order for any acyclic directed graph by repeatedly removing nodes that have no predecessors among the remaining nodes. For a given G, there may exist several topological orders.

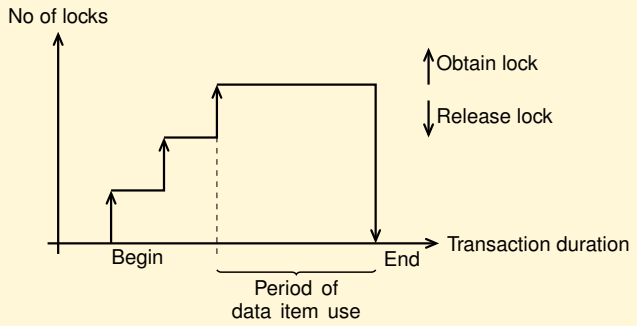


Figure 10.10: Strict two-phase locking protocol

another transaction, and the columns correspond to the mode of a lock on x that is requested:

	S	X
S	Yes	No
X	No	No

A strict two-phase locking protocol is so named because for each transaction there is a first phase in which it requests locks, and a second phase in which it releases locks. Requests to acquire and release locks are automatically inserted into transactions by the *Transaction Manager* module of the DBMS.

Lock and unlock requests are handled by the scheduler with the use of a data structure called ***Lock Table***, an in-memory hash table of lock entries. An entry contains the identifier of the data item being locked (the lock table key), the type of lock granted or requested, a list of transactions holding lock, a queue of lock requests. When a lock request arrives, it is handled atomically and added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.

The following theorem shows the importance of *Strict 2PL*.

Theorem 10.2

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

Proof

Suppose not. Let H be a *Strict 2PL* history non c-serializable. Then by the serializability theorem, the serialization graph $SG(H)$ has cycle $T_{i_1} \rightarrow T_{i_2} \rightarrow \cdots \rightarrow T_{i_p} \rightarrow T_{i_1}$. Then T_{i_1} released a lock before T_{i_2} obtained some lock; T_{i_2} released a lock before T_{i_3} obtained some lock, and so on. Finally, T_{i_p} released a lock before T_{i_1} obtained some lock. Therefore, T_{i_1} released a lock before T_{i_1} obtained some lock, contradicting the assumption that T_{i_1} is *Strict 2PL* locked. ■

Another important property holds:

The set of Strict 2PL histories is a proper subset of the c-serializable histories.

For example, the history in Figure 10.11 is c-serializable — the serialization graph ($T_3 \to T_1 \to T_2$) is acyclic and the history is c-equivalent to the serial one (T_3, T_1, T_2) — but could not have been generated using the *Strict 2PL* protocol.

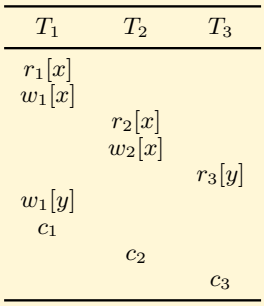


Figure 10.11: History c-serializable that cannot be generated by a *Strict 2PL* protocol

10.4.2 Deadlocks

Two-phase locking is simple, but the scheduler needs a strategy to detect *deadlocks*. Consider a situation in which transaction T_i has locked item A and needs a lock on item B, while at the same time transaction T_j has locked item B and now needs a lock on item A. A deadlock occurs because none of the transactions can proceed.

Deadlock Detection. A strategy to detect deadlocks uses a *wait-for graph* in which the nodes are active transactions and an arc from T_i to T_j indicates that T_i is waiting for a resource held by T_j. Then a cycle in the graph indicates that a deadlock has occurred, and one of the transactions of the cycle must abort.

Once a deadlock has been detected, the standard method to decide which transaction to abort is to choose the "youngest" transaction by some metric. For example, the transaction with the newest timestamp, or the transaction that holds the least number of locks, etc.).

Although the check for cycles requires a linear complexity algorithm with respect to the number of nodes of the graph, the actual cost of the management of the graph in real cases may discourage the use. For example, a system with a workload of 100 transactions per second, each of which reads 10 pages and lasts 10 seconds, has to manage a graph of 1 000 nodes and 10 000 requests for locks by the 1 000 active transactions. For this reason in commercial systems the existence of a wait cycle may be controlled at predetermined time intervals, or even the wait-for graph is not build, but the *timeout* strategy is used: if a transaction has been waiting too long for a lock, then the scheduler simply presumes that deadlock has occurred and aborts the transaction.

Deadlock Prevention. Another approach to deal with deadlocks is to use a locking protocol that disallows the possibility of a deadlock occurring as follows.

Each transaction T_i receives a timestamp $t_s(T_i)$ when it starts: T_i is older than T_j if $t_s(T_i) < t_s(T_j)$. Moreover, to each transaction is assigned a priority on the basis of its timestamp: the older a transaction is, the higher priority it has.

When a transaction T_i requests a lock on a data item that conflicts with the lock currently held by another active transaction T_j, two algorithms are possible:

1. *Wait-die* (or *non-preemptive* technique):

   if T_i is older than T_j
   then T_i waits until T_j terminates
   else T_i dies (aborts);

   Therefore, an older transaction *waits* only for a younger one, otherwise the younger *dies*. With this method, once a transaction has acquired a lock, it will never be aborted (preempted) by a higher priority transaction.
2. *Wound-wait* (or *preemptive* technique):

   if T_i is older than T_j
   then T_i wounds (aborts) T_j
   else T_i waits until T_j terminates;

   Therefore, an older transaction *wounds* a younger one to take its lock, otherwise the younger *waits* for the older one. This method use a *preemptive* technique, and a lower priority transaction is *killed* when interferes with a higher priority transaction.

In both methods, *when an aborted transaction T_i is restarted, it has the same priority it had originally*, i.e. it restarts with the same timestamp. The methods have the following property.

Theorem 10.3

Both the *Wait-Die* and *Wound-Wait* methods do not create deadlocks.

Proof

Let us show that cycles cannot exist in the *wait-for* graphs, which are not constructed. In fact, suppose that the active transactions are T_1, T_2 and T_3, and the following cycle exists $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$. Let T_2 the older transaction.

With the method *wait-die* a transaction can only wait for a younger one, and therefore it is not possible that $T_1 \rightarrow T_2$.

With the method *wound-wait* a transaction can only wait for an older one, and therefore it is not possible that $T_2 \rightarrow T_3$.

Thus the cycle cannot exist and then deadlocks cannot be created. ■

Another interesting property is that when the methods need to choose between two transactions which one to abort, it is always preferred the *younger* one, because it has probably done less work, and therefore it is less costly to abort. Moreover, the aborted transaction is restarted again with the same value of the time stamp, and therefore it sooner or later will become the oldest transaction and will not be interrupted (*no starvation*)

However, the behavior of the two methods is very different:

– With the method *wound-wait* a transaction T can wait for data locked by an older transaction or restarts a younger one T_y.
  The most likely case is the first and then the method favors the waiting rather than restarting. In addition, when the younger transaction T_y restarts, and it requests the data locked by the older one T, it waits.
– With the method *wait-die* a transaction T can wait for data locked by a younger transaction or it restarts due to an older one T_o.
  The most likely case is the second and then the method favors the restarting rather than waiting. Furthermore, when the transaction T restarts, and requests the data locked by the older T_o, it must restart. However, the work of T to undo for a restart should be less than that of T_y of the previous case, because the conflict arises when it requests a lock on the data and not when the transaction is operating on the data previously locked.

Compared with the deadlock detection solution, the deadlock prevention methods are easier to implement, but some transactions are needlessly aborted while they would run without restarts using the technique of deadlock detection.

The deadlock detection solution is usually preferred because, statistically, the deadlock conditions are much less frequent than the conditions involving the aborts and restarts of transactions with the other methods.

10.5 Serializability without Locking

The methods of concurrency control with the use of locks, presented above, are based on the fact that whenever a transaction attempts to perform a database operation, it must request a permission. However, the transactions can commit at any time without requesting permission. For this reason, the methods are called ***pessimistic*** because are based on the idea that a bad thing is likely to happen: the accesses that transactions make to the database are likely to conflict.

Other methods, called ***optimistic***, have been studied for concurrency control which, instead, do not lock data because are based on the idea that bad things are not likely to happen: the accesses that transactions make to the database are not likely to conflict. However, when a transaction requests to commit, the system controls that no bad thing has happened. Let us show one of them and for further details refer to the references cited in the bibliographic notes.

Snapshot Isolation. With this method a transaction can perform any database operation without requesting permission, and taking advantage of *multiversions* of each data item. However, the transactions must request permission to commit. For this reason, the method is called *optimistic*.

The solution is used by Oracle, and other commercial systems, and has the following rules:

– Reads and writes are performed without the use of locks on data.
– Each transaction T_i reads the data of the database version (*snapshot*) produced by all the transactions that committed before T_i starts, but no effects are seen of other concurrent transactions (two transactions are concurrent if one starts with its own version of the database while the other is active).
– The writings of a transaction T_i are collected in the *write set* WS_i and they are visible by T_i but not by other concurrent transactions.

– If a transaction T_1 makes some writes, then it can commit only if another concurrent transaction T_2 does not exist such that (a) T_2 committed and (b) T_2 updated a data item that T_1 also updated ($\mathrm{WS}_1 \cap \mathrm{WS}_2 \neq \emptyset$). Otherwise T_1 aborts. This is called the "First-Committer-Wins" rule.

This solution is unfortunately also called serializable, but the following example shows that snapshot isolation permits non-serializable executions.

Suppose that a bank manages two joint accounts x and y, each with an initial balances of 100, and the following business rule hold, not specified in the database schema: the sum of the balances satisfies the condition $x + y \geq 0$.

Suppose that there are two concurrent transactions T_1 and T_2; T_1 is withdrawing 150 from account x while T_2 is withdrawing 150 from account y. T_1 and T_2 begin by computing the total balance of accounts x and y and find that balance to be 200. $200 > 150$, so each transaction believes that a 150 withdrawal is permissible, and produce the history of Figure 10.12. The history is not *c-serializable* for the cycle $T_1 \rightarrow T_2 \rightarrow T_1$, but it is possible if the transactions are executed with the *snapshot* technique, even if the end result is a negative total balance of -100 violating the condition $x + y \geq 0$!

T_1	T_2
$r[x = 100]$	
$r[y = 100]$	
	$r[x = 100]$
	$r[y = 100]$
	$w[y = -50]$
	c
$w[x = -50]$	
c	

Figure 10.12: History with the *snapshot* technique

A review of the *snapshot* technique to make it *c-serializable* is given in [Cahill et al., 2008; Fekete et al., 2005].

10.6 Multiple-Granularity Locking *

The concurrency control techniques seen so far, based on the idea of a single record lock, is not sufficiently general to treat transactions that operate on collections of records. For example, if a transaction has to update all the records of a table, with the techniques seen each record must be locked while it would be sufficient to lock exclusively the entire table. On the other hand, if a transaction needs to update one record, it should not be required to lock the entire table, since otherwise concurrency is not possible.

For these reasons other techniques have been developed based on the idea that the data to lock can have different granularities (database, table, page, and record) and among them is defined an inclusion relationship: a database contains a set of tables, each table contains a set of pages, each page contains a set of records (*multiple-granularity locking*).

The inclusion relation between data can be thought of as a tree of objects where each node contains all its children. If a transaction gets an *explicit* S or X lock on a

node, then it has an *implicit lock* in the same lock mode on all the descendants of that node.
To manage locks on data of different granularity, the S and X locks are not enough, but new lock types are introduced, called ***intention locks***. If a data is locked in an intention mode, explicit locking is being done at a finer granularity. Multiple-granularity locking requires that before a node is explicitly locked, a transaction must first have a proper intention lock on all the ancestors of that node in the granularity hierarchy.

The intention lock types are the following:

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

The need of the *SIX* lock is justified by the the following considerations. Consider a transaction that wants to read a table and then update a small subset of the records. Without the *SIX* lock such transaction would have to own both an S and a *IX* lock on the table. Since this situation occurs frequently, the two lock type, S and IX on a table are combined into one, the *SIX*, to simplify the lock manager.

The compatibility matrix for the lock types is as follows:

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

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

1. A node (which is not the root) can be locked by a transaction T_i in S or *IS* mode only if the parent is locked by T_i in *IS* or *IX* mode.
2. A node (which is not the root) can be locked by a transaction T_i in X, *IX* or *SIX* mode only if the parent is locked by T_i in *SIX* or *IX* mode.

Note that the multiple-granularity protocol requires that the locks be acquired from the root of the hierarchy down to the level where it was made the request, whereas locks must be released in the reverse order.

10.7 Locking for Dynamic Databases *

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

Insertion and Deletion. Consider the following cases:

1. The transaction T_1 reads all records with attribute $A_1 = 100$ of table F_1. Under the assumption that the lock is at the record level, T_1 obtained prior blocks S on records that reads. This means that if the records with attribute $A_1 = 100$ are found with a table scan, the transaction must lock as S all the table; if instead the records are selected with an index, only them will be locked. In both cases, the locking of the records currently present in the table does not prevent another transaction T_2 to insert a new record r with attribute $A_1 = 100$ into the table, because the lock X that transaction T_2 needs on r is not incompatible with any lock held by T_1. When T_2 terminates and releases all locks, then if T_1, before terminating, reads again all records with attribute $A_1 = 100$, it gets a different result because of r. This execution is not serializable because does not guarantee a repeatable read of all records with $A_1 = 100$.
2. The transaction T_3 deletes a record with the attribute $A_1 = 100$ from table F_1. The question arises whether a record that no longer exists should be kept locked, and how. If T_3 takes no X lock on the deleted record, another transaction T_4 that wants to read all records with $A_1 = 100$ could see the deletion of the record before the termination of T_3, then making a dirty read of the records with $A_1 = 100$. On the other hand, if T_3 holds a lock X on the deleted record, the transaction T_4 that requires S locks on the records that must read, would be waiting to obtain a lock S on a record deleted without his knowledge and then, once T_3 terminates releases all locks, T_4 after getting the lock would have to read a non-existent record.

The inserted or deleted records are called *phantoms* because they are records that appear or disappear from sets, that is are invisible only during a part of a transaction execution. There is no record blocking solution to solve the *phantom problem*. A solution is *predicate locking*, a locking scheme in which locks are acquired by specifying a predicate: two operations conflicts if at least one of them is a write and the set of records described by their predicates have non-null intersections.

The problem with predicate locking is that it is both difficult to implement, and can have a disastrous impact on performance. So few commercial systems implement it, if any. Instead an acceptable practical solution to this problem is *index locking*. When a set of records that match some predicate is locked, the database system also checks to see if there is an index whose key matches the predicate. If such an index exists, the structure of the index should allow us to easily lock all the pages in which new tuples that match the predicate appear or will appear in the future.

The practical issue of how to locate the relevant pages in the index that need to be locked and what locks need to be acquired is discussed in the following section.

Concurrency Control in B^+–trees The concurrent use of a B^+–tree index by several transactions may be treated in a simple way with *Strict 2PL* protocol, by considering each node as a granule to lock appropriately. This solution, however, would lead to a low level of concurrency due to locks on the first tree levels. A better solution is obtained by exploiting the fact that the indexes are used in a particular way during the operation.

In the case of a search, the nodes visited to reach the leaves, where the data is located, are locked in reading during the visit and unlocked as soon as the search proceeds from one node to the child.

In the case of an insertion, during the visit of the tree when switching from one node A to a child B not full, the locks on A can be released because a possible propagation of the effect of the insert into a leaf stops at node B, called a ***safe node***.

A node is safe for a delete operation if it is at least half full.

The general case with node splits, merging and balancing is more complex and several tree locking techniques have been proposed.

10.8 Summary

1. The concurrency manager is the module of the system that ensures the execution of concurrent transactions without interference. An execution of a set of transactions is said to be *serializable* if it produces the same effect on the database as that obtainable by serially performing the only transactions terminating normally in some order. The *scheduler* controls the concurrent execution of transactions to only allow serializable executions.
2. The *serializability theory* is a mathematical tool that allows one to prove whether a given scheduler is correct. The theory uses a structure called *history* to represent the concurrent execution of a set of transactions and defines the properties that a history has to meet to be serializable.
3. Two histories are *equivalent* with respect to operations in conflict if they involve the same operations of the same transactions terminated normally, and every pair of conflicting operations is ordered the same way. A history is *c-serializable* if it is c-equivalent to a serial history. To determine if a history is *c-serializable* it is tested whether the *serialization graph* is acyclic.
4. The most common scheduler used by commercial DBMS is based on the *Strict 2PL* protocol, which releases all locks only when a transaction ends. The DBMS components involved with concurrency control are the *Lock Manager*, which was previously called the *Concurrency Manager*, and the *Storage Structure Manager*. While the *Lock Manager* uses appropriate internal data structures to deal with lock and unlock requests, and to deal with deadlocks, the *Storage Structure Manager* has the task of choosing from time to time the data to lock and to send the request to the lock manager, before proceeding with the read and write operations on the database. A deadlock is a cycle of transactions, all waiting for another transaction in the cycle to release a lock. The techniques of deadlock detection are usually preferred to those of deadlock prevention.
5. The concurrency control technique based on the idea of locks is not sufficiently general to treat databases where transactions use tables and indexes. For this reason, other techniques have been developed based on the idea of multiple granularity locking. If collections of records change for insertions and deletions, the phantom problem must be solved.
6. An alternative technique to the locking of data, to ensure the serializability of concurrent transactions, is the *optimistic* control: data changes are made in a local area and included in the database only if the operations have not violated serializability, which is checked in a validation phase.

Bibliographic Notes

The main references are those of the previous chapter.

Exercises

Exercise 10.1 Consider the following transactions and the history H:

$T_1 = r_1[a], w_1[b], c_1$
$T_2 = w_2[a], c_2$

$T_3 = r_3[a], w_3[b], c3$
$H = r_1[a], w_2[a], c_2, r_3[a], w_1[b], w_3[b], c_3, c_1$

Answer the following questions:

1. Is H c-serializable?
2. Is H a history produced by a strict 2PL protocol?

Exercise 10.2 Consider the following transactions:

$T_1 = r_1(X), r_1(Y), w_1(X)$
$T_2 = r_2(X), w_2(Y)$

and the history $H = r_1(X), r_2(X), r_1(Y) \ldots$

Show how the history can continue on a system that adopts the strict 2PL protocol.

Exercise 10.3 Consider the following transactions and the history H:

$T_1 = r_1[a], w_1[a], c_1$
$T_2 = r_2[b], w_2[a], c_2$
$H = r_1[a], r_2[b], w_2[a], c_2, w_1[a], c_1$

Answer the following questions:

1. Is H c-serializable?
2. Is H a history produced by a strict 2PL protocol?
3. Suppose that a strict 2PL serializer receives the following requests (where rl and wl means *read lock* and *write lock*):

   $rl_1[a], r_1[a], rl_2[b], r_2[b], wl_2[a], w_2[a], c_2, wl_1[a], w_1[a], c_1$

   Show the history generated by the serializer.

Exercise 10.4 Consider the following history H of transactions T_1, T_2 and T_3 initially arrived at time 10, 20, 30, respectively.

$$H = r_3[B], r_1[A], r_2[C], w_1[C], w_2[B], w_2[C[, w_3[A]$$

We make the following assumptions:

1. A transaction requests the necessary lock (shared lock for read and exclusive lock for write) on a data item right before its action on that item is issued,
2. If a transaction ever gets all the locks it needs, then it instantaneously completes work, commits, and releases its locks,
3. If a transaction dies or is wounded, it instantaneously gives up its locks, and restarts only after all current transactions commit or abort,
4. When a lock is released, it is instantaneously given to any transaction waiting for it (in a first-come-first-serve manner).

Answer the following questions:

1. Is H c-serializable?
2. If the *strict 2PL* is used to handle lock requests, in what order do the transactions finally commit?

3. If the *wait-die* strategy is used to handle lock requests, in what order do the transactions finally commit?
4. If the *wound-wait* strategy is used to handle lock requests, in what order do the transactions finally commit?
5. If the *snapshot* strategy is used, in what order do the transactions finally commit?

Exercise 10.5 Consider the transactions:

$T_1 = r_1[x], w_1[x], r_1[y], w_1[y]$
$T_2 = r_2[y], w_2[y], r_2[x], w_2[x]$

1. Compute the number of possible histories.
2. How many of the possible histories are c-equivalent to the serial history (T_1, T_2) and how many to the serial history (T_2, T_1)?

Exercise 10.6 The transaction T_1 *precedes* T_2 in the history S if all actions of T_1 precede actions of T_2. Give an example of a history S that has the following properties:

1. T_1 precedes T_2 in S,
2. S is c-serializable, and
3. in every serial history c-equivalent to S, T_2 precedes T_1.

The schedule may include more than 2 transactions and you do not need to consider locking actions. Please use as few transactions and read or write actions as possible.

Exercise 10.7 Assume the transactions are managed with the *undo-redo* algorithm, the concurrency mechanism used is strict 2PL protocol, there are only read and write locks, and the checkpoint method used is the *Buffer-consistent – Version 1*.

Assume the log contents shown below (the sequence is ordered left to right, top to bottom) when a system failure occurs. Assume the log entries are in the format (W, Trid, Variable, Old value, New value) and for simplicity the variables are pages with an integer value:

(BEGIN T1)	(W, T1, X, 5, 10)	(BEGIN T2)
(W, T2, X, 10, 20)	(COMMIT T1)	(W, T2, Y, 30, 60)
(CKP {T2})	(W, T2, W, 35, 70)	(BEGIN T3)
(W, T3, Z, 60, 40)	(COMMIT T2)	

1. Is it possible for the log to have the above contents? Explain briefly. If the answer is yes, give a possible sequence of actions of a possible schedule based on the log. If the answer is no, remove the first "impossible" log entry and repeat the process until you get a possible sequence of log entries.
2. For the sequence of entries you got in the previous point, what are the possible values of X, Y, W and Z after the last of these records is written to the permanent memory and before recovery.

Chapter 11

IMPLEMENTATION OF RELATIONAL OPERATORS

This chapter begins the analysis of problems that arise in the implementation of the ***Query Manager***, one of the critical components in any database system. The focus is on the algorithms for evaluating relational algebra operations, and on how to estimate both their execution cost and result size. The next chapter will show how the query optimizer uses these results to choose a query execution plan.

11.1 Assumptions and Notation

To simplify the presentation, we will study the problem under certain restrictive assumptions about the *physical data organization*, the *cost model*, the *selectivity factor* of the selection conditions and the *available statistics*. A simple database schema used for the examples is also presented.

11.1.1 Physical Data Organization

Let us assume that each relation has attributes without *null* values, and it is stored in a heap file, or with the primary organization *index sequential*.

To make the operations on relations more efficient, indexes on one or more attributes are used. The indexes are organized as a B^+–tree and those for non-key attributes are inverted indexes, with sorted RID lists. We also distinguish two types of indexes: *clustered* and *unclustered*.

A clustered index is built on one or more attributes of a relation sorted on the index key. With a B^+–tree index, accessing the data with a scan of the leaf nodes does not involve repeated access to the same relation page. A relation can only have one clustered index.

An unclustered index is built on one or more attributes which are not used to sort a relation. With a B^+–tree index, unlike the previous case, accessing the data with a scan of the leaf nodes causes random access to the relation pages, which can be visited more than once at different times.[1]

1. In the DBMS literature there is no uniform terminology. Sometimes an index built on the primary key is called a *primary index*, regardless of whether or not it is a clustered index, while sometimes the index is called *primary* when it is clustered, even if it is not built on the primary key. Similarly, *secondary* indexes can be those built either on non-unique attributes, or on a primary key which is

11.1.2 Physical Query Plan Operators

The query optimizer has the task of determining how to execute a query in an "optimal" way, by considering the physical parameters involved, such as the size of the relations, the data organization and the presence of indexes. The problem is particularly difficult because, as we will see later, each relational algebra operator can be implemented in different ways and the query optimizer must use appropriate strategies to estimate the costs of the alternatives and choose the one with lowest cost.

The algorithm chosen by the optimizer to execute a query is represented as a tree (*physical query plan* or *physical plan*). The nodes of this tree are *physical operators*, each of which is a particular implementation of an operator of the relational algebra extended on *multisets* (*bags*) as follows to model SQL queries:

- ***Projection with duplicates***: $\pi^b_X(O)$, with X attributes of O.
- ***Duplicate elimination***: $\delta(O)$.
- ***Sorting***: $\tau_X(O)$, with X attributes of O.
  The operator is used to represent the SQL **ORDER BY** clause, and it returns a list of records, rather than a multiset. The operator is meaningful as the root of a logical plan only.
- ***Multiset union, intersection and difference***: $O_1 \cup^b O_2$, $O_1 \cap^b O_2$, $O_1 -^b O_2$.
  If an element t appears n times in O_1 and m times in O_2, then
  - t appears $n + m$ times in the multiset union of O_1 and O_2:

$$\{1, 1, 2, 3\} \cup^b \{2, 2, 3, 4\} = \{1, 1, 2, 3, 2, 2, 3, 4\}$$

  - t appears $\min(n, m)$ times in the multiset intersection of O_1 and O_2:

$$\{1, 1, 2, 3\} \cap^b \{2, 2, 3, 4\} = \{2, 3\}$$

  - t appears $\max(0, n - m)$ times in the multiset difference of O_1 and O_2:

$$\{1, 1, 2, 3\} -^b \{1, 2, 3, 4\} = \{1\}$$

The extension of the other operators (selection, grouping, product, join) from sets to multisets is obvious.

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

For each physical operator we estimate the data access cost C and the cardinality of its result E_{rec}.[2]

Physical operators, like the operators of relational algebra, return collections of records with a type that depends on the operator.

Physical Query Plan Execution. Physical operators are implemented as *iterators* that produce the records of the result one at a time on request. An iterator behaves as an object with state, and methods like:

not used to sort the file.

2. The number of pages of a physical operator result W is estimated as

$$N_{\mathsf{pag}}(W) = \left\lceil \frac{E_{\mathsf{rec}} \times \sum_{i=1}^{n} L(A_i)}{D_{\mathsf{pag}}} \right\rceil$$

where $A_1, A_2, \ldots, A_n$ are the W attributes.

open	Initializes the process of getting records: initializes the state and calls the *open* function of all its operands.
isDone	Tests if the iterator has more data to return.
next	Returns the next result record.
reset	Re-initializes the iterator already open.
close	Performs some clean-up operations and ends the iterator.

A plan with root Plan is executed according to the scheme:

```
Plan.open();
while not Plan.isDone() do
        print(Plan.next());
Plan.close();
```

For example, the statement Plan.next() produces the following actions in the case of the physical plan in Figure 11.1:

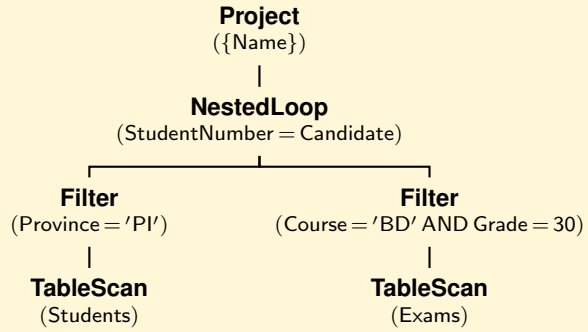


Figure 11.1: Example of a physical query plan

1. The root **Project** operator requests a record to its operand, which implements the join with the **NestedLoop** operator.
2. **NestedLoop** requests
   (a) a record to the left operand, the operator **Filter**, which in turn requests a record to the operator **TableScan** until a record satisfying the condition is found;
   (b) a record to the right operand, which in turn requests a record to its operand, and so on. If the record obtained satisfies the join condition, it belongs to the result, otherwise the request is repeated, and so on.

In the DBMS literature, the technique is referred to as a *pipeline*, and the way in which the physical query plan is executed is called *demand driven*.

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

11.1.3 Cost Model

The cost estimate C of executing a physical operator is the number of pages read from or written to the permanent memory to produce the result. For example, a scan of the file R costs $C = N_{\mathsf{pag}}(R)$. In the case of access through an index, the cost is:

$$C = C_I + C_D$$

where C_I is the cost of accessing an index pages to find the RIDs of the records that satisfy the condition, while C_D is the cost of accessing the data pages containing the records. If a B^+–tree index is used, the cost C_I is usually approximated by the cost of accessing the leaf nodes, ignoring the cost of the visit of the path from the root to a leaf node.

11.1.4 Statistics

The values of each attribute are assumed to be uniformly distributed across its active domain.

Regarding the distribution of records with a given value of an attribute, in the pages of the file containing the relation, we distinguish two cases:

- The relation is sorted by that attribute, and therefore the records with the same attribute value are physically adjacent.
- The relation is not sorted on that attribute and records with the same attribute value are uniformly distributed in the pages of the file containing the relation.

We also assume that the distribution of values of an attribute in the relation records is independent of the distribution of values of another attribute of the relation (uniform distribution and independence between the values of different attributes). Finally, we will assume that the database system catalog stores the following statistics information:

1. For each relation $R(A_1, A_2, \ldots, A_n)$:

$N_{\mathsf{rec}}(R)$	the number of records in R.
$N_{\mathsf{pag}}(R)$	the number of pages of R.
c_R	the average number of records in a page of R ($N_{\mathsf{rec}}(R)/N_{\mathsf{pag}}(R)$).
$N_a(R)$	the number of attributes of R.
$L_r(R)$	the size of a record of R in bytes.

2. For each attribute A_i:

L_{A_i}	the average size of an A_i value.
$N_{\mathsf{key}}(A_i)$	the number of A_i distinct values in R ($N_{\mathsf{rec}}(\pi_{A_i}(R))$).
$\min(A_i)$	minimum and maximum A_i values.
$\max(A_i)$	

3. For each index I defined on one or more attributes, called the *index key*:[3]

$N_{\mathsf{key}}(I)$	the number of distinct values of the index key.
$\min(I)$	the minimum and maximum values of the index key.
$\max(I)$	
$N_{\mathsf{leaf}}(I)$	the number of leaf nodes in the index.

As we will see, these statistics are the basic parameters on which the formulas of the costs for the implementation of relational operators will be defined. In relational systems, they are maintained in the system catalog and are used by the optimizer to choose the best physical query plan.

3. The term *index key* should not be confused with *relation key*: the first refers to the attributes of the index, the second to a key of the relation.

The assumption that the distribution of an attribute values is uniform simplifies calculations but it is typically not valid. We will see how to improve accuracy if the system can maintain histograms about the actual distribution of values.

The statistics in the catalog are updated periodically by an appropriate command, such as **UPDATE STATISTICS**, which can be run by any user. As an example, see the structure of the tables in the JRS catalog, which are some of those provided in commercial systems.

A Database Schema for Examples. In the following examples of queries we consider the schema:

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

Underlined attributes are the keys of the three relations.

Table 11.1 lists the physical parameters of the three relations, stored in a heap file with pages of $D_{\text{pag}} = 4$ KB, with unclustered indexes on primary and foreign keys.

Table 11.1: Physical parameters

	R	S	T
N_{rec}	40 000	100 000	20 000
L_r	50 B	40 B	20 B
N_{pag}	500	1000	100
N_{leaf}(IPkR)	120		
N_{leaf}(IPkS)		300	
N_{leaf}(IPkT)			60
N_{leaf}(IFkR)		180	
N_{key}(IFkR)		40 000	
N_{leaf}(IFkT)		140	
N_{key}(IFkT)		20 000	

11.2 Selectivity Factor of Conditions

The *selectivity factor* of a condition, also called *filter* or *reduction factor*, is an estimate of the fraction of records from a relation which will satisfy the condition. Let us consider various cases of conditions and how to estimate their selectivity factors using the approximations proposed for *System R*.

Simple Condition. We use the following notation:

- ψ_{A_i} is a predicate on the attribute A_i of relation R.
- $\text{dom}(A_i)$ is the set of A_i distinct values in R (A_i *active domain*).
- $\min(A_i), \max(A_i)$ are the miminum and maximum value of A_i.
- v, v_i are constants of any type.
- c, c_1, c_2 are numeric constants.

Assuming a uniform distribution of the A_i values in the records of R, the selectivity factor of the predicates of interest is estimated as follows:[4]

4. In all the estimates of the selectivity factor of a predicate, the constants are those suggested by *System R* in case of lack of information or for expressions with non-numeric attributes. For the use that we make of these estimates no distinction is made between $<$ and $\leq$ as well as between $>$ and $\geq$.

$$
\begin{aligned}
s_f(A_i = v) &= \begin{cases} \dfrac{1}{N_{\text{key}}(A_i)} & \\ 1/10 & \end{cases} \\
s_f(A_i > c) &= \begin{cases} \dfrac{\max(A_i) - c}{\max(A_i) - \min(A_i)} & \text{if } \min(A_i) < c < \max(A_i) \\ 0 & \text{if } c \geq \max(A_i) \\ 1 & \text{if } c \leq \min(A_i) \\ 1/3 & \end{cases} \\
s_f(A_i < c) &= \begin{cases} \dfrac{c - \min(A_i)}{\max(A_i) - \min(A_i)} & \text{if } \min(A_i) < c < \max(A_i) \\ 0 & \text{if } c \leq \min(A_i) \\ 1 & \text{if } c \geq \max(A_i) \\ 1/3 & \end{cases} \\
s_f(c_1 < A_i < c_2) &= \begin{cases} \dfrac{c_2 - c_1}{\max(A_i) - \min(A_i)} & \\ 1/4 & \end{cases} \\
s_f(A_i = A_j) &= \begin{cases} \dfrac{1}{\max(N_{\text{key}}(A_i), N_{\text{key}}(A_j))} & \text{if } \text{dom}(A_i) \subseteq \text{dom}(A_j) \vee \text{dom}(A_j) \subseteq \text{dom}(A_i) \\ \dfrac{1}{N_{\text{key}}(A_i)} & \text{if only one of the two } N_{\text{key}}(A_i) \text{ is known} \\ 0 & \text{if } \text{dom}(A_i) \cap \text{dom}(A_j) = \emptyset \\ 1/10 & \end{cases} \\
s_f(A_i \in \{ v_1, \ldots, v_n \}) &= \begin{cases} n \times s_f(A_i = v) & \text{if less than 1/2} \\ 1/2 & \end{cases}
\end{aligned}
$$

In the absence of information, the values of the constants are only intended to express that we assume a larger selectivity (and therefore less $s_f(\psi_{A_i})$) for an equal-

ity predicate; and a lesser selectivity (and therefore greater $s_f(\psi_{A_i})$) for comparison predicates. As in *System R*, we assume that the selectivity of the predicate $s_f(A_i \in \{ v_1, \ldots, v_n \})$ is not greater than $1/2$.

Negation. The records satisfying the condition $\neg\psi$ ($s_{\neg\psi}$) are those that do not satisfy ψ:

$$s_{\neg\psi} = N_{\mathsf{rec}}(R) - s_\psi$$

$$s_f(\neg\psi) = (N_{\mathsf{rec}}(R) - s_\psi)/N_{\mathsf{rec}}(R) = 1 - s_f(\psi)$$

Conjunction. The conjunctive condition $\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n$ is satisfied by the records which satisfy all the conditions ψ_i. Assuming that the conditions ψ_i are independent of each other:

$$s_f(\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n) = s_f(\psi_1) \times s_f(\psi_2) \times \cdots \times s_f(\psi_n)$$

Disjunction. The disjunctive condition $\psi_1 \vee \psi_2 \vee \cdots \vee \psi_n$ is satisfied by the records which satisfy at least one of the conditions ψ_i.

The disjunctive condition can be transformed with the De Morgan rules:

$$\psi_1 \vee \psi_2 \vee \cdots \vee \psi_n = \neg(\neg\psi_1 \wedge \neg\psi_2 \wedge \cdots \wedge \neg\psi_n)$$

and then, because of the two previous cases, and assuming that the conditions ψ_i are independent of each other:

$$s_f(\psi_1 \vee \psi_2 \vee \cdots \vee \psi_n) = 1 - ((1 - s_f(\psi_1)) \times (1 - s_f(\psi_2)) \times \cdots \times (1 - s_f(\psi_n)))$$

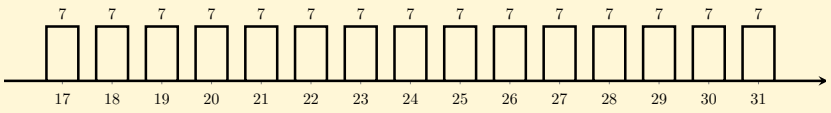
In particular, in the case of disjunction of two conditions:

$$s_f(\psi_1 \vee \psi_2) = 1 - ((1 - s_f(\psi_1)) \times (1 - s_f(\psi_2))) = s_f(\psi_1) + s_f(\psi_2) - s_f(\psi_1) \times s_f(\psi_2)$$

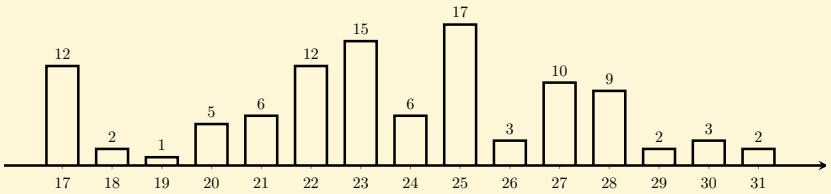
Using Histograms

The selectivity factor of a predicate can be estimated more accurately by knowing the actual distribution of the attribute values instead of assuming a uniform distribution between the minimum and maximum values. For instance, let us consider a relation E with 105 records, and two cases of distribution for an integer-valued attribute A that takes values in the range 17 to 31 (Figure 11.2).

The selectivity factor of the predicate $A > 29$ is estimated $2/14 = 0.1428$ under the assumption of uniform distribution, and $5/105 = 0.0476$ given the actual distribution. The advantage of uniform distribution is that it requires storing in the catalog only three values ($\min(A)$, $\max(A)$ and N_{rec}), while the actual distribution is more expensive to store. The solution preferred by the DBMSs is to use a histogram of values as an approximation of the actual distribution.



(a) *Uniform distribution*



(b) *Nonuniform distribution*

Figure 11.2: Uniform and nonuniform distributions of values

The histogram can be of different types and the most used one is the *Equi-Height*: the active domain of the attribute A is divided into k intervals containing a number of records of about N_{rec}/k (Figure 11.3).

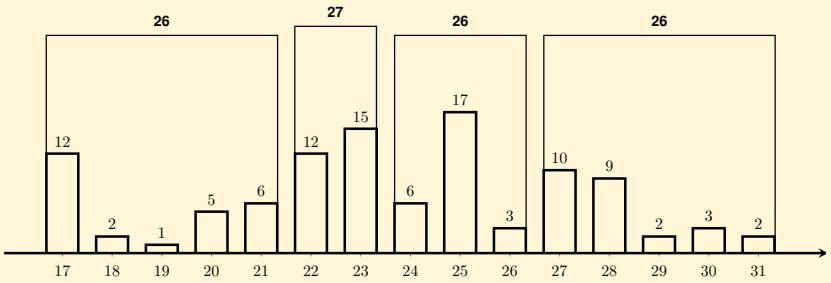


Figure 11.3: Equi-Height histogram

Example 11.1
Let us see how to use histograms to estimate the selectivity of a predicate. Let $h_i, i = 1..k$, the k histogram intervals of A values. For each of them are known

1. the minimum and maximum value ($\min(h_i)$, $\max(h_i)$),
2. the number of records ($N_{\mathsf{rec}}(h_i)$) and
3. the number of attribute values ($N_{\mathsf{key}}(h_i)$).

The selectivity factor of a predicate on A is estimated as $s_f(\psi_A) = N_{\mathsf{rec}}(\psi_A)/N_{\mathsf{rec}}$, with $N_{\mathsf{rec}}(\psi_A)$ an estimate of the number of records that satisfy

the predicate. Let us see some estimation of $N_{\mathsf{rec}}(\psi_A)$, with c, c_1 and c_2 three A values:

1. $N_{\mathsf{rec}}(A = c) = \dfrac{N_{\mathsf{rec}}(h_i)}{N_{\mathsf{key}}(h_i)}$, if $c \in h_i$.
2. $N_{\mathsf{rec}}(c_1 \leq A \leq c_2) = (c_2 - c_1 + 1) \times \dfrac{N_{\mathsf{rec}}(h_i)}{N_{\mathsf{key}}(h_i)}$, if $c_1 \in h_i \wedge c_2 \in h_i$ (*partial range*).
3. $N_{\mathsf{rec}}(c_1 \leq A \leq c_2) = N_{\mathsf{rec}}(h_i)$, if $c_1 = \min(h_i) \wedge c_2 = \max(h_i)$ (*total range*).
4. If c_1 and c_2 belong to different histogram intervals, the estimate of $N_{\mathsf{rec}}(c_1 \leq A \leq c_2)$ is calculated by adding the estimates for the total and partial ranges concerned.

Let us estimate the selectivity factor of the following predicates on the attribute A of the relation E with 105 records.

1. $s_f(A = 18)$
   (a) *Actual distribution*.
   $$s_f(A = 18) = 2/105 = 0.019$$
   (b) *Uniform distribution*.
   $$s_f(A = 18) = 1/15 = 0.0666$$
   (c) *Equi-Height histogram*. The value 18 is in the histogram interval $[17, 21]$ with 26 records.
   $$s_f(A = 18) = N_{\mathsf{rec}}(h_i)/(N_{\mathsf{key}}(h_i) \times N_{\mathsf{rec}}) = 26/(5 \times 105) = 0.0495$$
2. $s_f(21 \leq A \leq 27)$
   (a) *Actual distribution*.
   $$s_f(21 \leq A \leq 27) = 69/105 = 0.6571$$
   (b) *Uniform distribution*.
   $$s_f(21 \leq A \leq 27) = 7/15 = 0.4667$$
   (c) *Equi-Height histogram*. The range $21 \leq A \leq 27$ is partially contained in the histogram intervals $[17, 21]$ with 26 records and $[27, 31]$ with 26 records, and fully contained in the intervals $[22, 23]$ with 27 records and $[24, 26]$ with 26 records.
   $$s_f(21 \leq A \leq 27) = (1 \times 26/5 + 1 \times 26/5 + 27 + 26)/105 = 0.6038$$

11.3 Physical Operators for Relation (R)

The records of a relation can be retrieved with any of the following operators, which have a relation R as an argument, and so they can only be a leaf of a physical query plan.

TableScan(R)
The operator returns the records of R, in the order that they are stored, and has the cost

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

SortScan$(R, \{A_i\})$
The operator returns the records of R sorted in ascending order on the attribute $\{A_i\}$ values. For simplicity, we ignore the possibility of also specifying for each attribute a DESC clause to sort in descending order.

Sorting is done with the *merge sort* algorithm. In general, the operator's cost depends on the $N_{\mathsf{pag}}(R)$ value, the number of pages B available in the buffer (*sort heap size*), and the implementation of *merge sort*.

If the *merge sort* is implemented so that it returns the final result of the merge without first writing it to a temporary file (*piped merge sort*), then the cost of **SortScan** is

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

For simplicity, in the following examples we assume that $N_{\mathsf{pag}}(R) \leq B \times (B-1)$, and so the sorting requires at most one merge phase. This assumption is realistic with current and expected sizes of the main memory of computers.

IndexScan(R, I)
The operator returns the records of R sorted in ascending order on the attribute $\{A_i\}$ values of the index I, with a cost that depends on the type of index and the type of attribute on which it is defined.

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

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

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

Estimating the cardinality of the result. In all the cases the cardinality of the result is

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

11.4 Physical Operator for Projection (π^b)

The physical operator that implements projection, without duplicate elimination, has as first argument the records returned by O, another physical operator.

Project$(O, \{A_i\})$
The operator returns the records of the projection of the records of O over the attributes $\{A_i\}$, with the cost

$$C = C(O)$$

If the argument O of the physical operator is a relation indexed on the projected attributes $\{A_i\}$, the projection can have a more efficient evaluation by means of following operator.

IndexOnlyScan$(R, I, \{A_i\})$
The operator returns the sorted records of $\pi^b_{\{A_i\}}(R)$ using the index I on the attributes $\{A_i\}$, without accessing the relation, with the cost

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

The index can also be used if it is defined on a set of attributes that contains $\{A_i\}$ as a prefix. The result is without duplicates if the attributes $\{A_i\}$ include a relation key, otherwise, if a tuple of values for the attributes $\{A_i\}$ is associated to n different RIDs, it is returned n times.

Estimating the Cardinality of the Result. The cardinality of the result is estimated as follows.

- **Project**$(O, \{A_i\})$

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

- **IndexOnlyScan**$(R, I, \{A_i\})$

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

11.5 Physical Operators for Duplicate Elimination (δ)

The physical operators for duplicate elimination have as argument the records returned by O, another physical operator.

Distinct(O)
The records of O *must be sorted* so that duplicates will be next to each other. The operator returns the records of O sorted, without duplicates, i.e. it converts a multiset to a set, with the cost

$$C = C(O)$$

HashDistinct(O)
The operator returns the records of O without duplicates using a hash technique.

Let us assume that the query processor has $B + 1$ pages in the buffer to perform duplicate elimination. There are two phases that use two different hash functions h_1, h_2: *partitioning* and *duplicate elimination*.

In the *partitioning* phase, for each record of O the hash function h_1 is applied to all attributes to distribute the records uniformly in the B pages. When a page i is full, it is written to the T_i partition file. At the end of the partitioning phase we have a partition of the records of O in B files, each of which contains a collection of records that share a common hash value (Figure 11.4). Possible duplicates are in the same partition. Let us assume that each partition has at most B pages.

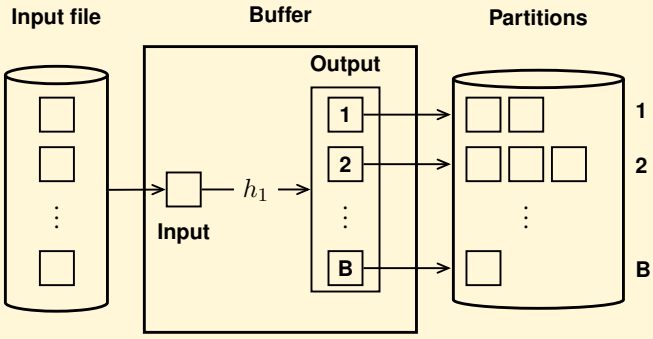


Figure 11.4: Partitioning Phase

In the *duplicate elimination* phase, the process becomes an intra-partition problem only: each T_i partition is read page-by-page to eliminate duplicates with the hash function h_2 applied to all record attributes. A record r is discarded only when it collides with another record r' with respect to h_2, and $r = r'$. The remaining record are included in the result. Then the contents of the B pages is cleared, and the duplicate elimination is applied to the records of the next partition.

If the number of pages of a partition is greater than B, the hash-based projection technique is applied recursively by dividing the partition into subpartitions, and this degrades performance.

The operator has the cost

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

From this analysis follows that the elimination of duplicates with this technique has the same cost of **Distinct** with the sorting of the operand records, and has the disadvantage of not producing a sorted result. However, the performance of **HashDistinct** can be improved with another technique, called *hybrid hashing*, to exploit extra buffer space to keep in memory the first partition, without duplicates, during the partitioning phase rather than saving it.

Estimating the Cardinality of the Result. The cardinality of the **Distinct**(O) and **HashDistinct**(O) result is estimated as follows.

– If A_i is the only O attribute, with $N_{\mathsf{key}}(A_i)$ distinct values, then

$$E_{\mathsf{rec}} = N_{\mathsf{key}}(A_i)$$

Projection in DBMS. To eliminate duplicates, Informix uses hashing. DB2, Oracle and Sybase ASE use sorting. Microsoft SQL Server and Sybase ASIQ use both hashing and sorting. JRS uses sorting.

– If $\{A_1, A_2, \ldots, A_n\}$ are the O attributes, and A_i is an indexed attribute in R then $s_f(A_i) = N_{\mathsf{key}}(A_i)/N_{\mathsf{rec}}(R)$, otherwise $s_f(A_i) = 1/10$:

$$E_{\mathsf{rec}} = \lceil E_{\mathsf{rec}}(O) \times \prod s_f(A_i) \rceil$$

11.6 Physical Operators for Sort (τ)

To sort the result of an operand O, the following operator is used.

Sort$(O, \{A_i\})$
As with the **SortScan**, we assume that the operator sorts the records of O on the attributes $\{A_i\}$ with the *piped merge sort* algorithm and B buffer pages. The cost is:

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

In the following we assume, for simplicity, that $N_{\mathsf{pag}}(O) \leq B \times (B-1)$, and so the sorting requires at most one merge phase.

Estimating the cardinality of the result. The cardinality of the result is

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

.

11.7 Physical Operators for Selection (σ)

Physical operators for the selection are of different types: without the use of indexes or with the use of one or more indexes.

Filter(O, ψ)
The operator returns the records of O satisfying the condition ψ, with the cost

$$C = C(O)$$

.

IndexFilter(R, I, ψ)
The operator returns the records of R satisfying the condition ψ with the use of the index I, defined on attributes of ψ. The condition ψ is a predicate or a conjunction of predicates that only involve attributes in a prefix of the index search key. The result is in the index search key sort order.

The operator does two things: it uses the index to find the sorted set of RIDs of the records satisfying the condition, and then retrieves the records of R.

These two operations may be performed with two physical operators, as it happens in commercial DBMSs:

- **RIDIndexFilter**(I, ψ), which returns a sorted set of RIDs, and
- **TableAccess**(O, R), which returns the records of R with the RIDs in O.

Here we combine the two operators into one — **IndexFilter**(R, I, ψ) — to simplify the physical query plans.

The first argument of an **IndexFilter** is the relation R on which the index is defined, therefore **IndexFilter** can only be a leaf of a physical query plan.

The operator has the cost

$$C = C_I + C_D$$

that depends on the type of index and the type of attribute on which it is defined.

- If the index is clustered

  $$C_I = \lceil s_f(\psi) \times N_{\mathsf{leaf}}(I) \rceil$$

  with $s_f(\psi)$ the fraction of the leaf nodes in the B^+–tree which will be visited to find the RIDs of records that satisfy the selection predicate.

  $$C_D = \lceil s_f(\psi) \times N_{\mathsf{pag}}(R) \rceil$$

  since the access to data by means of the RIDs in the index results in an ordered access to the relation pages, so a data page will not be visited more than once.
- If the index is unclustered

  $$C_I = \lceil s_f(\psi) \times N_{\mathsf{leaf}}(I) \rceil$$

  $$\begin{aligned} C_D &= \mathsf{NoListToVisit} \times \mathsf{NoPageToVisitForList} \\ &= \lceil s_f(\psi) \times N_{\mathsf{key}}(I) \rceil \times \Phi(\lceil N_{\mathsf{rec}}(R)/N_{\mathsf{key}}(I) \rceil, N_{\mathsf{pag}}(R)) \end{aligned}$$

  where Φ is the Cardenas formula.
  If the index is defined on a key of R

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

IndexSequentialFilter(R, I, ψ)
The operator returns the sorted records of R, stored with the primary organization *index sequential* I, satisfying the condition ψ, a predicate or a conjunction of predicates that involve only attributes of the index search key.

The operator has the cost

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

IndexOnlyFilter$(R, I, \{A_i\}, \psi)$
The operator returns the sorted records of $\pi^b_{\{A_i\}}(\sigma_\psi(R))$, using *only* the index I. The condition ψ is a predicate or a conjunction of predicates that only involve attributes in a prefix of the index search key.

The operator has the cost

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

The result is without duplicates if the attributes $\{A_i\}$ include a relation key.

OrIndexFilter$(R, \{I_i, \psi_i\})$
The operator returns the records of R satisfying the disjunctive condition $\psi = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_n$ using all the indexes I_i, one for each term ψ_i.

The operator performs two operations: it uses the indexes to find a sorted union of the RID lists of the records matching each terms ψ_i, and then retrieves the records of R.

Let n be the number of indexes used, and C_I^k the access cost of the k-th index, we obtain:

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

The number of data page accesses is calculated taking into account the estimated number E_{rec} of records to retrieve:

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

Since the union of the RID lists is sorted:

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

AndIndexFilter$(R, \{I_i, \psi_i\})$
The operator returns the records of R satisfying the conjunctive condition $\psi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n$ using the indexes I_i, one for each term ψ_i.

The operator performs two operations: it uses the indexes to find a sorted intersection of the RID lists of the records matching each terms ψ_i, and then retrieves the records of R using the RIDs.

Let n be the number of indexes used, and C_I^k the access cost of the k-th index, we obtain:

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

The number of data page accesses is calculated taking into account the estimated number E_{rec} of records to retrieve:

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

Since the intersection of the RID lists is sorted:

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

Selection in DBMS. To make a conjunctive selection commercial DBMSs use different algorithms for RID set intersection. Oracle uses bitmaps operations or a *hash join* (see below) of indexes on the values of the RID lists that satisfy the selection condition. Microsoft SQL Server uses join indexes (see below). DB2 uses Bloom filters. Sybase ASE uses one index only and does not do RID set intersection, while Sybase ASIQ uses bitmap operations. Informix does RID set intersection, JRS uses one index only.

To make a disjunctive selection, usually bitmaps are used. Oracle eliminates the OR by rewriting the query using the IN or UNION operators.

Estimating the cardinality of the result. The cardinality of the result depends on the kind of operators and on the selectivity factor of the condition $s_f(\psi)$:

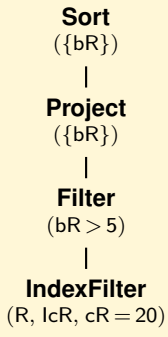
$$E_{\text{rec}} = \begin{cases} \lceil s_f(\psi) \times E_{\text{rec}}(O) \rceil & \text{if the operator is } \textbf{Filter} \\ \lceil s_f(\psi) \times N_{\text{rec}}(R) \rceil & \text{otherwise} \end{cases}$$

Example 11.2
Let us consider the query

```
SELECT   bR
FROM     R
WHERE    bR > 5 AND cR = 20
ORDER BY bR;
```

and the cost of some physical plans to execute it.

1. Let us assume that there is only one unclustered index on cR.



$$\begin{aligned} C_{\text{Sort}} &= C_O + 2 \times N_{\text{pag}}(O) \\ &= C_{\text{IndexFilter}} + 2 \times \left\lceil \frac{L_{bR} \times E_{\text{rec}}(\text{Filter})}{D_{\text{pag}}} \right\rceil \end{aligned}$$

where:

$$\begin{aligned} s_f(cR = 20) &= \frac{1}{N_{\text{key}}(\text{IcR})} \\ C_{\text{IndexFilter}} &= C_I + C_D \\ E_{\text{rec}}(\text{Filter}) &= \lceil s_f(cR = 20) \times s_f(bR > 5) \times N_{\text{rec}}(R) \rceil \end{aligned}$$

2. Let us assume that there are two unclustered indexes on bR and cR.

Sort
({bR})
|
Project
({bR})
|
AndIndexFilter
(R, {IcR, cR = 20}, {IbR, bR > 5})

$$
\begin{aligned}
C_{\text{Sort}} &= C_O + 2 \times N_{\text{pag}}(O) \\
&= C_{\text{AndIndexFilter}} + 2 \times \left\lceil \frac{L_{bR} \times E_{\text{rec}}(\text{Project})}{D_{\text{pag}}} \right\rceil \\
C_{\text{AndIndexFilter}} &= \lceil s_f(bR > 5) \times N_{\text{leaf}}(\text{IbR}) \rceil + \lceil s_f(cR = 20) \times N_{\text{leaf}}(\text{IcR}) \rceil + \\
&\quad \Phi(\lceil s_f(bR > 5 \wedge cR = 20) \times N_{\text{rec}}(R) \rceil, N_{\text{pag}}(R)) \\
E_{\text{rec}}(\text{Project}) &= E_{\text{rec}}(\text{AndIndexFilter}) = \lceil s_f(bR > 5 \wedge cR = 20) \times N_{\text{rec}}(R) \rceil
\end{aligned}
$$

3. Let us assume that there is only one unclustered index on {bR, cR}.

IndexOnlyFilter
(R, IbRcR, {bR}, bR > 5 ∧ cR = 20)

$$
C_{\text{IndexOnlyFilter}} = \lceil s_f(bR > 5 \wedge cR = 20) \times N_{\text{leaf}}(\text{IbRcR}) \rceil
$$

11.8 Physical Operators for Grouping (γ)

The result of ${}_{\{A_i\}}\gamma_{\{f_i\}}(R)$ is a set of records with attributes $\{A_i, f_i\}$, obtained as follows.

The records of R are partitioned according to their values in one set of attributes $\{A_i\}$, called the *grouping attributes*. Then, for each group, the values in certain other attributes are aggregated with the functions $\{f_i\}$. The result of this operation is one record for each group. Each record has the grouping attributes, with the values common to records of that group, and an attribute for each aggregation, with the aggregated value for that group.

To execute the grouping of the operand O result, the following physical operators are used, which differ in the way the records of O are partitioned.

GroupBy$(O, \{A_i\}, \{f_i\})$
The records of O must be *sorted* on the grouping attributes $\{A_i\}$, so that the records of each group will be next to each other.

The operator returns the records sorted on $\{A_i\}$ and has the cost

$$C = C(O)$$

HashGroupBy$(O, \{A_i\}, \{f_i\})$
The records of O are partitioned using a hash function on the attributes $\{A_i\}$, proceeding in two phases as in the case of the operator **HashDistinct**. In the *partitioning* phase a partition is created using the hash function h_1, while in the second phase, renamed *grouping*, the records of each partition are grouped with the hash function h_2 applied to all grouping attributes and the records of the result are returned: when two records with the same grouping attributes are found, a step to compute the aggregate functions is applied.

The result is not sorted on the grouping attributes $\{A_i\}$ and the operator has the cost

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

Estimating the cardinality of the result. The cardinality of the result is estimated as in the case of duplicate elimination.[5]

11.9 Physical Operators for Join $(\bowtie)$

We will consider here only the equijoin $(O_E \overset{\bowtie}{_{\psi_J}} O_I)$ computed with one of the following physical operators, where O_E is the external operand, O_I is the internal operand, and ψ_J the join condition. In general, the result is a multiset of records, and each of them is the concatenation $< r, s >$ of a record r of O_E and one s of O_I, that is they have the attributes of both r and s.

NestedLoop(O_E, O_I, ψ_J)
The operator can be used regardless of what the join condition is, and it computes the join result with the following algorithm,

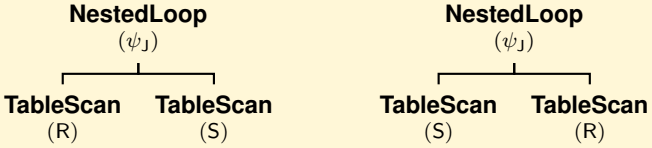
```
for each r ∈ O_E do
   for each s ∈ O_I do
         if ψ_J then add < r, s > to the result;
```

The operator has the cost

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

The order of the loops matters.

Example 11.3
Let us consider two physical query plans to perform a join, with R or S external:



5. Note that $\delta(\pi^b_{A_1,A_2,\ldots,A_n}(R))$ is equivalent to ${}_{A_1,A_2,\ldots,A_n}\gamma(R)$.

The cost of the two physical query plans are

$$N_{\mathsf{pag}}(R) + N_{\mathsf{rec}}(R) \times N_{\mathsf{pag}}(S)$$
$$N_{\mathsf{pag}}(S) + N_{\mathsf{rec}}(S) \times N_{\mathsf{pag}}(R)$$

The two quantities are in general different. Indeed, with R as external the following approximation holds

$$N_{\mathsf{pag}}(R) + N_{\mathsf{rec}}(R) \times N_{\mathsf{pag}}(S) \approx N_{\mathsf{rec}}(R) \times \frac{N_{\mathsf{pag}}(S)}{N_{\mathsf{rec}}(S)} \times N_{\mathsf{rec}}(S)$$

and with S as external the following approximation holds

$$N_{\mathsf{pag}}(S) + N_{\mathsf{rec}}(S) \times N_{\mathsf{pag}}(R) \approx N_{\mathsf{rec}}(S) \times \frac{N_{\mathsf{pag}}(R)}{N_{\mathsf{rec}}(R)} \times N_{\mathsf{rec}}(R)$$

Therefore, R as external is better if

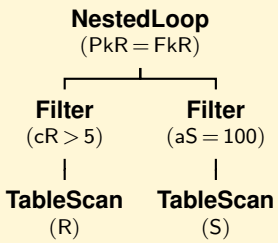
$$\frac{N_{\mathsf{pag}}(S)}{N_{\mathsf{rec}}(S)} < \frac{N_{\mathsf{pag}}(R)}{N_{\mathsf{rec}}(R)}$$

namely whether the capacity of the relation pages are in the relationship $c_S > c_R$, i.e. if R has the longest record.

Example 11.4
Let us consider the relations R and S stored in heap files with the following characteristics:

$N_{\mathsf{pag}}(R) = 500$, $N_{\mathsf{rec}}(R) = 40\,000$, $N_{\mathsf{key}}(\mathsf{IPkR}) = 40\,000$
$N_{\mathsf{pag}}(S) = 1000$, $N_{\mathsf{rec}}(S) = 100\,000$, $N_{\mathsf{key}}(\mathsf{IFkR}) = 40\,000$

and the physical query plan



Let us assume that $s_f(\mathsf{aS = 100}) = 1/100$ and $s_f(\mathsf{cR} > 5) = 1/2$. The plan has the cost

$$\begin{aligned} C_{\mathsf{NL}} &= C(O_E) + E_{\mathsf{rec}}(O_E) \times C(O_I) \\ &= N_{\mathsf{pag}}(R) + N_{\mathsf{rec}}(R)/2 \times N_{\mathsf{pag}}(S) \\ &= 500 + 20\,000 \times 1000 = 20\,000\,500 \end{aligned}$$

The estimation of the plan cost shows that the **NestedLoop** does not benefit from the fact that there is a selection on the result of the internal operand.

In this case we might think of another physical operator, called **NestedLoop-Mat**, which computes the join result with the following algorithm:

The result of the internal **Filter**(O, ψ) is stored in a temporary file T;
for each $r \in O_E$ **do**
  for each $s \in T$ **do**
    if ψ_J **then** add $< r, s >$ to the result;

The estimated physical query plan cost will be quite different

$$C_{\mathsf{NLM}} = C(O_E) + (C(O) + N_{\mathsf{pag}}(T)) + E_{\mathsf{rec}}(O_E) \times N_{\mathsf{pag}}(T)$$

$$C_{\mathsf{NLM}} = 500 + (1000 + 10) + 20\,000 \times 10 = 201\,510$$

PageNestedLoop(O_E, O_I, ψ_J)

The cost of the nested loop join can be reduced considerably if, instead of scanning the result of O_I once per record of O_E, we scan it once per page of O_E with the following algorithm:

for each page p_r of O_E **do**
  for each page p_s of O_I **do**
    for each $r \in p_r$ **do**
      for each $s \in p_s$ **do**
        if ψ_J **then** add $< r, s >$ to the result;

The operator has the cost

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

If the operands are **TableScan** of R and S, the operator has the cost

$$C_{\mathsf{PNL}} = N_{\mathsf{pag}}(R) + N_{\mathsf{pag}}(R) \times N_{\mathsf{pag}}(S)$$

while, exchanging the roles of R and S, we obtain:

$$C_{\mathsf{PNL}} = N_{\mathsf{pag}}(S) + N_{\mathsf{pag}}(S) \times N_{\mathsf{pag}}(R)$$

The two formulas differ in the first term, and so the *algorithm cost is lower when the external relation is the one with fewer pages*.

This method is better than the **NestedLoop**, being $N_{\mathsf{pag}}(R) < N_{\mathsf{rec}}(R)$, but it has the defect of producing an unsorted result due to the ordered scanning of the external relation.

BlockNestedLoop(O_E, O_I, ψ_J)

The operator computes the join result by extending the **PageNestedLoop** with the use of more memory for a group of pages of the external operand.

Let us assume that the operands are **TableScan** of R and S, and the query processor has $B + 2$ pages in the buffer to perform the join. B pages are used by the external operand R, 1 page by the internal operand S (*input* page), and the last page is used as output buffer. For each record r of a page group of R, and for each joining record s of a page of S, the $< r, s >$ is written to the *output* buffer page (Figure 11.5).

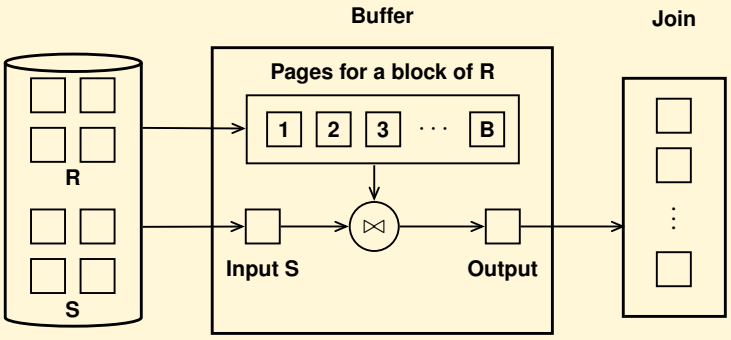


Figure 11.5: *Block nested loop*

The external relation R is read once with a cost $N_{\text{pag}}(R)$. The internal relation S is read $\lceil N_{\text{pag}}(R)/B \rceil$ times with a cost $N_{\text{pag}}(S)$, and so the total cost is

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

Again, the *algorithm cost is lower when the external relation is the one with fewer pages*.

If the B pages are enough to contain one of the two relations, then the cost is reduced to $N_{\text{pag}}(R) + N_{\text{pag}}(S)$.

IndexNestedLoop(O_E, O_I, ψ_J)

The operator requires that

- the join is an *equi-join* ($\psi_J = (O_E.A_i = O_I.A_j)$);
- the internal operand leaf is an **IndexFilter**(S, I, ψ_J), with S a relation and I an index on the join attribute A_j of S.

The operator computes the join result with the following algorithm:

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

The operator has the cost

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

where $(C_I + C_D)$ is the cost to retrieve the matching records of S with a record of O_E, that depends on the index type (whether it is clustered or not) and on the fact that the join attribute of S is a key or a foreign key, bearing in mind that the condition is an equality predicate.

MergeJoin(O_E, O_I, ψ_J)

The operator requires that

- the join is an *equi-join* ($\psi_J = (O_E.A_i = O_I.A_j)$);
- O_E and O_I records are *sorted* on the join attributes $O_E.A_i$ and $O_I.A_j$;
- in the join condition $O_E.A_i$ is a key of O_E.

Since the join attribute of O_E has distinct values, the algorithm proceeds by reading the records of both operands for increasing values of the join attribute only once (Figure 11.6).

```
r := first record of O_E; // r = null if O_E is empty
s := first record of O_I; // s = null if O_I is empty
// let succ(w) the next record of w in W
// or the value null if w is the last one;
while not r == null and not s == null do
        if r[A_i] = s[A_j]
        then(
                add < r, s > to the result;
                s := succ(s);
             )
        else r := succ(r);
```

Figure 11.6: Merge-join with join attribute a key of O_E

The operator has the cost

$$C_{\mathsf{MJ}} = C(O_E) + C(O_I)$$

The result of the physical operators **NestedLoop**, **IndexNestedLoop** and **MergeJoin** is sorted on the attributes of O_E as the records of O_E.

HashJoin(O_E, O_I, ψ_J)

The operator computes the join result with a hash technique in two phases with the algorithm in Figure 11.7.

```
// Partition R in B partitions, flushed as page fills.
for each r ∈ R do
    add r to the page buffer h_1(r[A_i]);
// Partition S in B partitions, flushed as page fills.
for each s ∈ S do
    add s to the page buffer h_1(s[A_j]);
// Probing Phase
for i = 1,...,B do (
    // Build in the buffer the hash table for R_i
    for each r ∈ group R_i do
        add r to the hash table address h_2(r[A_i]);
    // Read S_i and check if they join with those of R_i
    for each s ∈ group S_i do (
        check the records in hash table address h_2(s[A_j]);
        if r[A_i] = s[A_j]
        then add < r, s > to the result; )
    Clear the hash table to deal with the next partition; )
```

Figure 11.7: Hash Join

In the first phase, called *partitioning* (or *building*), the records of O_E and O_I are partitioned using the function h_1 applied to the join attributes, and it is similar to the partitioning phase for the operator **HashDistinct** (Figure 11.4). O_E records in partition i will only match O_I records in partition i.

In the second phase, called *probing* (or *matching*), for each B_i partition (a) the records of O_E are read and inserted into the buffer hash table with B pages using the function h_2 applied to the join attributes values; (b) the records of O_I are read one page at a time, and which of them join with those of O_E is checked with h_2, and they are added to the result (Figure 11.8).

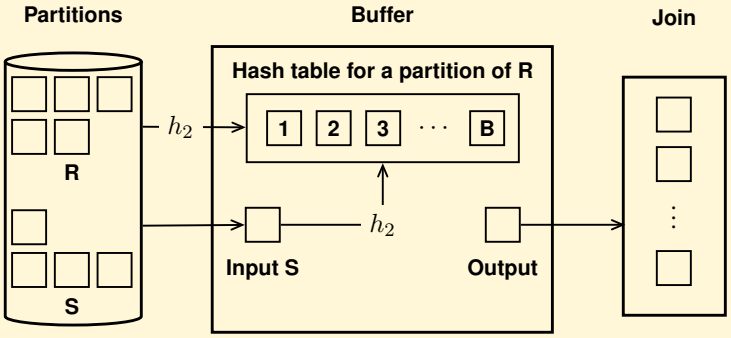


Figure 11.8: Probing Phase of Hash Join

In the partitioning phase, the records of the operands are read and written with a cost

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

In the probing phase, assuming that the hash table for each partition of O_E is held in the buffer, each partition is read once with the cost

$$N_{\mathsf{pag}}(O_E) + N_{\mathsf{pag}}(O_I)$$

Therefore, the operator has a total cost

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

From this analysis follows that **HashJoin** has the same cost of **MergeJoin** with the sorting of the operand records. **MergeJoin** is a good choice if one or both operands are already sorted non join attributes and the result is required to be sorted on join attributes. The **HashJoin** performance can be improved with the *hybrid hash join* technique to exploit extra buffer space to keep in memory the first O_E partition during the partitioning phase rather than saving it. Similarly, while partitioning O_I, rather than write out the first partition, we can directly probe the in–memory table of the first O_E partition and write out the join result. Therefore, we avoid writing the first partitions of O_E and O_I during the partition phase and reading them in again during the probing phase.

Estimating the cardinality of the result. Since

$$R \underset{\psi_J}{\bowtie} S = \sigma_{\psi_J}(R \times S)$$

the cardinality of the result is estimated as

Join in DBMS. The *nested loop* is used by all systems. Sybase ASE also uses the *index nested loop* and the *merge join*, while Sybase ASIQ also uses the *page nested loop* and the *hash join*; Informix, Oracle, DB2 and Microsoft SQL Server use all the algorithms. JRS uses the *page nested loop*, the *index nested loop* and the *merge join*.

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

for the physical operators different from **IndexNestedLoop**, otherwise if the internal operand is an **IndexFilter**(S, I, ψ_J)

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

while if the internal operand is a **Filter(IndexFilter**(S, I, ψ_J)**,** ψ**)**

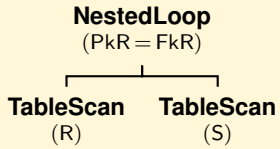
$$E_{\mathsf{rec}} = \lceil s_f(\psi_J) \times E_{\mathsf{rec}}(O_E) \times (s_f(\psi) \times N_{\mathsf{rec}}(S)) \rceil$$

Example 11.5
Let us consider the relations R and S stored in heap files with the following characteristics:

$N_{\mathsf{pag}}(R) = 500$, $N_{\mathsf{rec}}(R) = 40\,000$, $N_{\mathsf{key}}(\mathsf{IPkR}) = 40\,000$, $N_{\mathsf{leaf}}(\mathsf{IPkR}) = 120$
$N_{\mathsf{pag}}(S) = 1000$, $N_{\mathsf{rec}}(S) = 100\,000$, $N_{\mathsf{key}}(\mathsf{IFkR}) = 40\,000$, $N_{\mathsf{leaf}}(\mathsf{IFkR}) = 180$

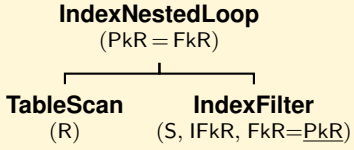
and let us estimate the join cost using different physical query plans.

1. **NestedLoop**



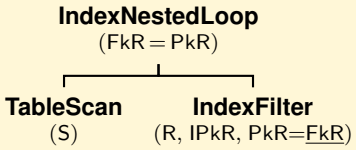
$$\begin{aligned} C_{\mathsf{NL}} &= C(O_E) + E_{\mathsf{rec}}(O_E) \times C(O_I) \\ &= N_{\mathsf{pag}}(R) + N_{\mathsf{rec}}(R) \times N_{\mathsf{pag}}(S) \\ &= 500 + 40\,000 \times 1000 = 40\,000\,500 \end{aligned}$$

2. **IndexNestedLoop**



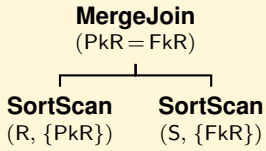
$$
\begin{aligned}
C_{\mathsf{INL}} &= C(O_E) + E_{\mathsf{rec}}(O_E) \times (C_I + C_D) \\
&= N_{\mathsf{pag}}(R) + N_{\mathsf{rec}}(R) \times \\
&\quad (\lceil N_{\mathsf{leaf}}(\mathsf{IFkR})/N_{\mathsf{key}}(\mathsf{IFkR}) \rceil + \Phi(\lceil N_{\mathsf{rec}}(S)/N_{\mathsf{key}}(\mathsf{IFkR}) \rceil, N_{\mathsf{pag}}(S))) \\
&= 500 + 40\,000 \times (\lceil 180/40\,000 \rceil + \lceil \Phi(100\,000/40\,000, 1000) \rceil) \\
&= 160\,500
\end{aligned}
$$

Using S as external



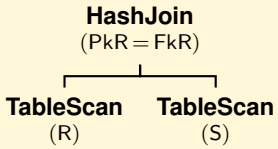
$$
\begin{aligned}
C_{\mathsf{INL}} &= C(O_E) + E_{\mathsf{rec}}(O_E) \times (C_I + C_D) \\
&= N_{\mathsf{pag}}(S) + N_{\mathsf{rec}}(S) \times (C_I + C_D) \\
&= 1000 + 100\,000 \times (1 + 1) = 201\,000
\end{aligned}
$$

3. **MergeJoin**



$$
\begin{aligned}
C_{\mathsf{MJ}} &= C(O_E) + C(O_I) \\
&= C_{\mathsf{Sort}}(R) + C_{\mathsf{Sort}}(S) \\
&= 3 \times N_{\mathsf{pag}}(R) + 3 \times N_{\mathsf{pag}}(S) = 1500 + 3000 = 4500
\end{aligned}
$$

4. **HashJoin**



$$
\begin{aligned}
C_{\mathsf{HJ}} &= C(O_E) + C(O_I) + 2 \times (N_{\mathsf{pag}}(O_E) + N_{\mathsf{pag}}(O_I)) \\
&= 500 + 1000 + 2 \times (500 + 1000) = 4500
\end{aligned}
$$

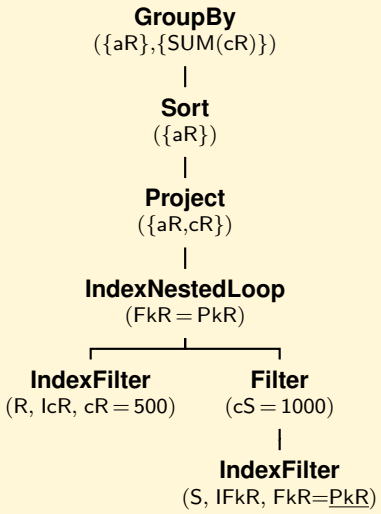
If the table R is small and all its records can be inserted into the hash table, the two tables are read only once and the cost of the hash join becomes

$$C_{\mathsf{HJ}} = C(O_E) + C(O_I) = 500 + 1000 = 1500$$

Example 11.6
Let us consider the relations R and S of the previous example and the query

```
SELECT    aR, SUM(cR)
FROM      R, S
WHERE     PkR = FkR AND cR = 500 AND cS = 1000
GROUP BY  aR
ORDER BY  aR;
```

Let us estimate the cost of the following physical query plan:



$$
\begin{aligned}
C_{\text{GroupBy}} &= C_{\text{Sort}} = C_{\text{IndexNestedLoop}} + 2 \times N_{\text{pag}}(\text{Project}) \\
N_{\text{pag}}(\text{Project}) &= \lceil (E_{\text{rec}}(\text{IndexNestedLoop}) \times (L(\text{aR}) + L(\text{cR})))/D_{pag} \rceil \\
C_{\text{IndexNestedLoop}} &= C(\text{IndexFilterOnR}) + E_{\text{rec}}(\text{IndexFilterOnR}) \times C(\text{Filter}) \\
E_{\text{rec}}(\text{IndexNestedLoop}) &= \lceil s_f(\text{PkR} = \text{FkR}) \times E_{\text{rec}}(\text{IndexFilterOnR}) \times E_{\text{rec}}(\text{Filter}) \rceil
\end{aligned}
$$

where

$$
\begin{aligned}
s_f(\text{PkR} = \text{FkR}) &= 1/max\{N_{\text{key}}(\text{PkR}), N_{\text{key}}(\text{FkR})\} \\
C(\text{IndexFilterOnR}) &= C_I + C_D \\
C_I &= N_{\text{leaf}}(\text{IcR})/N_{\text{key}}(\text{IcR}) \\
C_D &= \Phi(\lceil N_{\text{rec}}(R)/N_{\text{key}}(\text{IcR}) \rceil, N_{\text{pag}}(R)) \\
E_{\text{rec}}(\text{IndexFilterOnR}) &= \lceil N_{\text{rec}}(R)/N_{\text{key}}(\text{IcR}) \rceil \\
C(\text{Filter}) &= C_I + C_D \\
C_I &= N_{\text{leaf}}(\text{IFkR})/N_{\text{key}}(\text{IFkR}) \\
C_D &= \Phi(\lceil N_{\text{rec}}(S)/N_{\text{key}}(\text{IFkR}) \rceil, N_{\text{pag}}(S)) \\
E_{\text{rec}}(\text{Filter}) &= \lceil s_f(\text{cS} = 1000) \times N_{\text{rec}}(S) \rceil
\end{aligned}
$$

11.10 Physical Operators for Set and Multiset Union, Intersection and Difference

Set operators are implemented with the following physical operators, with operands of the same type:

Union(O_E, O_I)**, Except**(O_E, O_I)**, Intersect**(O_E, O_I)
The operators require that the records of the operands are sorted and without duplicates. The operators have the cost

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

HashUnion(O_E, O_I)**, HashExcept**(O_E, O_I)**, HashIntersect**(O_E, O_I)
The operators use a hash technique by applying the same hash function to the results of the two operands. The operators have the cost

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

Multiset operators are implemented with the following physical operators, with operands of the same type:

UnionAll(O_E, O_I)**, ExceptAll**(O_E, O_I)**, IntersectAll**(O_E, O_I)
The operators have the cost

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

The algorithm to implement the multiset union is simple, while those to implement the multiset difference and intersection are more complex and often are not supported by commercial DBMSs, therefore when needed the query must be written in a different way.

Estimating the cardinality of the result.

The **Union** size is estimated as the average of two extreme values: the sum of the sizes of the two operands and the larger size of them. The value is the same as the larger plus half of the smaller:

$$E_{\mathsf{rec}}(\cup) = \max(E_{\mathsf{rec}}(O_E), E_{\mathsf{rec}}(O_I)) + \min(E_{\mathsf{rec}}(O_E), E_{\mathsf{rec}}(O_I))/2$$

The **Except** size is estimated as the average of the O_E size and the $O_E - O_I$ size

$$E_{\mathsf{rec}}(-) = E_{\mathsf{rec}}(O_E) - E_{\mathsf{rec}}(O_I)/2$$

The **Intersect** size is estimated as the average of two extreme values: 0 and the size of the smaller operand

$$E_{\mathsf{rec}}(\cap) = \min(E_{\mathsf{rec}}(O_E), E_{\mathsf{rec}}(O_I))/2$$

The **UnionAll** size is the sum of the sizes of the two operands

$$E_{\mathsf{rec}}(\cup^b) = E_{\mathsf{rec}}(O_E) + E_{\mathsf{rec}}(O_I)$$

The **ExceptAll** and **IntersectAll** sizes are estimated as in the set operator cases.

11.11 Summary

1. The relational algebra operators can be implemented with different algorithms and costs. The choice of the best algorithm depends on the amount of data and the selectivity of selection conditions.
2. The estimation of the selectivity of a condition depends on the physical parameters stored in the system catalog. The hypothesis of uniform distribution and independence between the values of different attributes may not be reasonable in many cases and for this reason the systems maintain more precise information about the type of distribution of values using histograms.
3. The operator for duplicate elimination can be implemented using sorting, hash functions or an index. The selection operator with a conjunctive condition can be implemented with a single index or multiple indexes. If the condition is a disjunction of simple conditions, the indexes can be used only if useful indexes are available for each simple condition.
4. The join of two relations R and S is the most critical relational algebra operator because it is used frequently and has a great impact on the cost of query execution. For these reasons, many algorithms for implementing it have been studied, with a cost which depends on the size of the relations involved and on the pages available in the buffer. The nested loop evaluates the join condition for each pair of records in R and S. Among the possible variants the more interesting is the *index nested loop* that uses an index on the join attribute of internal relation. The *merge join* exploits instead sortedness of the relations on the join attributes. The *hash join* uses a hash function to partition first the records of R and S on the join attributes, and then build pairs of joining records considering the partitions with the same hash function value.
5. The set operators and the grouping can be implemented by exploiting the sortedness of their operands or using a hash technique. The grouping operator, as well as that for projection, can be implemented using an index only and thus avoiding the cost of accessing the data pages to produce the result.

Bibliographic Notes

The implementation of the relational operators is discussed in all books cited in Chapter 1, in particular [Garcia-Molina et al., 1999] and [Ramakrishnan and Gehrke, 2003]. A review of great interest is [Graefe, 1993]. For the use of histograms to estimate the selectivity of the conditions see [Garcia-Molina et al., 1999].

Exercises

Exercise 11.1 Briefly answer the following questions:

1. Define the term *useful index for a query*.
2. Which relational algebra operators can be implemented with a sorting operator.
3. Describe an algorithm for the join implementation and give an estimate of its cost.
4. Compare two algorithms for the join implementation.
5. If the join condition is not equality, which join algorithms cannot be used?

Exercise 11.2 Consider the relation $R(A, B, C, D)$ with key A and the following SQL query. All the attributes have a type string of the same length.

```
SELECT DISTINCT A, B FROM R;
```

1. Estimate the cost of a physical query plan without the use of indexes.
2. Estimate the cost of a physical query plan with the use of a clustered B^+–tree index on B.
3. Estimate the cost of a physical query plan with the use of an unclustered B^+–tree index on A.
4. Estimate the cost of a physical query plan with the use of a multi-attribute clustered B^+–tree index on A, B.

Exercise 11.3 Consider the relation $R(A, B, C, D)$ with key A. All the attributes have a type string of the same length.

Suppose that a B^+–tree inverted index on C is available. Estimate the cost of a physical query plan for the following SQL query:

```
SELECT   *
FROM     R
WHERE    C BETWEEN 'C1' AND 'C10';
```

Exercise 11.4 Consider the relation $R(A, B, C, D)$ with the attributes C and D of type integer. Suppose there is a clustered inverted index on C and an unclustered inverted index on D. Estimate the cost of an access plan for the following SQL query:

```
SELECT   *
FROM     R
WHERE    C = 10 AND D = 100;
```

Exercise 11.5 Consider the relation R(K:int, A:int, B:int) organized as a *sorted file* on the attribute A in N_{pag} pages. R contains N_{rec} records. The attributes A and B have $N_{\text{key}}(A)$ and $N_{\text{key}}(B)$ values.

Suppose that the following indexes are available:

1. An hash index on the primary key K.
2. Two B^+–tree inverted indexes on A and B.

For each of the following relational algebra queries, estimate the cost of an access plan to execute the queries with the use of only one index:

1. $\sigma_{\text{(K isin [k1; k2; k3]) AND (B = 10)}}(R)$
2. $\sigma_{\text{(A = 100) AND (B = 10)}}(R)$.

Exercise 11.6 Consider the relations $R(A, B)$, $S(B, C)$, $T(C, D)$ and the following information about them:

$N_{\text{rec}}(R) = 200, N_{\text{key}}(R.A) = 50, N_{\text{key}}(R.B) = 100$
$N_{\text{rec}}(S) = 300, N_{\text{key}}(S.B) = 50, N_{\text{key}}(S.C) = 50$
$N_{\text{rec}}(T) = 400, N_{\text{key}}(T.C) = 40, N_{\text{key}}(T.D) = 100$

For each of the following relational algebra queries, estimate the size of the results:

1. $(\sigma_{S.B=20}(S)) \bowtie T$;
2. $\sigma_{R.A \neq S.C}(R \bowtie S)$.

Chapter 12

QUERY OPTIMIZATION

The *Relational Engine* of a DBMS manages the SQL commands for the following operations: starting and ending a session; creating and deleting a database; creating, deleting and modifying tables; creating and deleting indexes or views; granting and revoking privileges on data; searching and modifying records of a table retrieved with appropriate queries. In this chapter the focus is on query processing and on how to find the best plan to execute a query. In particular, we study the *optimizer* organization, a fundamental component of the *Query Manager*, which selects the optimal *physical plan* to execute queries using the operators and data structures of the *Storage Engine*. We will also show how functional dependencies, well known for relational schema design, are also important for query optimization.

12.1 Introduction

In general there are several alternative strategies to execute a query, in particular when it is complex, and the optimizer task is to find the best one. The problem falls into the category of "difficult" problems and therefore the optimizer uses heuristic methods to find a good solution quickly. The complexity of the problem is due to the fact that (a) a query can be rewritten in several equivalent ways, and (b) a relational algebra operator can be implemented with different physical operators.

The optimization can be *dynamic* or *static*. In the case of the interactive use of a database, the optimization is always dynamic, that is, the physical plan is generated at run time when a query is executed, taking into account information stored in the system catalog concerning database statistics and the available data structures.

In the case of queries embedded in a program, the optimization can be either dynamic or static. In the first case the physical plan is generated when the query is executed. In the case of a static optimization, the physical query plan is generated at the program compilation time. Therefore, the optimization is performed only once, regardless of the number of times that a query will be executed. A change of the database physical design, for example an index is added or removed, may render a physical query plan invalid and require a new optimization to be generated.

12.1.1 Query Processing Phases

Query processing proceeds in four phases:

1. ***Query analysis***, in which the correctness of the SQL query is checked, and the query is translated into its internal form, which is usually based on the relational algebra (*initial logical query plan*).
2. ***Query transformation***, in which the initial logical plan is transformed into an equivalent one that provides a better query performance.
3. ***Physical plan generation***, in which alternative algorithms for the query execution are considered, using the available physical operators to implement the operators of the logical plan, and the *physical query plan*, also called a *query plan*, with the lowest cost is chosen.
4. ***Query evaluation***, in which the physical plan is executed.

The phases *Query transformation* and *Physical plan generation* are often called *Query optimizer*. The *physical plan generation* phase makes query processing hard, because there is a large number of alternative solutions to consider in order to choose the one with the least estimated cost.

12.2 Query Analysis Phase

During the *query analysis* the following steps are performed:

1. Lexical and syntactic query analysis.
2. Semantic query analysis. During this step the system checks both the query semantic correctness and that the user has appropriate privileges, taking into account the definitions of the relations used and the authorizations contained in the system catalog.
3. Conversion of the **WHERE** condition into conjunctive normal form.
4. Simplification of the condition by applying the equivalence rules of boolean expressions.
5. Elimination of contradictory conditions (e.g. $A > 20 \wedge A < 18 \equiv false$), by possibly exploiting information about integrity constraints.
6. Elimination of operator NOT that precedes a simple condition, by rewriting it with its complementary. For example, $\neg(A > 20) \equiv A \leq 20$.
7. Generation of the internal representation of the SQL query as a logical plan, represented as an expression tree of relational algebra operators. Since SQL allows relations with duplicates, we will consider the *bag* (also called *multiset*) versions of the relational algebra operators. The user may request that duplicates be removed from the result of an SQL query by inserting the keyword **DISTINCT** after the keyword **SELECT**. For example, the query

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

   is represented with the logical plan in Figure 12.1, with the projection operator π^b, which does not eliminate duplicates.

12.3 Query Transformation Phase

The main purpose of this phase is to transform a query in order to increase the possibilities of optimizing its execution in the *Physical plan generation* phase.

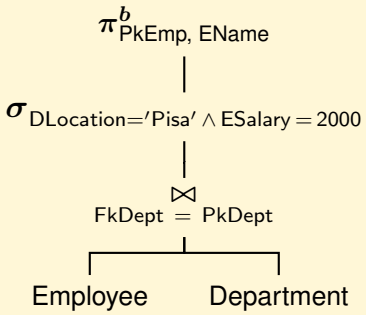


Figure 12.1: A logical query plan

A logical plan is transformed into an equivalent one using a set of equivalence rules, hereafter indicated, with the right hand side expression can be evaluated more efficiently than the left hand side expression.

Equivalence rules

Let E be a relational algebra expression, X, Y be sets of attributes, ψ_X be a condition on the attributes X, and XY be an abbreviation for $X \cup Y$. The projection operator under consideration is $\boldsymbol{\pi}^b$.

Example of equivalence rules are the following:

1. *Cascading of selections*

   $$\boldsymbol{\sigma}_{\psi_X}(\boldsymbol{\sigma}_{\psi_Y}(E)) = \boldsymbol{\sigma}_{\psi_X \wedge \psi_Y}(E)$$

2. *Commutativity of selection and projection*

   $$\boldsymbol{\pi}^b_Y(\boldsymbol{\sigma}_{\psi_X}(E)) = \boldsymbol{\sigma}_{\psi_X}(\boldsymbol{\pi}^b_Y(E))$$

   if $X \subseteq Y$, otherwise

   $$\boldsymbol{\pi}^b_Y(\boldsymbol{\sigma}_{\psi_X}(E)) = \boldsymbol{\pi}^b_Y(\boldsymbol{\sigma}_{\psi_X}(\boldsymbol{\pi}^b_{XY}(E)))$$

3. *Commutativity of selection and join*

   $$\boldsymbol{\sigma}_{\psi_X}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\psi_X}(E_1) \bowtie E_2$$

   if X are attributes of E_1.

   $$\boldsymbol{\sigma}_{\psi_X \wedge \psi_Y}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\psi_X}(E_1) \bowtie \boldsymbol{\sigma}_{\psi_Y}(E_2)$$

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

   $$\boldsymbol{\sigma}_{\psi_X \wedge \psi_Y \wedge \psi_Z}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\psi_Z}(\boldsymbol{\sigma}_{\psi_X}(E_1) \bowtie \boldsymbol{\sigma}_{\psi_Y}(E_2))$$

   if X are attributes of E_1, Y are attributes of E_2 and Z are attributes of both E_1 and E_2.

4. *Cascading of projections*

   $$\boldsymbol{\pi}^b_Z(\boldsymbol{\pi}^b_Y(E)) = \boldsymbol{\pi}^b_Z(E)$$

   if $Z \subseteq Y$.

5. *Elimination of unnecessary projections*

   $$\boldsymbol{\pi}^b_Z(E) = E$$

   if Z are the attributes of E.

6. *Commutativity of projection and join*

   $$\boldsymbol{\pi}^b_{XY}(E_1 \bowtie E_2) = \boldsymbol{\pi}^b_X(E_1) \bowtie \boldsymbol{\pi}^b_Y(E_2)$$

   where X are attributes of E_1, Y are attributes of E_2 and the join condition involves only attributes in XY.
   If the join condition involves attributes not in XY, then

   $$\boldsymbol{\pi}^b_{XY}(E_1 \bowtie E_2) = \boldsymbol{\pi}^b_{XY}(\boldsymbol{\pi}^b_{XX_{E_1}}(E_1) \bowtie \boldsymbol{\pi}^b_{YX_{E_2}}(E_2))$$

   where X_{E_1} are attributes of E_1 that are involved in the join condition, but are not in XY, and X_{E_2} are attributes of E_2 that are involved in the join condition, but are not in XY.

7. *Commutativity of selection and grouping*

   $$\boldsymbol{\sigma}_\theta({}_X\boldsymbol{\gamma}_F(E)) \equiv {}_X\boldsymbol{\gamma}_F(\boldsymbol{\sigma}_\theta(E))$$

   where θ uses only attributes from X and F is a set of aggregate functions that use only attributes from E. This equivalence is helpful because evaluation of the right hand side avoids performing the aggregation on groups which are anyway going to be removed from the result. Other interesting equivalences rules involving grouping are discussed in Section 12.4.4.

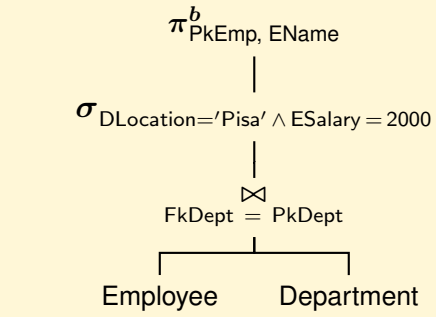
Possible steps of an algorithm to transform a query are the following:

1. Selections are pushed below projections using rule 2.
2. Selections are grouped using rule 1.
3. Selections and projections are pushed below an inner join using rule 3 and rule 6.
4. Repeat the three steps above until it is possible.
5. Unnecessary projections are eliminated using rule 5.
6. Projections are grouped using rule 4.

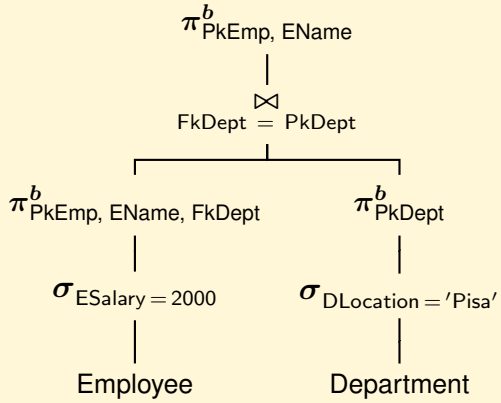
In general, the result of this algorithm is an expression in which the selection and the projection are executed as soon as possible, and the selection is ahead of projections. In particular, in the case of expressions with a single join, the expression rewritten has the form:

$$\boldsymbol{\pi}^b_{X_1}(\boldsymbol{\sigma}_{\psi_{Y_1}}(\boldsymbol{\pi}^b_{X_2}(\boldsymbol{\sigma}_{\psi_{Y_2}}(R)) \bowtie \boldsymbol{\pi}^b_{X_3}(\boldsymbol{\sigma}_{\psi_{Y_3}}(S))))$$

An example of transformation of the logical query plan in Figure 12.1 is shown in Figure 12.2.



(a) *Initial logical query plan*



(b) *Transformed logical query plan*

Figure 12.2: Query rewriting

DISTINCT Elimination

DISTINCT normally requires a physical plan with duplicate elimination, which often involves an expensive **Sort** operation. Therefore it is worthwhile to identify a useless **DISTINCT**, e.g. if a query result is without duplicates.

For simplicity, we assume that

- The database tables are without null values and are *sets* because have been defined with keys.
- The **FROM** clause uses a set of tables $\mathcal{R}$, where no attribute appears in two different tables.
- The condition C in the **WHERE** clause is a conjunctive normal form (CNF) of predicates involving attributes of the tables in $\mathcal{R}$.
- The **SELECT** clause uses the set of attributes $\mathcal{A}$.
- The **GROUP BY** clause uses a set of grouping attributes which also appear in the **SELECT** clause.

A SQL query with the **DISTINCT** clause is translated into a relational algebra expression with the projection operator $\pi^b_{\mathcal{A}}$ and the duplicate elimination operator δ.

To decide whether the operator δ for duplicate elimination is *unnecessary* we will use a basic algorithm of functional-dependency theory for determining if an *interest-*

ing functional dependency can be inferred from the set F of functional dependencies which hold in the query result.
Let us briefly recall some basic properties of functional dependencies.

Definition 12.1 *Functional Dependency*

Given a relation schema R and X, Y subsets of attributes of R, a functional dependency $X \rightarrow Y$ (X *determines* Y) is a constraint that specifies that for every possible instance r of R and for any two tuples $t_1, t_2 \in r$, $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$.

A peculiar example of functional dependency, that will be used in the following, is $\emptyset \rightarrow Y$: *it specifies that the value of* Y *is the same for every tuple of an instance* r *of* R.

Example 12.1
Let us consider the following relation containing information about students and exams at the University of Pisa.

StudentsExams

StudCode	Name	City	Region	BirthYear	Subject	Grade	University
1234567	N1	C1	R1	1995	DB	30	Pisa
1234567	N1	C1	R1	1995	SE	28	Pisa
1234568	N2	C2	R2	1994	DB	30	Pisa
1234568	N2	C2	R2	1994	SE	28	Pisa

Let us see if the following functional dependencies are properties of the meaning of the attributes in the relation:

- StudCode $\rightarrow$ Name, City, Region, BirthYear holds because each student code has the same name, city, region and birth year.
- City $\rightarrow$ Region holds because each city is in a region.
- StudCode, Subject $\rightarrow$ Grade holds because each students receive one grade in each subject.
- $\emptyset \rightarrow$ University holds because the data are only about students and exams at the University of Pisa.
- Subject $\rightarrow$ Grade does not hold because a subject may have different grades.
- Subject $\rightarrow$ Subject is not useful because always holds, and it is called "trivial".

Given a set F of functional dependencies, we can prove that certain other ones also hold. We say these ones are *logically implied* by F.

Definition 12.2 *Logical Implication*

Given a set F of functional dependencies on a relation schema R, another functional dependency $X \rightarrow Y$ is *logically implied* by F if every instance of R that satisfies F also satisfies $X \rightarrow Y$:

$$F \vdash X \rightarrow Y$$

This property holds if $X \rightarrow Y$ can be derived by F using the following set of inference rule, known as the *Armstrong's axioms*:

$$\begin{aligned} (\textit{Reflexivity}) \quad & \text{If } Y \subseteq X, \text{ then } X \rightarrow Y \\ (\textit{Augmentation}) \quad & \text{If } X \rightarrow Y, Z \subseteq T, \text{ then } XZ \rightarrow YZ \\ (\textit{Transitivity}) \quad & \text{If } X \rightarrow Y, Y \rightarrow Z, \text{ then } X \rightarrow Z \end{aligned}$$

A simpler way of solving the implication problem follows from the following notion of *closure* of an attribute set.

Definition 12.3 *Closure of an Attribute Set*

Given a schema $R < T, F >$, and $X \subseteq T$, the *closure* of X, denoted by X^+, is

$$X^+ = \{A_i \in T | F \vdash X \rightarrow A_i\}$$

The closure of attributes is used in the following result.

Theorem 12.1

$F \vdash X \rightarrow Y$ iff $Y \subseteq X^+$

Thus testing whether a functional dependency $X \rightarrow Y$ is implied by F can be accomplished by computing the closure X^+. The following simple algorithm can be used to compute this set, although a more efficient and complex one exists.[1]

Algorithm 12.1 *Computing the Closure of an Attribute Set X*

$X^+ = X$;
while (changes to X^+) **do**
  for each $W \rightarrow V$ **in** F **with** $W \subseteq X^+$ **and** $V \not\subseteq X^+$
    do $X^+ = X^+ \cup V$;

Functional dependencies are commonly used to normalize a database schema, but in the following we will show that they are also useful to reason about the properties of a query result.

A set of functional dependencies F which hold in the result of a query with joins and selections is found as follows:

1. Let F be the initial set of functional dependencies where their determinants are the keys of every table used in the query.
2. Let C the condition of the $\boldsymbol{\sigma}$ operator. If a conjunct of C is a predicate $A_i = c$, where c is a constant, F is extended with the functional dependency $\emptyset \rightarrow A_i$.
3. If a conjunct of C is a predicate $A_j = A_k$, e.g. a join condition, F is extended with the functional dependencies $A_j \rightarrow A_k$ and $A_k \rightarrow A_j$.

1. For an example see the application implemented by R. Orsini available at this URL:http://dblab.dsi.unive.it:8080

Note that the following properties hold.

– If $X \rightarrow Y$ in R or in S, then this is still the case in $R \times S$.
– If X is a key for R and Y is a key for S then $X \cup Y$ is a key for $R \times S$.

A way for determining if an *interesting* functional dependency $X \rightarrow Y$ may be inferred from the set F is to use the previous algorithm to compute X^+. Instead, in the following examples we will use the following version of the algorithm to compute X^+, which does not use explicitly the functional dependencies.

Algorithm 12.2 *Computing the Closure of an Attribute Set X*

1. Let $X^+ = X$.
2. Add to X^+ all attributes A_i such that the predicate $A_i = c$ is a conjunct of $\boldsymbol{\sigma}$, where c is a constant.
3. Repeat until X^+ is changed:
   (a) Add to X^+ all attributes A_j such that the predicate $A_j = A_k$ is a conjunct of $\boldsymbol{\sigma}$ and $A_k \in X^+$.
   (b) Add to X^+ all attributes of a table if X^+ contains a key for that table.

The following theorem gives a sufficient condition for duplicate elimination.

Theorem 12.2 *Duplicate Elimination*

Let $\mathcal{A}$ be the set of attributes of the result and $\mathcal{K}$ be the union of the attributes of the key of *every* table used in the query. If $\mathcal{A} \rightarrow \mathcal{K}$, e.g. if $\mathcal{A}^+$ contains a key of *every* table used in the query, then δ is *unnecessary*.

Although the use of the closure $\mathcal{A}^+$ is not enough to discover all cases where duplicate elimination is unnecessary, it handles a large subclass of queries [Ceri and Widom, 1991], [Paulley and Larson, 1994].

Example 12.2
Let us consider the database

Products(PkProduct, ProductName, UnitPrice)
Invoices(PkInvoiceNo, Customer, Date, TotalPrice)
InvoiceLines(FkInvoiceNo, LineNo, FkProduct, Qty, Price)

and the query

```
SELECT  DISTINCT FkInvoiceNo,TotalPrice
FROM    InvoiceLines, Invoices
WHERE   FkInvoiceNo = PkInvoiceNo;
```

To check whether **DISTINCT** is unnecessary let us compute the closure of the projection attributes.

$$\begin{aligned} \mathcal{A}^+ &= \{\text{FkInvoiceNo,TotalPrice}\} \\ &= \{\text{FkInvoiceNo,TotalPrice, PkInvoiceNo, Customer, Date}\} \end{aligned}$$

DISTINCT is necessary because $\mathcal{A}^+$ does not contain the key of the table InvoiceLines.

Let us change now the previous query as follows and compute $\mathcal{A}^+$:

```
SELECT DISTINCT FkInvoiceNo,TotalPrice
FROM   InvoiceLines, Invoices
WHERE  FkInvoiceNo = PkInvoiceNo AND LineNo = 1;
```

$$\begin{aligned}\mathcal{A}^+ &= \{\text{FkInvoiceNo,TotalPrice}\}\\ &= \{\text{FkInvoiceNo,TotalPrice, LineNo}\}\\ &= \{\text{FkInvoiceNo,TotalPrice, LineNo, PkInvoiceNo, Customer, Date,}\\ &\quad\ \text{FkProduct, Qty, Price}\}\end{aligned}$$

DISTINCT becomes unnecessary because $\mathcal{A}^+$ contains the primary keys of all tables.

*When a query contains **GROUP BY**, let $\mathcal{G}$ be the grouping attributes. The **DISTINCT** is unnecessary if $\mathcal{A} \rightarrow \mathcal{G}$, e.g. if $\mathcal{A}^+$ contains the grouping attributes.*

Let us consider the following query and the closure of the projection attributes:

```
SELECT   DISTINCT FkInvoiceNo, COUNT(*) AS N
FROM     InvoiceLines, Invoices
WHERE    FkInvoiceNo = PkInvoiceNo
GROUP BY FkInvoiceNo, Customer;
```

$$\begin{aligned}\mathcal{A}^+ &= \{\text{FkInvoiceNo}\}\\ &= \{\text{FkInvoiceNo, PkInvoiceNo}\}\\ &= \{\text{FkInvoiceNo, PkInvoiceNo, Customer, Date, TotalPrice}\}\end{aligned}$$

The **DISTINCT** is unnecessary because $\mathcal{A}^+$ contains the grouping attributes.

GROUP BY Elimination

A **GROUP BY**, as a **DISTINCT**, frequently requires a physical plan with an expensive **Sort** operation. Therefore it is worthwhile to test if a **GROUP BY** can be eliminated, either because *each group consists only of a single record* or because *there is only a single group*.

The test can be done as shown with the following examples.

Example 12.3

Let us consider the database

Products(PkProduct, ProductName, UnitPrice)
Invoices(PkInvoiceNo, Customer, Date, TotalPrice)
InvoiceLines(FkInvoiceNo, LineNo, FkProduct, Qty, Price)

and the query

```
SELECT   FkInvoiceNo, COUNT(*) AS N
FROM     InvoiceLines, Invoices
WHERE    FkInvoiceNo = PkInvoiceNo
         AND TotalPrice > 10000 AND LineNo = 1
GROUP BY FkInvoiceNo, Customer;
```

*The **GROUP BY** is unnecessary if each group consists only of a single record.*

To check whether the **GROUP BY** is unnecessary, let us check whether the following query producing the data to be grouped is without duplicates.

```
SELECT   FkInvoiceNo, Customer
FROM     InvoiceLines, Invoices
WHERE    FkInvoiceNo = PkInvoiceNo
         AND TotalPrice > 10000 AND LineNo = 1;
```

$$\begin{aligned}\mathcal{A}^+ &= \{\text{FkInvoiceNo, Customer}\}\\ &= \{\text{FkInvoiceNo, Customer, LineNo}\}\\ &= \{\text{FkInvoiceNo, Customer, LineNo, PkInvoiceNo, } \ldots\}\end{aligned}$$

$\mathcal{A}^+$ contains the primary keys of all tables, and therefore the query result is without duplicates and the **GROUP BY** query can be rewritten as a projection with the aggregation functions rewritten with the rules: COUNT as 1; MIN(A), MAX(A), SUM(A), AVG(A) as A.

```
SELECT   FkInvoiceNo, 1 AS N
FROM     InvoiceLines, Invoices
WHERE    FkInvoiceNo = PkInvoiceNo
         AND TotalPrice > 10000 AND LineNo = 1;
```

*The **GROUP BY** is unnecessary if there is only a single group.*

To check whether the **GROUP BY** is unnecessary let us check whether the value of the *grouping attributes is identical for every tuple*, e.g. whether the closure of the empty set of attributes $\{\}^+$ contains the grouping attributes.

```
Q1: SELECT   PkProduct, COUNT(*) AS N
    FROM     InvoiceLines, Products
    WHERE    FkProduct = PkProduct AND FkProduct = 44
    GROUP BY PkProduct;
```

$$\begin{aligned}\{\}^+ &= \{\text{FkProduct}\}\\ &= \{\text{FkProduct, PkProduct, ProductName, UnitPrice}\}\end{aligned}$$

The **GROUP BY** is unnecessary since $\{\}^+$ contains the grouping attributes, and the query can be rewritten as follows:

```
SELECT   44 AS PkProduct, COUNT(*) AS N
FROM     InvoiceLines, Products
WHERE    FkProduct = PkProduct AND FkProduct = 44;
```

Note the following points:

– The rewriting is only possible if the values of the constant attributes in the **SELECT** are known in the query. For instance, the query:

Q2: **SELECT** PkProduct, ProductName, COUNT(*) **AS** N
FROM InvoiceLines, Products
WHERE FkProduct = PkProduct **AND** FkProduct = 44
GROUP BY PkProduct, ProductName;

cannot be rewritten although the grouping attributes are contained in $\{\}^+$, since ProductName is a constant, but its value is not known in the query and so it cannot be rewritten in the **SELECT**. Therefore the **GROUP BY** cannot be eliminated but, since there is only a single group, *in the physical query plan the operator does not require the **Sort** operator*.

– Let us change the query Q1 by eliminating the aggregation in the **SELECT**:

SELECT PkProduct
FROM InvoiceLines, Products
WHERE FkProduct = PkProduct **AND** FkProduct = 44
GROUP BY PkProduct;

The **GROUP BY** is again unnecessary because produces only a single group, and the query is rewritten by using **DISTINCT** as follows, for the equivalence rule $\delta(\pi^b_{A_1,A_2,...,A_n}(E)) \equiv {}_{A_1,A_2,...,A_n}\gamma(E)$:

SELECT **DISTINCT** PkProduct
FROM InvoiceLines, Products
WHERE FkProduct = PkProduct **AND** FkProduct = 44;

The same rewriting with **DISTINCT** is now possible also of the query Q2 without the aggregation in the **SELECT**:

SELECT **DISTINCT** PkProduct, ProductName
FROM InvoiceLines, Products
WHERE FkProduct = PkProduct **AND** FkProduct = 44;

In both cases, since there is only a single group, *in the physical query plan the **DISTINCT** does not require the **Sort** operator*.

WHERE-Subquery Elimination

Subqueries can occur in the **WHERE** clause through the operators $=, <, >, <=, >=, <>$; through the quantifiers **ANY**, or **ALL**; or through the operators **EXISTS** and **IN** and their negations **NOT EXISTS** and **NOT IN**. We assume that all of these case have been converted into an equivalent form using only **EXISTS** and **NOT EXISTS**.

In general, to execute such queries, the optimizer generate a physical plan for the subquery, which then will be executed for each record processed by the outer query. Therefore, the presence of a subquery makes the physical plan more expensive, and for this reason techniques have been studied to transform a query into an equivalent one without the subquery, which the optimizer processes more efficiently.

Kim has proposed some basic *subquery types* and for each of them has given the transformation rule to rewrite the query [Kim, 1982], [Ganski and Wong, 1987]. The general case is a complex problem. Let us restrict the subquery elimination to the case of **EXISTS** *without* **NOT** and correlated subquery without aggregations, discussed then by examples:

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

is equivalent to the join query

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

DISTINCT is necessary in the join form when a (1:N) relationship exists between R1 and R2.

Example 12.4
Let us consider the database

Courses(CrsName, CrsYear,Teacher, Credits)
Transcripts(StudId, CrsName, Year, Date, Grade)

and the query with **EXISTS** to list all courses with at least a transcript, for the year 2012.

```
SELECT    *
FROM      Courses C
WHERE     CrsYear = 2012 AND
          EXISTS
          (
          SELECT    *
          FROM      Transcripts T
          WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          );
```

The unnested equivalent query is

```
SELECT    DISTINCT C.*
FROM      Courses C, Transcripts T
WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          AND CrsYear = 2012;
```

Let us consider now a query with **EXISTS** and a correlated subquery with an aggregation, to list all courses with an average grade greater than 27, for the year 2012.

```
SELECT    CrsName, Teacher
FROM      Courses C
WHERE     CrsYear = 2012 AND
          EXISTS
          (
          SELECT    AVG(Grade)
          FROM      Transcripts T
          WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          HAVING    27 < AVG(Grade)
          );
```

The unnested equivalent query is

```
SELECT    C.CrsName AS CrsName, Teacher
FROM      Courses C, Transcripts T
WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          AND CrsYear = 2012
GROUP BY  C.CrsName, CrsYear, Teacher
HAVING    27 < AVG(Grade);
```

Let us assume that there are courses without transcripts, and we want to list all of them for the year 2012. The following query with **EXISTS** is correct because it does not eliminate a course without a transcript from the result:

```
SELECT    CrsName, Teacher
FROM      Courses C
WHERE     CrsYear = 2012 AND
          EXISTS
          (
          SELECT    COUNT(Grade)
          FROM      Transcripts T
          WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          HAVING    0 = COUNT(Grade)
          );
```

Let us unnest the query as in the previous case:

```
SELECT    C.CrsName AS CrsName, Teacher
FROM      Courses C, Transcripts T
WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          AND CrsYear = 2012
GROUP BY  C.CrsName, CrsYear, Teacher
HAVING    0 = COUNT(Grade);
```

Unfortunately the unnested query is not equivalent to the query with **EXISTS** because the join does not include in the result a course without a transcript.

This wrong example of **WHERE**-subquery elimination is well-known as the *count bug problem*, and only arises when the aggregation function is COUNT. A correct solution in this case consists in replacing the unnested query join by an *outer join* as follows:

```
SELECT    C.CrsName AS CrsName, Teacher
FROM      Courses C OUTER JOIN Transcripts T
                    ON (T.CrsName = C.CrsName AND T.Year = CrsYear)
          AND CrsYear = 2012
GROUP BY  C.CrsName, CrsYear, Teacher
HAVING    0 = COUNT(Grade);
```

The use of the *outer join* in the **WHERE**-subquery elimination is also necessary for queries with **NOT EXISTS**. For example, to list all courses without a transcript for the year 2012, the following query

```
SELECT    *
FROM      Courses C
WHERE     CrsYear = 2012 AND
          NOT EXISTS
          (
          SELECT    *
          FROM      Transcripts T
          WHERE     T.CrsName = C.CrsName AND T.Year = CrsYear
          );
```

must be unnested as follows

```
SELECT   C.*
FROM     Courses C OUTER JOIN Transcripts T
                   ON (T.CrsName = C.CrsName AND T.Year = CrsYear)
WHERE    CrsYear = 2012 AND T.CrsName IS NULL;
```

View Merging

Complex queries are much easier to write and understand if views are used. A view can be created with a **CREATE VIEW** VName clause, and the definition stays in the database until a command **DROP VIEW** VName is executed. Instead, the use of the **WITH** clause provides a way of defining temporary views available only to the query in which the clause occurs.

In general, when a query uses a view, the optimizer generates a physical sub-plan for the **SELECT** that defines the view, and optimizes the query considering the scan as the only access method available for the result of the view. This technique usually leads to a suboptimal physical query plan, because the view is optimized separately from the rest of the query. Better physical plans can be generated when in the query transformation phase the **SELECT** which defines the view can be absorbed into the query definition, and so a view sub-plan is no longer necessary.

Example 12.5
Let us consider the database Personnel

Department(PkDept, DName, Location)
Employee(PkEmp, EName, Job, Salary, FkDept)

and the query

```
WITH     Technician AS
         (
            SELECT  PkEmp, EName, Salary
            FROM    Employee
            WHERE   Job = 'Technician'
         )
SELECT   EName, Salary
FROM     Technician
WHERE    Salary > 2000;
```

Using view merging, the query can be transformed into:

```
SELECT   EName, Salary
FROM     Employee
WHERE    Salary > 2000 AND Job = 'Technician';
```

The transformation is made by replacing, in the logical query plan, the reference to a view name with its logical plan. Then the new logical plan is rewritten using the equivalence rules of relational algebra to put it in the following *canonical form* of an SQL query without the use of views, with grouping and selection above all the joins:

$$\pi^b(\sigma(\gamma(\sigma(\bowtie_J R_i))))$$

The transformation of a query to avoid the use of views is not always possible, especially if a view is defined with a **GROUP BY**.

Example 12.6
Let us consider the database Personnel and the view

```
CREATE  VIEW  NoEmployeeByDept AS
        SELECT    FkDept, COUNT(*) AS NoEmp
        FROM      Employee
        GROUP BY  FkDept;
```

To find the average number of employees of the departments we use the query

```
SELECT  AVG(NoEmp)
FROM    NoEmployeeByDept;
```

which cannot be transformed to avoid using the view.

The transformation of a query to avoid the use of views defined with a **GROUP BY** generally requires pulling the **GROUP BY** up above a join present in the query using the following algebraic equivalence rule:

Definition 12.4 *Pulling-up-grouping-over-join*

Let X_R be a set of attributes of R, with $f_k \in X_R$ the foreign key of R, p_k the primary key of S, with attributes $\mathcal{A}(S)$, then

$$({}_{X_R}\gamma_{\,F}(R)) \bowtie_{f_k=p_k} S \equiv {}_{X_R \cup \mathcal{A}(S)}\gamma_{\,F}(R \bowtie_{f_k=p_k} S)$$

Example 12.7
Let us consider the database Personnel

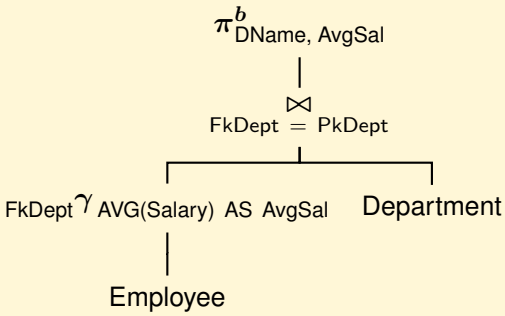
Department(PkDept, DName, Location)
Employee(PkEmp, EName, Job, Salary, FkDept)

and the query

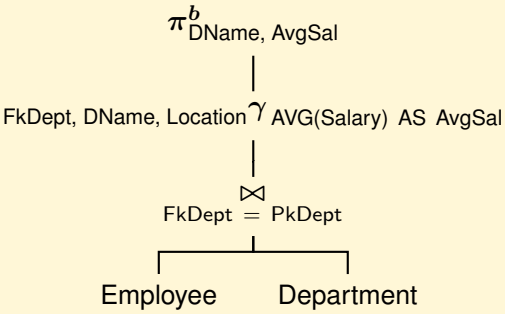
```
WITH      EmpGby AS
          (
             SELECT    FkDept, AVG(Salary) AS AvgSal
             FROM      Employee
             GROUP BY  FkDept
          )
SELECT    DName, AvgSal
FROM      EmpGby, Department
WHERE     FkDept = PkDept;
```

The manual of a DBMS says: "Because there is no equivalent statement that accesses only base tables, the optimizer cannot transform this statement. Instead, the optimizer chooses an execution plan that issues the view's query and then uses the resulting set of rows as it would the rows resulting from a table access."

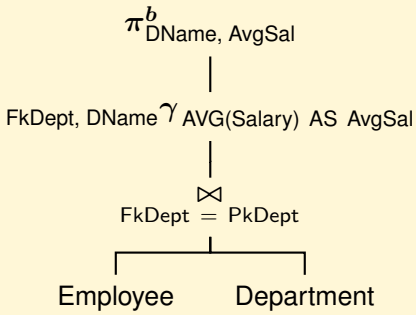
We show, instead, how the query can be rewritten without the use of the view. Consider the logical query plan, where the view EmpGby has been replaced with its logical plan:



For the *pulling-up-grouping-over-join* rule, the tree can be rewritten as:



Location can be eliminated from the grouping attributes because FkDept DName $\rightarrow$ Location and it is not used in the projection attributes, and so the logical query plan becomes



The rewritten logical query plan represents the query

```
SELECT    DName, AVG(Salary) AS AvgSal
FROM      Employee, Department
WHERE     FkDept = PkDept
GROUP BY  FkDept, DName;
```

12.4 Physical Plan Generation Phase

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

The main steps of the *Physical plan generation* phase are:

- Generation of alternative physical query plans.
- Choice of the physical query plan with the lowest estimated cost.

To estimate the cost of a physical query plan it is necessary to estimate, for each node in the physical tree, the following parameters by means of the techniques seen in the previous chapter:

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

Before seeing how to generate alternative plans, and how to choose the least expensive one, let us see an example of alternative physical plans for a query and their cost.

Example 12.8
Let us consider the database

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

and the query:

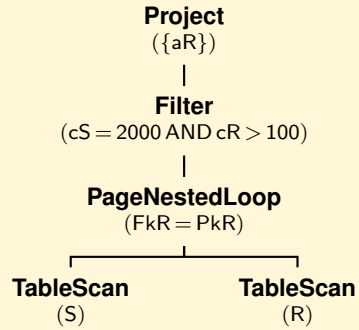
```
SELECT  aR
FROM    S, R
WHERE   FkR = PkR  AND  cS = 2000  AND  cR > 100;
```

Suppose that:

- The attribute values have a uniform distribution in the relations;
- $N_{\mathsf{pag}}(\mathsf{S}) = 1000$, $N_{\mathsf{rec}}(\mathsf{S}) = 100\,000$ and $N_{\mathsf{key}}(\mathsf{cS}) = 100$;
- $N_{\mathsf{pag}}(\mathsf{R}) = 500$, $N_{\mathsf{rec}}(\mathsf{R}) = 40\,000$ and the simple condition on cR has a selectivity factor of 1/3;
- $N_{\mathsf{pag}}(\mathsf{T}) = 100$, $N_{\mathsf{rec}}(\mathsf{R}) = 20\,000$;
- There is a clustered index on cS, and unclustered indexes on primary and foreign keys, and on (cS, FkR, bS);
- In the cost estimation with the use of an index, the component C_I will not be considered.

Let us consider two alternative physical query plans, with S as external operand:

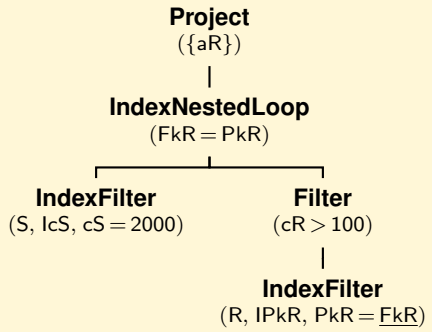
1. The query is executed with the following physical plan using the join operator **PageNestedLoop**.



The query plan cost is

$$\begin{aligned} C_{\mathsf{PNL}} &= N_{\mathsf{pag}}(\mathsf{S}) + N_{\mathsf{pag}}(\mathsf{S}) \times N_{\mathsf{pag}}(\mathsf{R}) \\ &= 1000 + 1000 \times 500 = 501\,000 \end{aligned}$$

2. The query is executed with the following physical query plan using the available indexes.



The query plan cost is

$$\begin{aligned} C_{\mathsf{INL}} &= C(O_E) + E_{\mathsf{rec}}(O_E) \times C(O_I) \\ &= 10 + 1000 \times 1 = 1010 \end{aligned}$$

The records of S that satisfy the condition are retrieved using the clustered index on cS with the cost of $\lceil 1/100 \times 1000 \rceil = 10$ page accesses. For each record retrieved (in total 1000), the relation R is accessed using the index on the primary key PkR with the cost of 1010 page accesses, which is significantly lower than the one of the previous case.

12.4.1 Working Hypothesis

We will study the query optimization problem under certain assumptions about the *query types*, in addition to those already made in the previous chapter about the *physical data organization*, the *cost model*, the *buffer management* strategies, and the *statistics* available.

Query Types

The main problems to solve in optimizing queries will be shown using a SQL language subset, and then we will see how to extend the results to deal with the general case. In particular, the focus will initially be on queries such as:

SELECT AttributesList
FROM RelationsList
WHERE Condition;

where:

- AttributesList are the attributes of the query result ($*$ stands for all the attributes).
- RelationsList are the relations used.
- Condition can be a conjunction of simple conditions on relations attributes A_i. We assume that simple conditions are of the following types:
  - $A_i\ \theta\ c_i$, with $\theta \in \{>, \geq, <, \leq, =, \neq\}$ and c_i a constant in $dom(A_i)$.
  - A_i IN $(c_1; \ldots; c_n)$.
  - A_i BETWEEN c_1 AND c_2.
  - $A_i = A_j$ (*equi-join condition*).

  where the same attributes A_i of different relations, R and S, are made different writing them as $R.A_i$ and $S.A_i$;

When a condition involves attributes of two relations R and S, we assume that it is in the form: ψ_R AND ψ_S AND ψ_{Join}, where ψ_R is a condition on the R attributes, ψ_S is a condition on the S attributes, and ψ_{Join} is a join condition.

Note, therefore, that initially the focus will be on queries in which the following clauses are not used: **DISTINCT**, **GROUPBY**, subqueries, set operators, and views. Once the optimization of this type of query has been considered, we will see how to release the restrictions for considering the general case. The subset of the SQL language taken initially into consideration simplifies the problem, but it is sufficient to focus on the fundamental problems to be solved during query optimization.

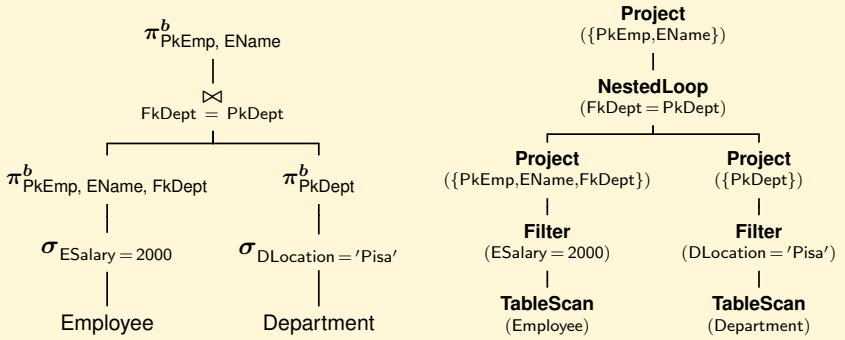
Physical Query Plan

A physical query plan is the algorithm selected by the optimizer to execute a query. For example, the query in Figure 12.3a is executed with the physical plan in Figure 12.3b.

In the following, some of the physical operators provided by the JRS will be used, described in the previous chapter:[2]

Relation (R)	**TableScan**(R), **IndexScan**(R, I), **SortScan**$(R, \{A_i\})$.
Selection (σ_ψ)	**Filter**(O, ψ), **IndexFilter**(R, I, ψ), **IndexOnlyFilter**(R, I, ψ).
Projection (π^b)	**Project**$(O, \{A_i\})$.
Duplicate elimination (δ)	**Distinct**(O).
Join $\overset{(\bowtie}{\psi_J)}$	**NestedLoop**(O_E, O_I, ψ_J), **IndexNestedLoop**(O_E, O_I, ψ_J), **MergeJoin**(O_E, O_I, ψ_J).
Sort $(\tau_{\{A_i\}})$	**Sort**$(O, \{A_i\})$.

2. The operands of a physical operator are relations R or other operators O. O_E stands for the external operand and O_I for the internal operand.



(a) *Logical query plan* **(b)** *Physical query plan*

Figure 12.3: An example of logical and physical query plans

The following sections will show how to choose the physical query plan of minimum cost considering first the case of queries on a single relation, with only a selection condition, and then the case of queries on several relations, with selection and join conditions. We will also make some considerations for the case of queries with **ORDER BY**.

12.4.2 Single-Relation Queries

If the query uses just one relation, the operations involved are the projection and selection. For example:

```
SELECT  bS
FROM    S
WHERE   FkR > 100 AND cS = 2000;
```

If there are no useful indexes, the solution is trivial: a relation scan is performed to find the records that match the condition, and then the projection is applied. If there are useful indexes, different solutions are possible.

Use of Single-Index. If there are different indexes usable for one of the simple conditions, the one of minimal cost is used, and then the operation is completed by testing whether the retrieved records satisfy the remaining conditions and the projection is applied.
The records of the result are sorted on the index attribute. If sorting the result on a different attribute is required, then in the choice of the index the cost of a sort should be taken into account too.

For the query under consideration, the best physical plan is with the index on cS. Then the records that match the condition on FkR are projected over bS.

Use of Multiple-Indexes. We proceed as in the previous chapter in the case of the selection operator. The operation is completed by testing whether the retrieved records satisfy the remaining conditions and the projection is applied.

For the query under consideration, both the indexes on cS and FkR are used to find the RIDs of the records that match both conditions. Then the lists are intersected and the records are retrieved from the relation and projected over bS.

Index-Only Plans. All the main DBMS always try to create index-only query plans, because they have better performance. Therefore, as we have seen, they support indexes that contain both search attributes and some extra attributes, chosen among those that are often accessed together with the search attributes. Such indexes are useful to support index-only query plans.

Use of Index-Only. If all the attributes of the condition of the **SELECT** are included in the prefix of the key of an index on the relation, the query can be evaluated using only the index with the query plan of minimum cost.

For the query under consideration, the index on (cS, FkR, bS) is used both to find the records matching the condition, and the value of the attribute bS in the **SELECT**, without accesses to the relation. Estimating the costs of the physical query plan in the three cases considered is left as an exercise.

12.4.3 Multiple-Relation Queries

Queries with two or more relations in the **FROM** clause require joins (or cross-products). Finding a good physical query plan for these queries is very important to improve their performances.

Several algorithms have been proposed for a *cost-based query optimization* with approaches based on heuristic search or dynamic programming techniques [Steinbrunn et al., 1997; Lanzelotte and Valduriez, 1991; Kossmann and Stocker, 2000]. In the following, for simplicity, we will consider only a version of the first approach, based on the idea of generating and searching a *state space* of possible solutions to find the one with minimal cost.

The state space is constructed step-by-step starting with an initial state and repeatedly applying a set of operators to *expand* a state s into other ones, called the *successors* of s.

Each state s corresponds to a relational algebra subexpression of a given query Q to optimize, and the successors of s are larger subexpressions of Q. The cost of a state s is that of the best physical plan to execute the expression associated to s.

The state space is represented as a tree of nodes, where the root is the empty subexpression, the first level nodes are the query relations or selections and projections on each of them; the following levels are alternative joins of the algebraic expressions of the previous level. The node of the "optimum" state is the expression the contains all the query joins, which is then extended with other operators (e.g. *project*, *sort* or *group by*) to become the final state of the query expression, with the associated minimal cost physical query plan.

Figure 12.4 gives a description of a *full search* algorithm to find a solution in the states space.

As queries become more complex, the full search algorithm cannot find the overall best physical query plan in a reasonable amount of time because of the exponential nature of the problem. Therefore, several heuristics have been proposed in the DBMS literature that allow the query optimizer to avoid bad query plans and find query plans that, while not necessarily optimal, are usually "good enough". Let us see some examples of the most commonly used heuristics, in particular those for reducing both the number of states generated and the number of states to consider for expansions.

– ***Limitation of the number of successors***. As we saw in the previous chapter, the cost of a join depends on the order in which the operands are considered. In the

Input: A query $Q = \tau(\pi(\boldsymbol{\sigma}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)))$
Output: The best query plan for Q

1: **Step 1**. Let *generated states* be the set that contains one query subexpressions for each R_i, with local conditions applied, and the associated best physical plan.
2: **Step 2**. Expand next state.
3: **for** $i = 2$ **to** n **do** {
4: Let S be the state in *generated states* with minimum cost;
5: If S contains all the query joins, **exit from for**;
6: Eliminate S from *generated states*;
7: Expand S with all possible joins of i relations, and the associated best physical plan, and add them to *generated states*
8: }
9: **Step 3**. Let *best plan* be the query plan associated to S completed with other operators in Q.
10: **return** *best plan*.

Figure 12.4: A *full search* algorithm.

case of a join between more than two relations the number of alternatives increases considerably. For example, to perform the join of three relations $R(A, B)$, $S(B, C)$ and $T(C, D)$, it is necessary to consider all the permutations and for each of them, for example R, S, T, the possible evaluation orders of the joins: $(R \bowtie S) \bowtie T$ e $R \bowtie (S \bowtie T)$. In general, with n relations there are $(2(n-1))!/(n-1)!$ ways to perform the joins. For $n = 5$, the number of ways is 1680, which becomes 17 297 280 for $n = 8$.

To reduce the number of cases to consider, the following heuristic is usually adopted, which has been proposed and experimented with the *System R* optimizer: each permutation is evaluated by associating to the left the join operators, thus producing *left-deep* trees, where the right operand of a join node is not another join, as it happens in *right-deep* or *bushy* trees (Figure 12.5a, instead of (Figure 12.5b). A *left-deep* tree has the advantage of allowing the use of an index nested loop join operator.

For example, to execute the join $R \bowtie S \bowtie T$, 4 cases only are considered instead of 12: $(R \bowtie S) \bowtie T$, $(S \bowtie R) \bowtie T$, $(S \bowtie T) \bowtie R$, $(T \bowtie S) \bowtie R$.

– ***Greedy search.*** Once the logical node of minimum cost has been selected to be expanded, the other nodes will not be considered any longer by the search algorithm, and will be eliminated from *generated states*.

  In general, the final physical query plan found with a greedy search is a suboptimal solution but, thanks to the fact that it is found in less time, this search is usually used by default by DBMSs.

– ***Iterative full search.*** In the case of complex queries, for example with more than 15 joins, another possibility for searching a good physical query plan is to use a mixed approach: the full search is made up to a predetermined number of levels, for example 4, then the best node to expand is chosen, and the other nodes will not be considered any longer, as with a greedy search. Then the full search is made again for the predetermined number of levels, and so on [Steinbrunn et al., 1997; Kossmann and Stocker, 2000].

– ***Interesting orders.*** When the *merge-join* operator is available, or the query result must be sorted as required by the presence of SQL **ORDER BY** or **GROUP BY** clauses, it is useful to organize the search as was proposed for the *System R*: At

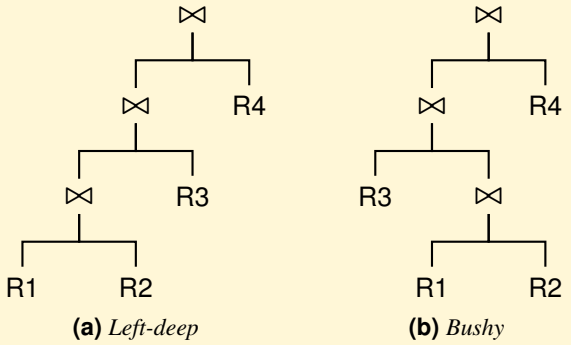


Figure 12.5: Examples of equivalent join trees

each step, for each logical query subexpression, in addition to the query plan with minimum cost, also the best plans producing an *interesting order* of the records potentially useful for the final physical query plan are preserved.

Example 12.9
Consider the relations:

P(aP :integer, FkQ :integer, bP :string, cP :string, dP :string)
Q(PkQ :integer, aQ :string, bQ :integer, cQ :string)

and the query:

```
SELECT  aQ, bP
FROM    Q, P
WHERE   PkQ = FkQ  AND  bQ = 13  AND  cP = 'AA';
```

Let us assume that

- there are four unclustered indexes on bQ, PkQ, cP and FkQ;
- the statistics shown in Table 12.1 are available;
- a join is executed with the methods *nested loop* or *index nested loop*, to simplify in this example the number of cases to be considered in the search for the best physical query plan.

Q	P
$N_{\text{rec}} = 200$	$N_{\text{rec}} = 2\,000$
$N_{\text{pag}} = 100$	$N_{\text{pag}} = 400$
$N_{\text{key}}(\text{IbQ}) = 20$	$N_{\text{key}}(\text{IcP}) = 100$
$N_{\text{leaf}}(\text{IbQ}) = 3$	$N_{\text{leaf}}(\text{IcP}) = 20$
$N_{\text{key}}(\text{IPkQ}) = 200$	$N_{\text{key}}(\text{IFkQ}) = 200$
$N_{\text{leaf}}(\text{IPkQ}) = 5$	$N_{\text{leaf}}(\text{IFkQ}) = 25$

Table 12.1: Statistics on relations Q and P

The query is optimized with a *greedy search* algorithm organized into the following steps.

Step 1
For each relation R used by the query, the relational algebra subexpression E_R with the selection pushed-down to the relation is considered. Alternative physical plans for each expression are generated, and the cheapest plan is chosen (Figure 12.6).

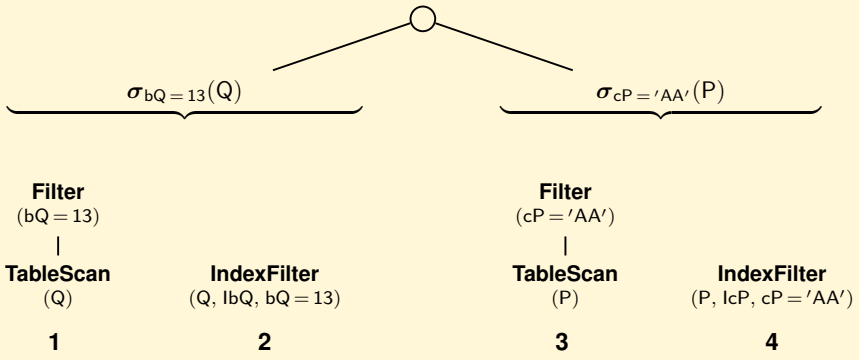


Figure 12.6: Search tree with alternative physical plans for subexpression on relations

The costs of the physical plans for $E_Q = \sigma_{bQ=13}(Q)$ are:

$$\begin{aligned} C^1(E_Q) &= N_{\mathsf{pag}}(\mathsf{Q}) = 100 \\ C^2(E_Q) &= \lceil f_s(\mathsf{bQ} = 13) \times N_{\mathsf{leaf}}(\mathsf{IbQ}) \rceil + \\ &\qquad \Phi(\lceil N_{\mathsf{rec}}(\mathsf{Q})/N_{\mathsf{key}}(\mathsf{IbQ}) \rceil, N_{\mathsf{pag}}(\mathsf{Q})) \\ &= \lceil 3/20 \rceil + \Phi(10, 100) = 1 + 10 = 11 \end{aligned}$$

In both cases the result size is

$$E_{\mathsf{rec}} = \lceil N_{\mathsf{rec}}(\mathsf{Q})/N_{\mathsf{key}}(\mathsf{IbQ}) \rceil = 200/20 = 10$$

The best physical plan for E_Q is 2, with the index on bQ.

The costs of the physical plans for $E_P = \sigma_{cP='AA'}(P)$ are:

$$\begin{aligned} C^3(E_P) &= N_{\mathsf{pag}}(\mathsf{P}) = 400 \\ C^4(E_P) &= \lceil f_s(\mathsf{cP = 'AA'}) \times N_{\mathsf{leaf}}(\mathsf{IcP}) \rceil + \\ &\qquad \Phi(\lceil N_{\mathsf{rec}}(\mathsf{P})/N_{\mathsf{key}}(\mathsf{IcP}) \rceil, N_{\mathsf{pag}}(\mathsf{P})) \\ &= \lceil 20/100 \rceil + \Phi(20, 400) = 1 + 20 = 21 \end{aligned}$$

In both cases the result size is

$$E_{\mathsf{rec}} = \lceil N_{\mathsf{rec}}(\mathsf{P})/N_{\mathsf{key}}(\mathsf{IcP}) \rceil = 2\,000/100 = 20$$

The best physical plan is 4 for E_P, with the index on cP, and cost 21 (Table 12.2).

	1	2	3	4
Plan	Fig. 12.6	Fig. 12.6	Fig. 12.6	Fig. 12.6
E_{rec}	10	10	20	20
Cost	100	11	400	21
Order	–	bQ	–	cP

Table 12.2: Parameters for the relation physical plans

Step 2

The expression $E_Q = \sigma_{bQ=13}(Q)$ with the minimum cost physical plan (Table 12.2) is expanded by considering the possible joins

$$E_{Q_1} = (\sigma_{bQ=13}(Q)) \underset{\text{PkQ = FkQ}}{\bowtie} (\sigma_{cP='AA'}(P))$$

$$E_{Q_2} = (\sigma_{cP='AA'}(P)) \underset{\text{FkQ = PkQ}}{\bowtie} (\sigma_{bQ=13}(Q))$$

Alternative physical plans for each expression are generated, and the cheapest plan is chosen (Figure 12.7).

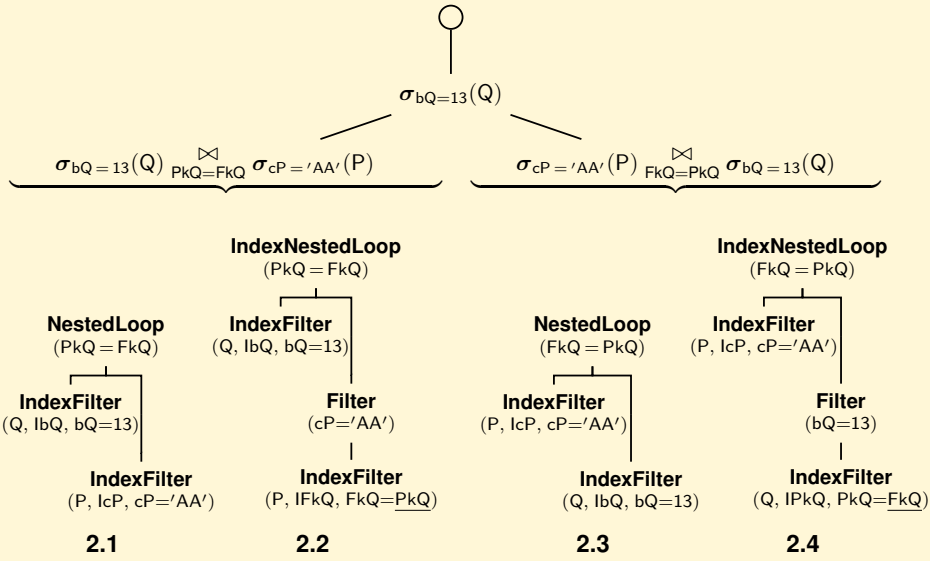


Figure 12.7: Search tree with physical plans for subexpressions with joins

The cost of E_{Q_1} with the *nested loop* (plan 2.1) is:

$$C^{2.1}_{\text{NL}}(E_{Q_1}) = C(O_E) + E_{\text{rec}}(O_E) \times C(O_I) = 11 + 10 \times 21 = 221$$

The cost of E_{Q_1} with the *index nested loop* (plan 2.2), and the index on FkQ of P, is:

$$C^{2.2}_{\text{INL}}(E_{Q_1}) = C(O_E) + E_{\text{rec}}(O_E) \times C(O_I)$$

The physical plans cost $C(O_I)$ with the index on FkQ is:

$$\begin{aligned} C(O_I) &= \lceil f_s(\text{FkQ = }) \times N_{\text{leaf}}(\text{IFkQ}) \rceil + \\ &\qquad \Phi(\lceil N_{\text{rec}}(\text{P}) / N_{\text{key}}(\text{IFkQ}) \rceil, N_{\text{pag}}(\text{P})) \\ &= \lceil 25/200 \rceil + \Phi(10, 400) = 1 + 10 = 11 \end{aligned}$$

and so

$$C^{2.2}_{\mathsf{INL}}(E_{Q_1}) = 11 + 10 \times 11 = 121$$

With both the join methods the result size is:

$$E_{\mathsf{rec}} = \lceil f_s(\mathsf{CD} = 13) \times f_s(\mathsf{cP = 'AA'}) \times f_s(\mathsf{PkQ = FkQ}) \times N_{\mathsf{rec}}(\mathsf{Q}) \times N_{\mathsf{rec}}(\mathsf{P}) \rceil = 1$$

Proceeding in a similar way for the join expression E_{Q_2}, we have the following costs for the physical plans with the *nested loop* and *index nested loop*, using the index on the primary key PkQ of Q:

$$C^{2.3}_{\mathsf{NL}}(E_{Q_2}) = C(O_E) + E_{\mathsf{rec}}(O_E) \times C(O_I) = 21 + 20 \times 11 = 241$$

$$C^{2.4}_{\mathsf{INL}}(E_{Q_2}) = C(O_E) + E_{\mathsf{rec}}(O_E) \times C(O_I) = 21 + 20 \times 2 = 61$$

With both the join methods the result size is $E_{\mathsf{rec}} = 1$, as in the previous case.

In conclusion, the best join physical plan is 2.4 for E_{Q_2}, with the method *index nested loop*, and cost 61 (Table 12.3).

	2.1	**2.2**	**2.3**	**2.4**
Plan	Fig. 12.7	Fig. 12.7	Fig. 12.7	Fig. 12.7
E_{rec}	1	1	1	1
Cost	221	121	241	61
Order	bQ	bQ	cP	cP

Table 12.3: Parameters for the join physical plans

Step 3

Since the possible joins have already been considered in the previous step, new subexpressions to consider are not generated. The *greedy search* algorithm terminates with the final physical query plan shown in Figure 12.8.

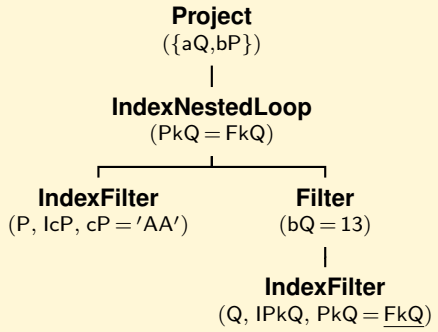


Figure 12.8: Final physical plan for the query of Example 12.9

12.4.4 Other Types of Queries

Let us show how to optimize a query when other clauses are used which have not yet been taken into consideration.

Optimization in DBMSs. It is easy to imagine that commercial systems devote much attention to the query optimization, which are, with the transaction manager, among the more complex DBMS modules to implement. Some systems are not limited to producing only left-deep plans. Oracle and Sybase ASE allow users to force the choice of join orders and indexes, while DB2 allows only the choice of the optimization level, for example using the algorithm *greedy*. JRS generates by default left-deep plans, but it is possible to select bushy trees and the level of optimization to use.

1. **DISTINCT** to specify duplicate elimination.
2. **GROUP BY** to group the result of a **SELECT** and to compute aggregation functions.
3. Queries with set operations.

12.4.5 Queries with SELECT DISTINCT

If the **DISTINCT** clause is necessary, assuming that the operation is performed by sorting, the physical plan for a **SELECT ORDER BY** is generated, and then extended with the physical operator **Distinct**(O).

Example 12.10
Let us consider the query

```
SELECT  DISTINCT aR
FROM    R
WHERE   cR > 100;
```

The physical plan is found as follows:

1. First the physical plan for the following query is found:

   ```
   SELECT    aR
   FROM      R
   WHERE     cR > 100
   ORDER BY  aR;
   ```

2. Then the physical plan is extended with the operator **Distinct** (Figure 12.9).

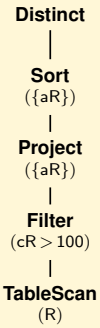


Figure 12.9: A physical plan for the **SELECT DISTINCT**

12.4.6 Queries with GROUP BY

If the **GROUP BY** is necessary, the optimizer produces a physical plan with the physical operator **GroupBy** or **HashGroupBy** as follows:

1. The best physical plan P is found to execute the subquery to produce the data to be grouped.
2. The best extensions of P is found to execute the grouping of its result:
   (a) with **GroupBy** and P is extended with **Sort** of the grouping attributes,
   (b) with **HashGroupBy** only.
3. Finally, if the **SELECT** has also a **HAVING** clause, the physical plan is extended with a selection operator, and the selected operator **GroupBy** or **HashGroupBy** computes all the aggregate functions used in the **HAVING** and **SELECT** clauses.

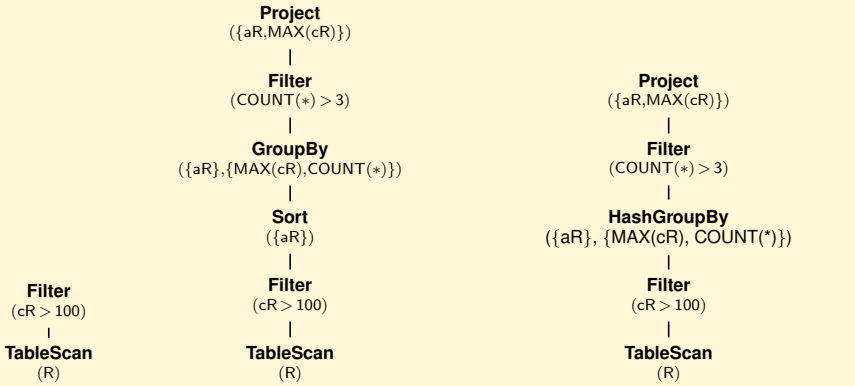
The result of **HashGroupBy** is not sorted, but usually a plan with **HashGroupBy** costs less than that with **GroupBy**, even in queries with **ORDER BY**, but when the **consolidation ratio** is high, i.e. the ratio of the number of rows of the operand and that of the grouping. For example, if the grouping concerns a very large table and the primary key or attributes with a few different values are used in the grouping, the consolidation ratio is low and the **GroupBy** is used.

Example 12.11
Let us consider the query:

```
SELECT    aR, MAX(cR)
FROM      R
WHERE     cR > 100
GROUP BY  aR
HAVING    COUNT(*) > 3;
```

The physical plan is generated as follows:

1. Let P be the physical plan generated by the optimizer to produce the data to be grouped (Figure 12.10a):
2. The physical plan P is extended with the operators **Sort** and **GroupBy** (Figure 12.10b) or with **HashGroupBy** only (Figure 12.10c) .
3. A **Filter** is added to take account of the clause **HAVING** .
4. A **Project** is added to produce the final result.



(a) *Plan P* (b) *Final physical plan with* **GroupBy** (c) *Final physical plan with* **HashGroupBy**

Figure 12.10: GROUP BY physical plans

Example 12.12
Let us see some examples of *physical plans* generated by the JRS cost-based query optimizer using different optimization strategies. Examples are given using the database schema in Figure 12.11 and the following query:

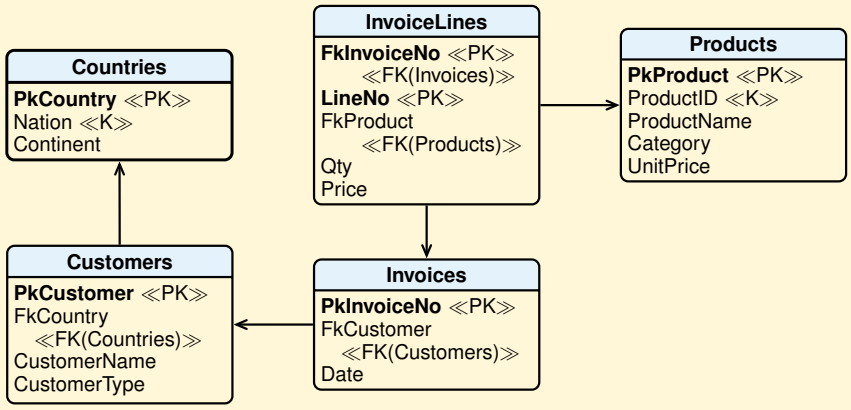


Figure 12.11: The database schema

```
SELECT    Nation, Category, SUM(Qty*Price) AS Revenue
FROM      Countries CNT, Customers C, Products P, Invoices I, InvoiceLines IL
WHERE     CNT.PkCountry = C.FkCountry
  AND     IL.FkProduct = P.PkProduct
  AND     IL.FkInvoiceNo = I.PkInvoiceNo
  AND     C.PkCustomer = I.FkCustomer
GROUP BY  Nation, Category
HAVING    COUNT(*) > 1;
```

The following information about the relations are available:

	Countries	Customers	Products	Invoices	InvoiceLines
N_{rec}	51	338	200	868	2141
N_{pag}	3	25	13	36	88

Table 12.4 compares the physical plans cost generated using the optimization strategies available and the physical operator **GroupBy** or **HashGroupBy**.

Optimization Strategy	Type of Query Plan Tree	Query Plan Cost GroupBy	Query Plan Cost HashGroupBy	Result Size
Greedy Search	Left-Deep	11 144	9 134	24
Greedy Search	Bushy	8 497	6 487	24
Full Search	Left-Deep	11 144	9 134	24
Full Search	Bushy	5 535	3 525	24
Iterative Full Search	Left-Deep	11 144	9 134	24
Iterative Full Search	Bushy	5 535	3 525	24

Table 12.4: A comparison of query plans

By default, the optimization strategy uses the heuristics *greedy search* and *left-deep* query plans, which produce, in this example, the same plan of the *full search* and *iterative full search* strategies, while in general they produce different *left-deep* query plans with lower costs (Figure 12.12).

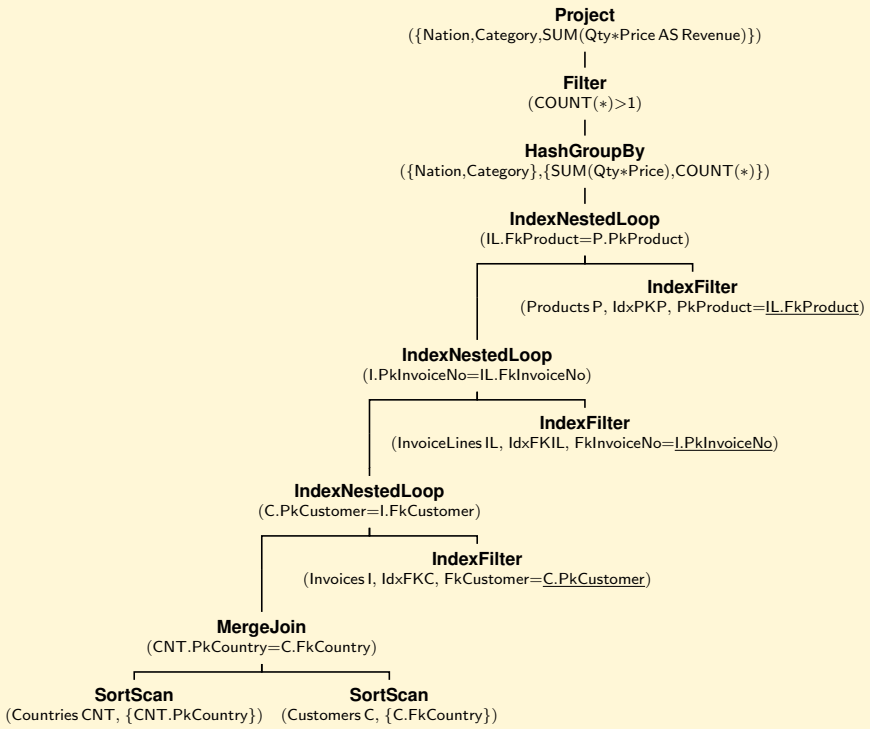


Figure 12.12: A left-deep query plan with the greedy search strategy

Better plans are produced by choosing the option *bushy* query plan, which is the most general type of plan (Figure 12.13).

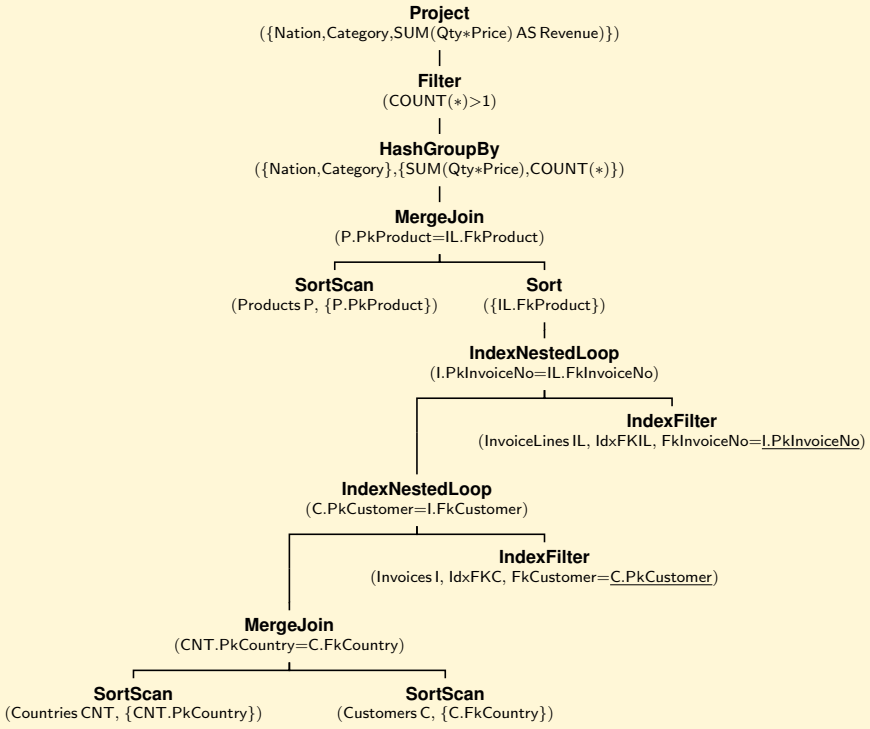


Figure 12.13: A bushy query plan with the greedy search strategy

The best plan is produced by the *full search* strategy (Figure 12.14).

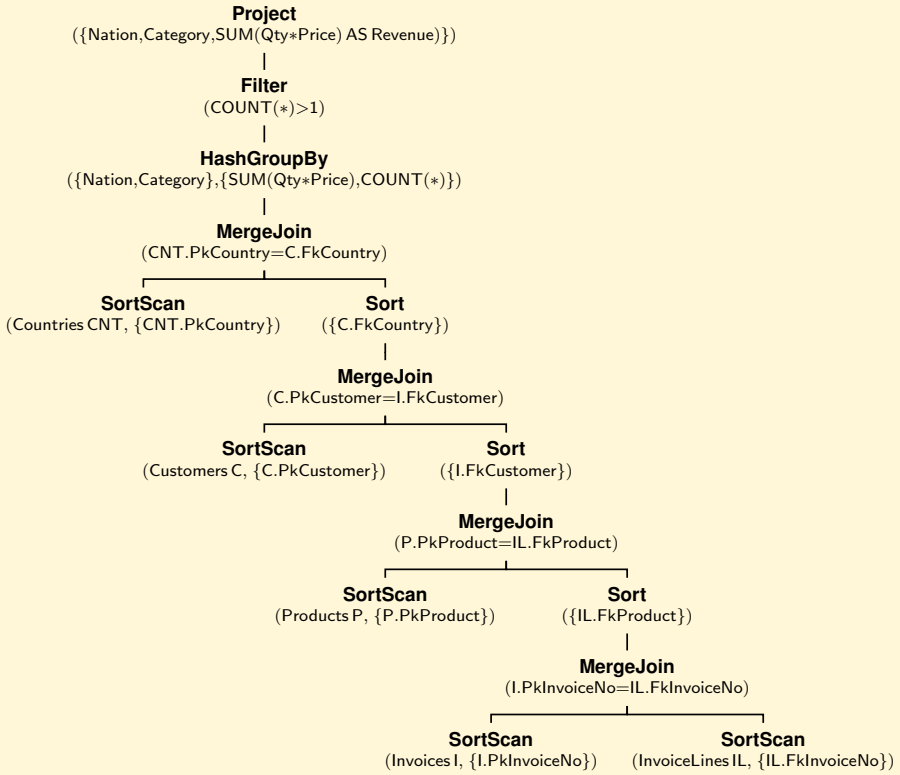


Figure 12.14: A bushy query plan with the full search strategy

Pre-Grouping Transformation

The standard way to evaluate join queries with grouping and aggregations is to perform the joins first. However, several authors have shown that the optimizer should also consider doing the grouping and aggregations before the join to produce cheaper physical query plans.

For simplicity, let us show a solution of the problem by making the following assumptions:

- The relations do not have attributes with null values.
- The joins are equi-joins using the primary and foreign keys with only one attribute, and all are *necessary* to produce the query result, therefore they can not be eliminated with a query rewriting.
- The aggregate functions are in the form AGG(attribute) AS Ide, with AGG = SUM, MIN, MAX, COUNT, AVG. The aggregation attributes are different from those for grouping.
- The grouping attributes and aggregate functions may be renamed with the operator AS.
- The problem of the pre-grouping transformation is presented by considering the logical query plan and assuming that the optimizer chooses the best physical plan for two logical plans with or without the pre-grouping.

Let us consider first some basic equivalence rules on relational algebra expressions involving the grouping operator $\boldsymbol{\gamma}$, which will also be useful for proving more complex ones later on. For brevity, we only give an informal justification of these equivalence rules.

1. A restriction can be moved before the grouping operator in the following cases.
   (a) If θ uses only attributes from X and F is a set of aggregate functions that use only attributes from E, then

   $$\boldsymbol{\sigma}_{\theta}({}_{X}\boldsymbol{\gamma}_{F}(E)) \equiv {}_{X}\boldsymbol{\gamma}_{F}(\boldsymbol{\sigma}_{\theta}(E)) \tag{12.1}$$

   The restriction of the left hand side eliminates a record r from the $\boldsymbol{\gamma}$ result if and only if it eliminates all the records from the group that has generated r.
   (b) If X and B are attributes from E, with $B \notin X$, v is a B value and MIN *is the only aggregate function*, then

   $$\boldsymbol{\sigma}_{\mathsf{mB}<v}({}_{X}\boldsymbol{\gamma}_{\mathsf{MIN}(B)\,\mathsf{AS}\,\mathsf{mB}}(E)) \equiv {}_{X}\boldsymbol{\gamma}_{\mathsf{MIN}(B)\,\mathsf{AS}\,\mathsf{mB}}(\boldsymbol{\sigma}_{B<v}(E)) \tag{12.2}$$

   If the restriction of the left hand side eliminates a record r such that the attribute value $r.\mathsf{mB}$ is the minimum among those of its group, then each record r_i of the group has $r_i.B \geq r.\mathsf{mB} \geq v$, and therefore all of them will be eliminated by the restriction of the right hand side. The equivalence also holds if $<$ is $\leq$.
   (c) If X and B are attributes from E, with $B \notin X$, v is a B value and MAX *is the only aggregate function*, then

   $$\boldsymbol{\sigma}_{\mathsf{MB}>v}({}_{X}\boldsymbol{\gamma}_{\mathsf{MAX}(B)\,\mathsf{AS}\,\mathsf{MB}}(E)) \equiv {}_{X}\boldsymbol{\gamma}_{\mathsf{MAX}(B)\,\mathsf{AS}\,\mathsf{MB}}(\boldsymbol{\sigma}_{B>v}(E)) \tag{12.3}$$

   If the restriction of the left hand side eliminates a record r such that the attribute value $r.\mathsf{MB}$ is the maximum of those of its group, then each record r_i of the group has $r_i.B \leq r.\mathsf{MB} \leq v$, and therefore all of them will be eliminated by the restriction of the right hand side. The equivalence also holds if $>$ is $\geq$.
2. If X and Y are attributes from E, $Y \not\subseteq X$, $X \rightarrow Y$ and F is a set of aggregate functions that use only attributes from E, then

   $${}_{X}\boldsymbol{\gamma}_{F}(E) \equiv \pi^{b}_{X \cup F}({}_{X \cup Y}\boldsymbol{\gamma}_{F}(E)) \tag{12.4}$$

   Each record of a group that has generated a record r of the $\boldsymbol{\gamma}$ in the left hand side belongs to the same group that has generated a record r of the $\boldsymbol{\gamma}$ in the right hand side.
3. An aggregate function f is called *decomposable* if there is a *local* aggregate function f_l and a *global* aggregate function f_g, such that for each multiset V and for any partition of it $\{V_1, V_2\}$ we have

   $$f(V_1 \cup^{all} V_2) = f_g(\{f_l(V_1), f_l(V_2)\})$$

   where $\cup^{all}$ is the SQL's union-all operator without duplicate elimination.
   For example, SUM, MIN, MAX and COUNT are decomposable:
   - $\mathsf{SUM}(V_1 \cup^{all} V_2) = \mathsf{SUM}(\{\mathsf{SUM}(V_1), \mathsf{SUM}(V_2)\})$
   - $\mathsf{MIN}(V_1 \cup^{all} V_2) = \mathsf{MIN}(\{\mathsf{MIN}(V_1), \mathsf{MIN}(V_2)\})$
   - $\mathsf{MAX}(V_1 \cup^{all} V_2) = \mathsf{MAX}(\{\mathsf{MAX}(V_1), \mathsf{MAX}(V_2)\})$
   - $\mathsf{COUNT}(V_1 \cup^{all} V_2) = \mathsf{SUM}(\{\mathsf{COUNT}(V_1), \mathsf{COUNT}(V_2)\})$[3]

If the aggregate functions in F are decomposable, X and Y are attributes from E, $Y \not\subseteq X$, $X \not\rightarrow Y$, then

$$ {}_X\gamma_F(E) \equiv {}_X\gamma_{F_g}({}_{X\cup Y}\gamma_{F_I}(E)) \tag{12.5}$$

Adding new grouping attributes to the internal γ of the right hand side produces smaller groups on which the local aggregate functions are computed, but then external γ combines these partial results to compute the global aggregate functions, and the result is the same as the γ of the left hand side.

Invariant Grouping. For simplicity we consider algebraic expressions of the type ${}_X\gamma_F(R \overset{\bowtie}{}_{C_j} S)$. Any local selections on relations of the join are always pushed-down and do not affect the pre-grouping transformations, for this reason they will be ignored in the following considerations.

Let $\mathcal{A}(\alpha)$ be the set of attributes in α and $R \overset{\bowtie}{}_{f_k=p_k} S$ an equi-join using the foreign key f_k of R and the primary key p_k of S.

Theorem 12.3

R has the *invariant grouping* property

$$ {}_X\gamma_F(R \overset{\bowtie}{}_{f_k=p_k} S) \equiv \pi^b_{X\cup F}(({}_{X\cup\mathcal{A}(C_j)-\mathcal{A}(S)}\gamma_F(R)) \overset{\bowtie}{}_{f_k=p_k} S) \tag{12.6}$$

if the following conditions hold:

1. $X \rightarrow f_k$, i.e. the foreign key of R is *functionally determined* by the grouping attributes X in $R \overset{\bowtie}{}_{f_k=p_k} S$.
2. Each aggregate function in F only uses attributes from R.

Example 12.13
Let us consider the database, where keys are underlined:

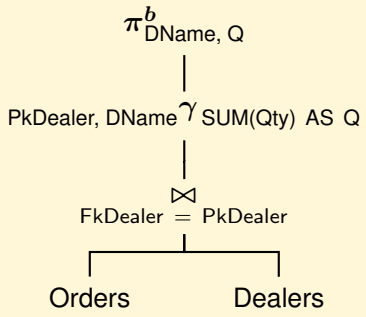
Products(PkProduct, PName, PCost, PCategory)
Orders(PkOrder, FkProduct, FkDealer, Price, Qty, Date)
Dealers(PkDealer, DName, DState, DAddress)

and the query

```
SELECT    DName, SUM(Qty) AS Q
FROM      Orders, Dealers
WHERE     FkDealer = PkDealer
GROUP BY  PkDealer, DName;
```

The logical plan without the pre-grouping is

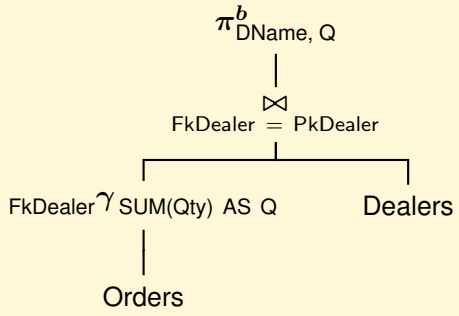
3. AVG is not decomposable according the given definition, but it can be computed with a different rule from two local functions SUM and COUNT:
AVG($V_1 \cup^{all} V_2$) = SUM({SUM(V_1), SUM(V_2)}) / SUM({COUNT(V_1), COUNT(V_2)})



To decide whether Orders has the invariant grouping property, since the Condition 2 holds, let us check whether (PkDealer, DName) $\rightarrow$ FkDealer:

$$(\text{PkDealer, DName})^+ = \{\text{PkDealer, DName, FkDealer, } \ldots\}$$

Since (PkDealer, DName)$^+$ contains FkDealer, (PkDealer, DName) $\rightarrow$ FkDealer holds and the γ can be pushed below the join on Orders. Therefore, we also get the following logic plan that the optimizer will consider in choosing the best physical query plan.



Example 12.14
Let us consider the database schema in Figure 12.11 and the following query:

```
SELECT      FkProduct, SUM(Qty) AS TotalQty
FROM        InvoiceLines, Products
WHERE       FkProduct = PkProduct
GROUP BY    FkProduct
HAVING      COUNT(*) = 1;
```

The JRS *cost-based query optimizer* does not consider the possibility of doing the group-by before the join, therefore, as usually happens in relational DBMSs, the standard way to evaluate a query with **HashGroupBy** is to first retrieve the records required by the operator, and then to execute it in order to produce the final result. The optimizer generates the physical plan in Figure 12.15 with the physical operator **NestedLoop** for join. A double click on the root node of the plan displays information about the operator involved, the estimated number of rows produced by the operator, and the estimated cost of the operation, which in this case is 27 921.

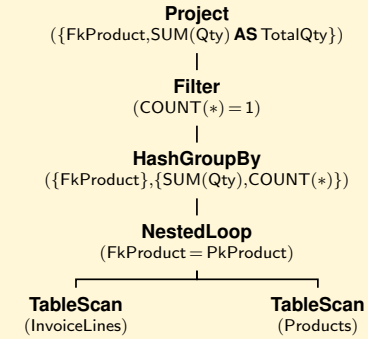


Figure 12.15: Physical query plan generated by the optimizer

Let us now design manually with JRS a physical plan that exploits the rule for pushing the group-by below the join (Figure 12.16). A double click on the root node of the plan displays information about the estimated cost of the operation, which in this case is 2 688, much lower than that of the plan generated by the optimizer.

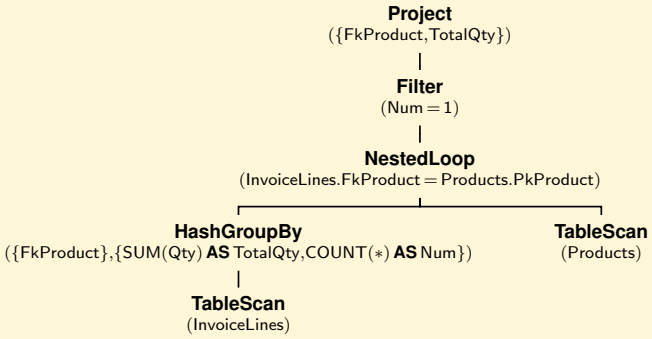


Figure 12.16: Physical query plan designed manually

12.4.7 Queries with Set Operations

Let us consider, for simplicity, only the case of the **UNION** operator, and the other set operators are treated in a similar way.

Suppose that the following physical operator exists to perform the set union operation:

Union(O_E, O_I) assuming that the operand records are sorted and without duplicates.

The optimizer is used to generate the physical plans for the two **SELECT** of the **UNION**, with duplicate elimination operators, which then become the operands of the **Union** operator.

When a set operator is used with the option **ALL**, the physical plans of the operand are without the duplicate elimination operators.

Example 12.15
Consider the query

```
SELECT  aR
FROM    R
WHERE   cR < 50
  UNION
SELECT  aR
FROM    R
WHERE   cR > 100;
```

The physical plan is generated with the following steps:

1. The plans of the two **SELECT**, extended with the clause **ORDER BY**, are generated.

```
SELECT    aR
FROM      R
WHERE     cR < 50
ORDER BY  aR;
```

```
SELECT    aR
FROM      R
WHERE     cR > 100
ORDER BY  aR;
```

2. If necessary, the physical plan of the two **SELECT** is extended with the physical operator **Distinct** to eliminate duplicates.
3. The physical operator **Union** is added to produce the final result (Figure 12.17).

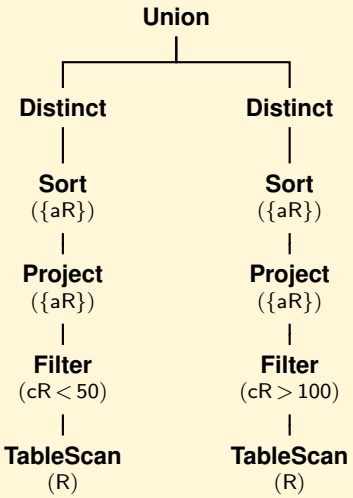


Figure 12.17: Physical plan for UNION

12.5 Summary

1. To execute an SQL query, it is first represented internally as a logical query plan, then is rewritten to normalize the condition, to perform selections and projections operations as early as possible, to eliminate subquery and views. Finally, the optimizer chooses the best physical query plan.
2. Physical operators implement algorithms to execute relational algebra operators. In general each logical operator can be implemented with several physical operators (e.g. a join can be implemeted with the physical operators *nested loops*, *page nested loops*, *index nested loops*, *merge-join*, *hash join*) and the optimizer chooses the most convenient.
3. The optimizer goal is to avoid expensive physical query plans and to look for a good plan, even if it is not the best, using appropriate heuristics. Examples of heuristics are: selection pushing to reduce the size of partial results, the use of joins of left-deep trees only.
4. The optimizer operates on algebraic expressions of the type $\pi(\sigma(\bowtie))$ with the option of providing the result sorted on some attributes of the projection. To deal with other SQL clauses, the relational algebra provides other operators and the initial physical plan is extended with the corresponding physical operators to have the final physical query plan.

Bibliographic Notes

The optimization of relational queries is presented in all books cited in Chapter 1, in particular in [Garcia-Molina et al., 1999], [Ramakrishnan and Gehrke, 2003] and [Graefe, 1993] where the use of iterators in query plans is discussed. An object-oriented implementation of the optimizer is presented in [Kabra and DeWitt, 1999; Lanzelotte and Valduriez, 1991]. For an interesting approach to query transformation see [Pirahesh and Hellerstein, 1992]. A review of other algorithms to optimize the performance of joins between more than 10-15 tables is presented in [Kossmann and Stocker, 2000].

The solutions for the *System R* and the *INGRES* system, the first relational systems to address the problem, are discussed in [Selinger et al., 1979] and [Wong and Youssefi, 1976]. Performance considerations about the optimizers for the two systems are shown in [Astrahan et al., 1980; Hawthorn and Stonebraker, 1979]. The solution for the Oracle system appears in [Antoshenkov and Ziauddin, 1996].

Exercises

Exercise 12.1 Briefly answer the following questions:

1. Define the term *selectivity factor*.
2. Explain the role of *interesting orders* in a *System R* like optimizer.
3. Describe *left-deep plans* and explain why optimizers typically consider only such plans.

Exercise 12.2 Consider the following schema, where keys are underlined:

Students(Id, Name, BirthYear, Status, Other)
Transcripts(Subject, StudId, Grade, Year, Semester)

Consider the following query:

```
SELECT   Subject, Grade
FROM     Students, Transcripts
WHERE    BirthYear = 1990 AND Id = StudId
```

Suppose that a clustered B^+–tree index on StudId is available.

Show two physical plans, and the estimated costs, one with the use of the join physical operator **IndexNestedLoop** and the other with the join physical operators **MergeJoin**.

Exercise 12.3 Consider the following schema, where the keys are underlined:

Students(Id, Name, BirthYear, Status, Other)
Transcripts(Subject, StudId, Grade, Year, Semester)

Consider the following query:

```
SELECT   Subject, COUNT(*) AS NExams
FROM     Students, Transcripts
WHERE    Year = 2012 AND Id = StudId
GROUP BY Subject
HAVING   AVG(Grade) > 25;
```

1. Suppose that no indexes are available. Show the physical plan with the lowest estimated cost.
2. If there is a B^+–tree index on Subject, and a B^+–tree index on Id, what is the physical plan with lowest estimated cost?

Exercise 12.4 Consider the following schema, where the keys are underlined (different keys are underlined differently):

Customer(PkCustPhoneNo, CustName, CustCity)
CallingPlans(PkPlanId, PlanName)
Calls(PkCustPhoneNo, FkPlanId, Day, Month, Year, Duration, Charge)

where PkPlanId e PlanName are two different keys, and the following query

```
Q:  SELECT   Year, PlanName, SUM(Charge) AS TC
    FROM     Calls, CallingPlans
    WHERE    FkPlanId = PkPlanId AND Year >= 2000 AND Year <=2005
    GROUP BY Year, PlanName
    HAVING   SUM(Charge) > 1000;
```

Give the initial logical query plan. Can the **GROUP BY** be pushed on the relation Calls?

Exercise 12.5 Consider the following schema without null values, where the keys are underlined:

Customer(PkCustomer, CName, CCity)
Order(PkOrder, FkCustomer, ODate)
Product(PkProduct, PName, PCost)
OrderLine(LineNo, FkOrder, FkProduct, Quantity, ExtendedPrice, Discount, Revenue)

Consider the following query:

```
SELECT      CCity, AVG(Revenue) AS avgR
FROM        OrderLine, Order, Customer
WHERE       FkOrder = PkOrder AND FkCustomer = PkCustomer
GROUP BY    CCity, FkCustomer
HAVING      SUM(Revenue) > 1000;
```

Give the initial logical query plan and show how the **GROUP BY** can be pushed on the join (OrderLine $\bowtie$ Order).

Can the **GROUP BY** be pushed on the relation OrderLine?

Exercise 12.6 Consider the following schema with attributes of type integer without null values, where the keys are underlined:

R(PkR, FkS, RC)
S(PkS, SE)

Show how the following query Q can be rewritten in SQL without the use of the view V.

```
CREATE      VIEW V AS                    Q:  SELECT      SE, SUM(N) AS SN
SELECT      FkS, COUNT(*) AS N               FROM        V, S
FROM        R                                WHERE       FkS = PkS;
GROUP BY    FkS;                             GROUP BY    SE;
```

Exercise 12.7 Consider the following schema, where the keys are underlined:

R(PkR integer, FkRS integer, RA varchar(10), RB integer)
S(PkS integer, SA varchar(20), SB varchar(10), SC varchar(10))
T(FkTS integer, TA integer, TB integer)
FkTS is both a primary key for T and a foreign key for S.

and the following query:

```
SELECT      SA, TA
FROM        R, S, T
WHERE       FkRS = PkS AND PkS = FkTS
            AND SC = 'P' AND RB > 130 AND RA = 'B';
```

Suppose the following indexes exist on the relations:

- R: four unclustered B^+–tree indexes on PkR, FkRS, RA and RB.
- S: two unclustered B^+–tree indexes on PkS and SC.
- T: one unclustered B^+–tree index on FkTS.

The following information about the relations are available:

	S	R	T
N_{rec}	300	10 000	300
N_{pag}	66	110	18
N_{key}(IdxPkS)	300		
N_{key}(IdxSC)	15		
N_{key}(IdxPkR)		10 000	
N_{key}(IdxFkRS)		300	
N_{key}(IdxRB)		50 (min = 70, max = 160)	
N_{key}(IdxRA)		200	
N_{key}(IdxFkTS)			300

Assume that the DBMS uses only the join physical operators **NestedLoop** and **IndexNestedLoop**, and only one index for selections.

1. Give the query logical plan that the optimizer uses to find the physical plan.
2. Assuming that the optimizer uses a greedy search algorithm, give an estimate of the cost and of the result size of the physical plan for each relation, approximating and index access cost with only the C_D.
3. Which join physical plan for two relations will be selected in the second query optimization step?
4. What is the cost and the result size of the final best physical query plan?

Chapter 13

PHYSICAL DATABASE DESIGN AND TUNING

After having seen the main solutions for the storage of databases, for the management of transactions and for query optimization, useful for those who have to implement systems for the management of data, in this chapter we show how this knowledge is also important for those who develop applications that use databases, or manage databases, and must ensure that applications run with the expected performances. To accomplish this task it is necessary to know how applications use the data in order to solve various types of problems that relate to (a) the physical design of the initial database, to choose the most suitable storage structures and, if the performance is not yet satisfactory, proceed with (b) tuning the database, to revise the solutions adopted at the physical, logical and application levels, and (c) tuning the system.

13.1 Physical Database Design

The design of a relational database proceeds in four phases:

1. ***Requirements Analysis***, for the specification of data to be taken into account and the services to be implemented;
2. ***Conceptual Design***, for the conceptual representation of the entities of interest for the purposes of the operational activities, their properties, associations among entities and the integrity constraints (*business rules*).
3. ***Logical Design***, for the definition of the DB logical schema, typically a normalized relational schema with controlled duplication, and appropriate integrity constraints to prevent improper data updates.
4. ***Physical Design***, for the definition of appropriate storage structures for a specific DBMS, to ensure the application performance desired, taking into account the resources available.

The physical design is driven by the statistics on the database relations and the operations to be carried out on them. In particular, it is important to know the following information.

Definition 13.1 *Statistics*

The *statistics on the database relations* are the following:

- For each relation, the number of records, the record size and the number of pages used.
- For each attribute of records, the size, the number of distinct values and the minimum and maximum value of each numeric attribute.
- The distribution of attribute values, if different from the uniform one.

Definition 13.2 *Workload Description*

A *workload description* contains

- The critical queries and their frequency.
- The critical modification operations and their frequency.
- The expected performance for critical queries.

To create a good physical design, the designer must have knowledge about the DBMS, especially data organizations, indexes and query processing techniques supported. Since the physical design of a database is an activity that depends on the characteristics of the specific DBMS, in the following we will provide a general overview of the main aspects to be taken into consideration, without focusing on a specific system, except to show some examples.

13.1.1 Workload Specification

The critical operations are those performed more frequently and for which a short execution time is expected. In the definition of the workload of a database, for each critical query it is necessary to specify

- the relations and attributes used;
- the attributes used in selection and join conditions;
- the selectivity factor of conditions.

Similarly, for each critical updates it is important to know

- the type (INSERT/DELETE/UPDATE);
- the attributes used in selection and join conditions;
- the selectivity factor of conditions.
- the attributes that are updated.

A simple way to specify this information is to use the ISUD (Insert, Select, Update, Delete) tables. A table of this type specifies, for each application, the execution frequency, the percentage of the data used and the type of operation that takes place on the various attributes of a relation. Based on this table, it is easy to see which operations are performed more frequently and those that affect large amounts of data.

Figure 13.1 shows an example of a ISUD table.

It is important to keep in mind that the commercial DBMSs such as DB2, Oracle and SQL Server, provide automated tools to analyze the workload of a database during normal use of the system.

Application	Frequency	Data	Attributes	
		(%)	Name	Salary
Wage	Montly	100	Select	Select
New employee	Daily	0.1	Insert	Insert
Delete employee	Montly	0.1	Delete	Delete
Update salary	Montly	10	Select	Update

Figure 13.1: Example of a ISUD table for the relation Employee(Name, Salary)

13.1.2 Decisions to be Taken

The purpose of the physical design is to establish

- the organization of the relations;
- the indexes on the relations.

If these choices do not allow the attainment of the desired performance, it is necessary to proceed as described later in the section on database tuning, in particular

- rewrite the critical queries to reduce their execution time;
- change the database logical schema, even giving up its normalization.

Storage structures for relations. Depending on the workload and the statistics on the database relations, one of the following solutions is chosen for relations:

- **Heap organization**, when there is little data or when queries use large data subsets.
- **Sequentiale organization**, when data is static and it is interesting to have it sorted by a key.
- **Hash organization**, when the most frequent operation on data is a record search by a key.
- **Index sequential organization**, when it is interesting to keep the data sorted by a key, even in the case of insertions, and the most frequent operation is a search for records both by a key and by a key range.

Index Selection. The main goal of an index is to avoid a relation scan when few records are looked for. The indexes on relations must be chosen so that they are used by the optimizer to execute the critical queries, but keeping in mind that they occupy memory and can slow the updates of the relations attributes on which they are defined. Of course there are cases in which the indexes are entirely unnecessary, as the following:

- Indexes on relations with few pages.
- Indexes on not very selective attributes.

In general, the problem of the selection of useful indexes does not have a trivial solution due to the large number of possibilities to be taken into consideration. For this reason, commercial systems provide tools to assist in the resolution of the problem, such as *DB2 Design Advisor*, *Oracle Access Advisor* and *SQL Server Tuning Advisor*.

In the absence of tools to choose indexes, one way to proceed is to consider, one at a time, the critical queries and, for each of them, consider the plan chosen by the optimizer, and then reconsider the choice of indexes.
Assuming that the system automatically builds indexes for primary keys and foreign keys, in general, the attributes to be considered for indexing are not only those appearing in the clause **WHERE**. For example:

- An index, possibly clustered, for range searches that derive the greatest benefit from a clustered index. In the case of range searches on different attributes it is necessary to consider the conditions' selectivity and the queries' frequency in deciding which index must be clustered.
- A multi-attribute (composite) index for *conjunctive conditions*: in this case the index can also be used for queries with conditions on prefixes of the index attributes.
- By analyzing the queries that would benefit from the use of an index, particular attention must be paid to *disjunctive conditions*. For example, having only one index on an attribute and an **OR** condition on two attributes, the index is not useful.
- Indexes are sorted on the key attributes, and so they are also useful on attributes used in queries with **ORDER BY** or that can be executed with physical operators on sorted data, such as **GROUP BY**, **DISTINCT** and set operators.
- Other indexes of interest are those defined on attributes that can be used to produce plans with physical operators **IndexOnly**, e.g. that use the index only, called also *covering index*, to find the query result without having to access the data file. Moreover the indexes are useful to execute joins with the operators **IndexNestedLoop** or **MergeJoin**.
- An index I_1 is unnecessary if it is *subsumed* by another I_2, i.e. because the attributes of I_1 are included as prefix within those of I_2. For example, an index on A is subsumed by an index on (A, B), and both are subsumed by an index on (A, B, C).
- Two indexes supporting two different queries can be *merged* into one index supporting both queries with similar performance (*index merging*). For example, consider the relation $R(A, B, C, \ldots)$ with attributes of type integer, and the queries:

```
SELECT DISTINCT A, B          SELECT A, C, COUNT(*)
FROM   R                      FROM   R
WHERE  A = 80;                GROUP BY A, C;
```

  An useful index for the first query is on the attributes (A, B), and one for the second query is on (A, C). The two indexes can be merged in one on (A, C, B).

13.1.3 Examples of Index Selection

Let us show some examples of index selection for queries on the following relational schema:

```
Lecturers(PkLecturer INTEGER, Name VARCHAR(20), ResearchArea VARCHAR(10)
          , Salary INTEGER, Position VARCHAR(10), FkDepartment INTEGER)
Departments(PkDepartment INTEGER, Name VARCHAR(20), City VARCHAR(10))
```

Table 13.1 shows the statistics on the relations, with indexes on the primary and foreign keys and on some attributes.

In the cost analysis, only data access is considered and it is assumed that the optimizer uses at most one index for each table.[1]

1. In the examples the cost estimates are those of the JRS system.

Table 13.1: Physical parameters

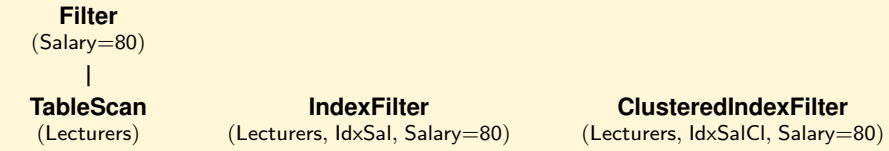
	Departments	Lecturers
N_{rec}	300	10 000
N_{pag}	30	1200
N_{key}(IdxSal)		50 (min = 40, max = 160)
N_{key}(IdxCity)	15	

Use of Clustered Indexes. Equality and range searches by a non-key attribute, are those that take greater advantage of the presence of clustered indexes, even with not very selective simple conditions. Here are some examples.

Example 13.1
Let us consider the query

```
SELECT  *
FROM    Lecturers
WHERE   Salary = 80;
```

With a selective simple condition the use of an index is always preferable, in particular when clustered, to a table scan. In our case, the interesting plans are:[2]

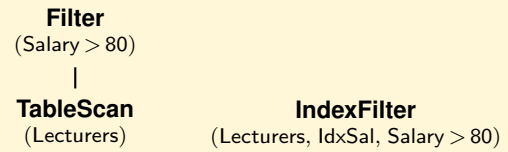


(a) *No index* (b) *With unclustered index* (c) *With clustered index*

– The plan without indexes has the cost 1200.
– The plan with the unclustered index has the cost 185.
– The plan with the clustered index has the cost 24.

The usefulness of an index with respect to the table scan depends on the condition selectivity and, therefore, on the number of records that satisfy the condition. With the decrease of the condition selectivity, an unclustered index quickly becomes less useful than scanning the table with the control of the condition on data, while a clustered index becomes less useful only with a very low selectivity.

For example, if the condition selectivity is low (Salary $>$ 80), the clustered index on Salary is useful because the records with Salary $>$ 80 are stored contiguously in the pages. The interesting plans are:



(a) *No index* (b) *With unclustered index*

ClusteredIndexFilter
(Lecturers, IdxSalCl, Salary > 80)

(c) *With clustered index*

– The plan without indexes has the cost 1200.
– The plan with the unclustered index has the cost 6290.
– The plan with the clustered index has the cost 800.

A clustered index is also useful when data is to be sorted, as in the case of **GROUP BY**, which is supposed to be performed with the sorting technique.

Example 13.2
Let us consider the query

```
SELECT   *
FROM     Lecturers
ORDER BY FkDepartment;
```

To obtain the sorted result, a clustered index on FkDepartment is useful, instead of one on Salary. The interesting plans are:

SortScan
(Lecturers, {FkDepartment})

(a) *No index*

IndexScan
(Lecturers, IdxFKDep)

(b) *With unclustered index*

ClusteredIndexScan
(Lecturers, IdxFKDep)

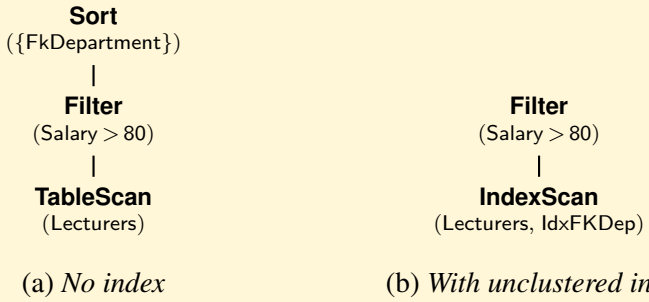
(c) *With clustered index*

– The plan without indexes has the cost 3600.
– The plan with the unclustered index on FkDepartment has the cost 9250.
– The plan with the clustered index on FkDepartment has the cost 1200.

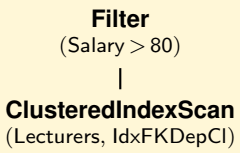
If the query also uses the simple condition on Salary, the best clustered index depends on the condition selectivity. Let us consider a query with a low selective condition.

```
SELECT   *
WHERE    Salary > 80
FROM     Lecturers
ORDER BY FkDepartment;
```

The interesting plans are:



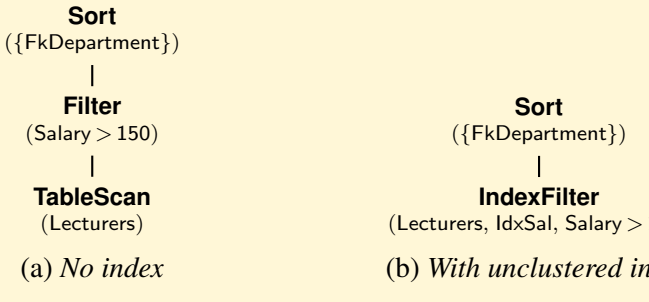
(a) *No index* (b) *With unclustered index*



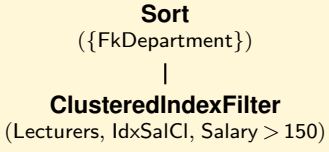
(c) *With clustered index*

– The plan without indexes has the cost 3600.
– The plan with the unclustered index on FkDepartment has the cost 9250.
– The plan with the clustered index on FkDepartment has the cost 1200.

If the simple condition is selective (Salary > 150), then the best clustered index becomes the one on Salary. The interesting plans are:



(a) *No index* (b) *With unclustered index*



(c) *With clustered index*

– The plan without indexes has the cost 1500.
– The plan with the unclustered index on Salary has the cost 1225.
– The plan with the clustered index on Salary, instead of the one on FkDepartment, has the cost 400.

From these considerations, one can imagine what the best type of index is if in the previous query there was a **GROUP BY** such as

```
SELECT    FkDepartment, SUM(Salary)
FROM      Lecturers
GROUP BY  FkDepartment;
```

with the different types of selection on Salary.

In conclusion, clustered indexes are very useful, but it is not enough to build traditional indexes on sorted data. The existence of this type of indexes must be known to the optimizer to estimate the cost of their use properly, as in *DB2* and *Sybase*. At the time of the creation of a clustered index on attributes X, the data in the table is first sorted on X and then the index is created.

Finally, note that the term *clustered index* is used with different meanings in other DBMSs. For example, in *SQL Server*, and *MySQL InnoDB Storage Engine* it means *a primary tree organization*, with the data stored in the leaves of a B^+–tree.

Multi-Attribute Indexes. In some cases, it is helpful to create indexes on multiple attributes. For example, referring to the relations of previous examples, to find the employees with (Position = 'R1' AND Salary = 70) an index on the attributes (Position, Salary) is preferable than an index on Position or on Salary. Multi-attribute indexes are interesting also in the case of range queries with selective conditions. For example, the performance of the following query

```
SELECT Name
FROM   Lecturers
WHERE  Position = 'R1' AND Salary BETWEEN 50 AND 70
```

is improved with an index of (Position, Salary).

When a multi-attribute index is created, it is important to pay attention to the order in which the index attributes are specified. An index on (Position, Salary) is also useful for a condition on the attribute Position only, but it is useless for one on Salary.

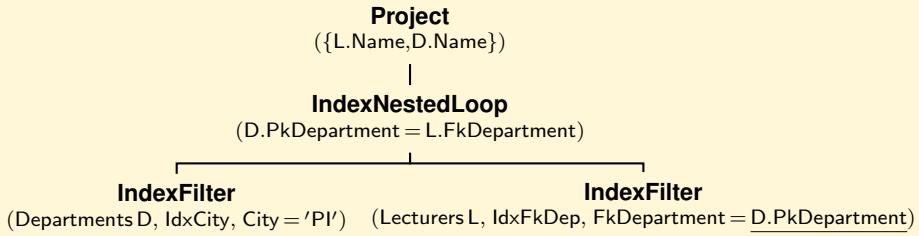
Finally, there are some general considerations about the use of multi-attribute indexes. Since each element of the index contains several attributes, there is a greater chance that the optimizer generates plans using only the index. On the other hand, a multi-attribute index is updated automatically after an update of any attribute on which it is defined, and it occupies more memory than an index on a single attribute.

Index for Join. Let us see some examples of how a clustered index on a non-key join attribute of the internal table is very useful in improving the performance of a **IndexNestedLoop**

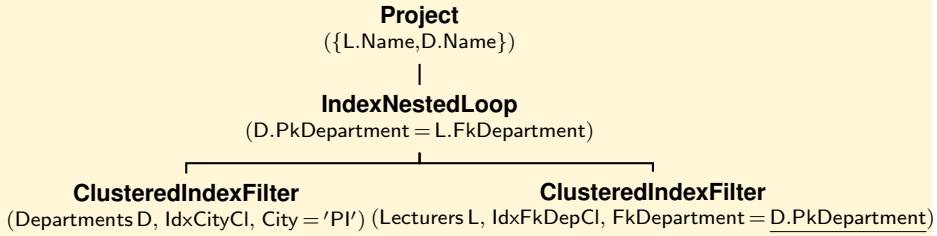
Example 13.3
Let us consider the query

```
SELECT L.Name, D.Name
FROM   Lecturers L, Departments D
WHERE  D.City = 'PI' AND L.FkDepartment = D.PkDepartment;
```

Since all conditions involve an equality comparison, in agreement with what has been said above, it is convenient to create two clustered indexes: one on City and another on FkDepartment to support the join with the method **IndexNestedLoop** and Departments as the outer table. The interesting plans are:



(a) *With unclustered index*

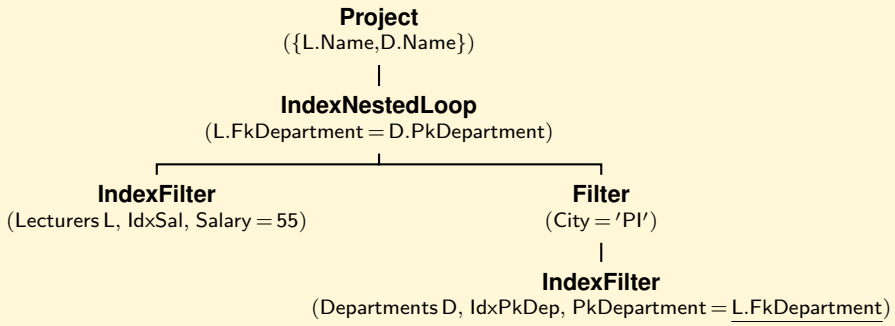


(b) *With clustered index*

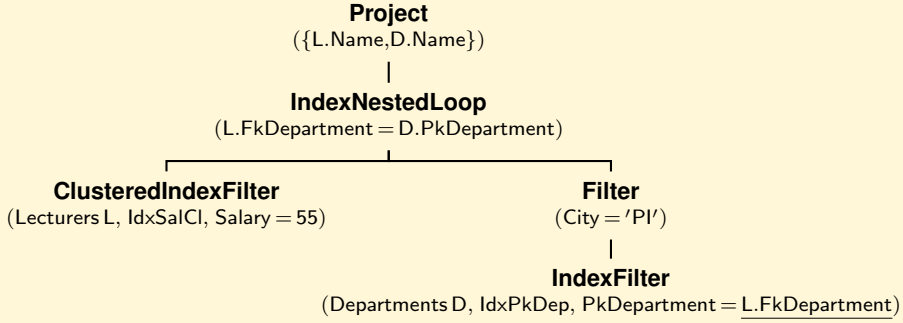
Figure 13.2: Examples of plans with City = 'PI'

– The plan with unclustered indexes has the cost 675.
– The plan with clustered indexes has the cost 82.

If in the **WHERE** clause there is also the selective simple condition (Salary = 55), and there is an index on Salary, then the interesting plans become:

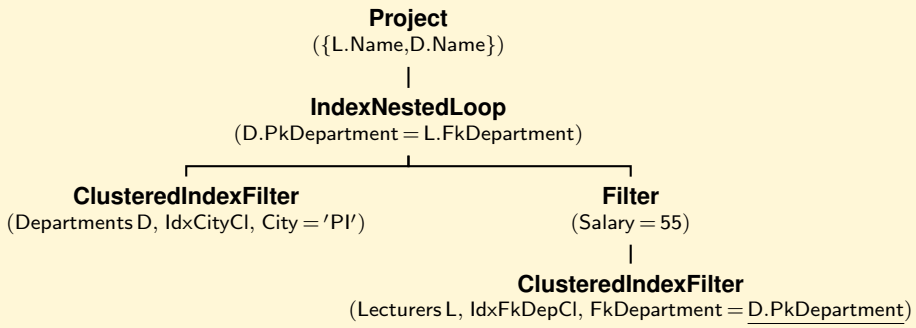


(c) *With unclustered index*



(d) *With clustered index on* City *and on* Salary

***Index-Only* Plans in DBMSs.** Oracle and DB2 support this kind of plans. SQL Server does not have this kind of physical operator. However the optimizer is able to recognize that all the information needed to produce the query result is available in the index that it is using, and so it generates a plan without table accesses.



(d) *With clustered index on* City *and on* FkDepartment

Figure 13.3: Examples of plans with City = 'Pl' and Salary = 55

- The plan with the unclustered indexes has the cost 385.
- The plan with the clustered indexes on City and on Salary has the cost 224.
- The plan with the clustered indexes on City and on FkDepartment has the cost 82.

Index-Only Plans. In some cases, it is possible to execute a query with *index-only* plans, e.g. that use indexes without accesses to the data.[3] In the case of queries on a single table, the index must be defined on the attributes that appear in the **SELECT** clause. Let us consider some examples.

```
SELECT  DISTINCT  FkDepartment
FROM    Lecturers;
```

An index on FkDepartment makes it unnecessary to access the relation Lecturers to find the different values of FkDepartment.

The index is also sufficient to perform the **GROUP BY** of the following query.

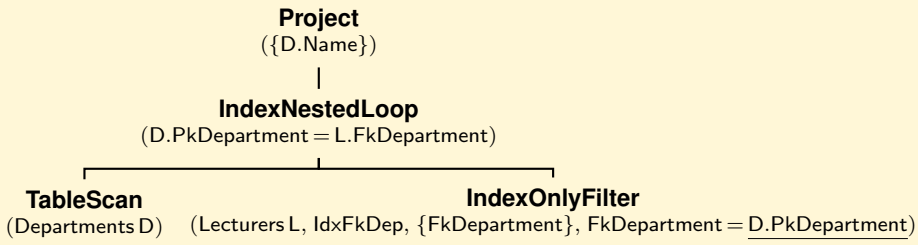
```
SELECT     FkDepartment, COUNT(*)
FROM       Lecturers
GROUP BY   FkDepartment;
```

Plans with an **IndexOnlyFilter** may also be used to perform joins:

```
SELECT  D.Name
FROM    Lecturers L, Departments D
WHERE   L.FkDepartment = D.PkDepartment;
```

3. In the JRS system, this happens when the operator IndexOnlyFilter is used.

An index on FkDepartment allows the generation of the following plan, which performs the join with a **IndexNestedLoop** and internal operand an **IndexOnlyFilter** with the index on FkDepartment of Lecturers, since all the information that is necessary to perform the join is available in the index.



Suppose we change the query by placing D.Name, I.Name in the **SELECT**.

In this case, to avoid accesses to the table Lecturers again, a multi-attribute index on (FkDepartment, Name) is necessary. Alternatively, SQL Server and DB2 allow the definition of an index only on FkDepartment, but it is also possible to included the attribute Name in the index, which cannot be used to make selections using the index, because it is not in the set of the index key attributes.[4]

With a multi-attribute index on (FkDepartment, Salary), the following query is also executed using an *index-only* physical operator.

```
SELECT   FkDepartment, MIN(Salary)
FROM     Lecturers
GROUP BY FkDepartment;
```

13.1.4 Concluding Remarks

The physical design of relational databases is a complex task and generally requires knowledge of the workload, the characteristics of the DBMS in use and, above all, the behavior of its optimizer. The choice of indexes is essential to speed up query execution, but remember that

- indexes must be useful, because they take up memory and need to be updated;
- indexes should be used for different types of queries;
- it is convenient to define indexes to obtain physical query plans that use indexes only;
- indexes are useful for physical operators on sorted data;
- the choice of clustered indexes is important, but only one of them can be defined for a relation;
- the order of the attributes in multi-attribute indexes is important.

4. The DB2 system only allows the addition of attributes to an index that are not part of a key with the command:

   CREATE UNIQUE INDEX *Name* **ON** *Table* (*KeyAttributes*)
   INCLUDE (*OtherAttributes*);

   The clause **UNIQUE** is instead optional in SQL Server.

13.2 Database Tuning

According to [Shasha and Bonnet, 2002] a definition of database tuning is

Definition 13.3

Database tuning comprises all activities carried out to meet requirements regarding the performance of database applications (i.e. queries or transactions).

The goal of database tuning is to *improve performance* of database applications, but performance may mean several things, and database tuning mostly refers to runtime performance and related aspects, e.g. *throughput* (number of applications that can be processes in a fixed time), *response time* (time to execute an application), *resource consumption* (temporary or permanent memory).[5]

In general, tuning is about the following aspects:

- *Applications*, e.g. user queries and transactions.
- *DBMS*, e.g. configuration and parameters, database schema, storage structures.
- *Operating System*, e.g. system parameters and configuration for IO, network communication.
- *Hardware*, e.g. used components for CPU, main memory, permanent memory.

Figure 13.4 shows the *Database Tuning Quadrilemma* presented by E. Schallehn.

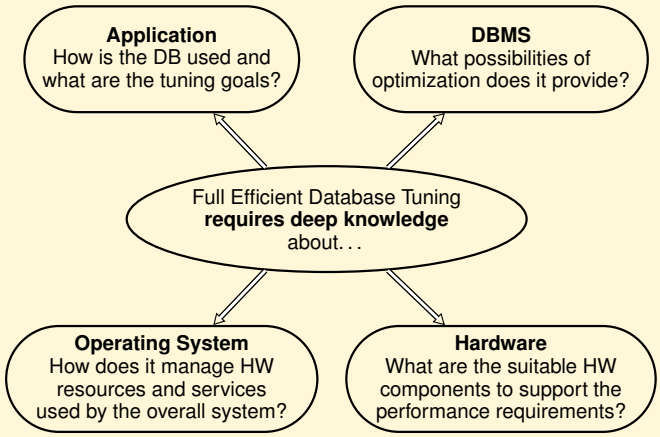


Figure 13.4: DBMS Tuning Quadrilemma

A database in use may have performance different from that expected. In this case, starting from quantitative data on the database and other measures provided by the tools monitoring the DBMS, it is necessary to intervene at different levels:

- Review the choice of indexes based on the physical query plans generated by the optimizer.

5. See the interesting Schallehn's slides on this topic, downloadable at this URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.173.8252&rep=rep1&type=pdf

– Review the formulation of queries that do not exhibit the expected performance.
– Review the critical transactions.
– Review the schema logical structure and define appropriate views for the logical data independence of application programs.

Database tuning is a process which begins with an analysis step to highlight the problems and decide which aspects are relevant. First, it is necessary to isolate the operations performed with slow response times, which can be either the critical ones or those reported by dissatisfied users. The next step is to examine the execution of queries with problems: the commercial DBMSs provide tools to analyze the performance of queries and their physical plans. In particular, looking at the physical plans, it is necessary to focus on

– the physical operators used for each relation;
– the order of intermediate results;
– the physical operators used in the plan to implement the logical operators.

Database tuning is a complex and *continuous* process, which restarts when application requirements or the overall system change, and is performed by the following professionals:

– *Database and application designers*, who have knowledge about applications and the DBMS, but may have only fair to no knowledge about OS and HW.
– *Database administrators*, who have mainly good knowledge about DBMS, OS, and HW.
– *Database experts*, who usually have strong knowledge about DBMS, OS and HW, but little knowledge about applications.

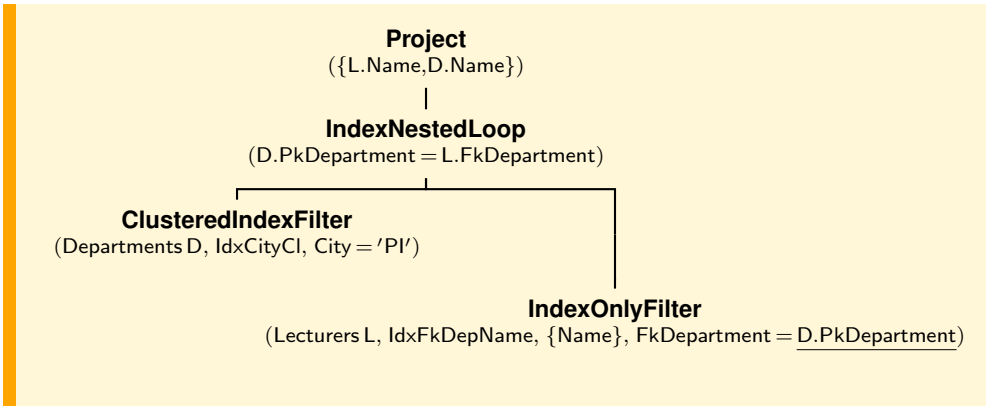
Nowadays their work is simplified because commercial DBMSs provide automated tools to support database tuning by "reducing the number of tuning knobs", such as *Oracle Self-Tuning Architecture* since Oracle 10g, *IBM Autonomic Computing Initiative*, and *Microsoft AutoAdmin* project for MS SQL Server.

13.2.1 Index Tuning

Among others, index tuning is one of the main tuning task. The initial choice of indexes can be revised for several reasons. The workload of the database used may reveal that some queries and update operations, initially considered important, do not really run very frequently, while there are other important ones which had not initially been taken into account. In some cases, it may also happen that the query optimizer does not generate the physical plan that was expected. For example, consider the query:

```
SELECT  L.Name, D.Name
FROM    Lecturers L, Departments D
WHERE   D.City = 'PI'  AND  L.FkDepartment = D.PkDepartment
```

A good physical query plan should use a clustered index on City, with Departments as the outer table, and perform the join with the method **IndexNestedLoop**, and an **IndexOnlyFilter** as an internal operand that uses the clustered index on FkDepartment of Lecturers, extended with the attribute Name. If the system in use does not allow the creation of indexes with the clause **INCLUDE**, the physical plan in Figure 13.2 with the cost 82 is obtained.



To overcome the DBMS's limitation, the choice of the indexes is reviewed by defining a clustered index on (FkDepartment, Name) for the relation Lecturers, and the following physical plan will be obtained with the cost 22.

Remember, finally, that the optimizer is based on statistics available in the system catalog, updated only on specific request with a command like **UPDATE STATISTICS**, which should be run periodically if the database is changed frequently.

13.2.2 Query Tuning

If the DBMS's optimizer does not transform SQL statements into forms that can be optimized more easily, queries might be rewritten as follows to improve performance:

1. Semantically equivalent rewriting.
   - Conjunctive conditions on the same attribute, if possible, are rewritten as conditions with **BETWEEN**. In this case the optimizer may use any index available to verify the condition, while with an **AND**, in some systems, the index is used only for one of the simple condition, while the other is then checked on the result.

     For the same reason, disjunctive conditions on the same attribute are rewritten with the operator **IN**.
   - If a disjunctive conditions on different indexed attributes ($p_1 \vee p_2$) are not evaluated using both indexes, it must be considered whether the response time improves by rewriting the query as a union of two that use p_1 and p_2 in the **WHERE** clause.
   - Subqueries are rewritten as joins.
   - Avoid writing complex queries by introducing temporary views. If the optimizer is not able to rewrite the query without the use of the views, in general the physical plan produced has a higher cost.
   - Avoid arithmetic expressions on indexed attributes used in conditions. For example, rewrite Salary*2 = 100 as Salary = 50, because in the first case an index on Salary may not be used.
   - Avoid useless **HAVING** or **GROUP BY** clauses; for example, the query

     ```
     SELECT   FkDepartment, MIN(Salary)
     FROM     Lecturers
     GROUP BY FkDepartment
     HAVING   FkDepartment = 10;
     ```

     is equivalent to the following one, for the equivalence rule on the *selection pushing*.

```
SELECT    FkDepartment, MIN(Salary)
FROM      Lecturers
WHERE     FkDepartment = 10
GROUP BY  FkDepartment;
```

If there was not FkDepartment in the **SELECT**, the **GROUP BY** also would become useless.

2. Semantically non-equivalent rewriting.
   - Avoid useless **DISTINCT**, in **SELECT** or subqueries, and **ORDER BY**.
   - Avoid cartesian products.

13.2.3 Transaction Tuning

So far we have seen how to improve the performance of SQL queries, but another common cause of inefficiency is the way in which transactions are programmed in the applications. In particular, the transaction should be as short as possible to reduce the length of the locks on the data used, and if it needs to collect information from the user of the application, this should be done before the start of the transaction, thus avoiding active locks during the interactions with the user. The possibility of subdividing long transactions into a sequence of short transactions must be also considered.

The serializability of concurrent transactions is an important requirement of the DBMS, but it reduces the number of concurrent transactions executable per unit of time. There are cases, however, in which this property is not essential. For example, in a flight reservation overbooking is tolerated, or a bank can make decisions on one of its customer even if the balance is not the current one.

For these reasons, the DBMS allows us to declare that a transaction is isolated from other concurrent transactions with a criterion weaker than serializability to prevent, or limit, the use of the locks during the execution of a transaction. The SQL standard requires that, in the absence of other specifications, the execution of concurrent transactions is serializable, but there are commercial systems that do not behave in this way, but take one of the weaker criteria, which we will see later, in the absence of other specifications. Therefore, it is necessary to know the default isolation level of the DBMS in use before writing an application so as to avoid surprises [Adya et al., 2000].

Atomicity and durability are instead essential requirements when a transaction modifies data. These kinds of operations are, however, less frequent than the read ones and for this reason the additional cost of the writing to the log file is accepted.

The SQL standard provides that write operations always take locks on data that are released at the end of the transaction, but read locks are treated differently depending on the isolation level specified at the beginning of the transaction:

– **READ UNCOMMITTED**. The reads of records are performed without the use of locks on data and so *dirty reads* are possible since the transaction can read a write locked data.[6]

  Figure 13.5 shows the history of the concurrent execution of two transactions: T_1 reads the database values of x and y, and then it modifies them, T_2 reads and prints the values of x and y, and it is executed with the isolation level **READ UNCOMMITTED**. The values printed by T_2 are not correct both in the case shown, and in the case that T_1 instead of the *commit* would have executed an *abort*.

6. To avoid improper writes on the database, some DBMSs require that transactions with the isolation level **READ UNCOMMITTED** are not allowed to modify the database.

T_1	$T_2(RU)$
$r[x = 200]$	
$w[x := 100]$	
	$r[x = 100]$
	$r[y = 500]$
	c
$r[y = 500]$	
$w[y := 600]$	
c	
	dirty read of x

Figure 13.5: The problem of dirty reads

– **READ COMMITTED**. The transaction obtains short-term read locks on *records*, i.e. released after reading, and exclusive locks before writing records and holds the locks until the end. This isolation level prevents the *dirty reads* problem, but not the problem of *non-repeatable reads*, because a read lock is released after reading (in Figure 13.6, T_1 reads x twice during its execution, and it finds different values) or *loses changes* (Figure 13.7).

$T_1(RC)$	T_2
$r[x = 100]$	
	$w[x := 500]$
	c
$r[x = 500]$	
c	
read of x non-repeatable	

Figure 13.6: The problem of non-repeatable reads

$T_1(RC)$	$T_2(RC)$
$r[x = 100]$	
	$r[x = 100]$
	$w[x := 500]$
	c
$w[x := 600]$	
c	
	loss of the change of x

Figure 13.7: The problem of loss of changes

– **REPEATABLE READ**. The read and write locks of *records* are released only at the end of the transaction. This solution avoids the above problems, but introduces the issue of *phantom records*: suppose that (a) transaction T_1 executes a query Q to retrieve records of R which satisfy a condition P, and (b) transaction T_2 insert other records into R which satisfy the condition P, before the termination of T_1. If T_1 re-executes Q see also the records inserted by T_2.

– **SERIALIZABLE**. The read and write locks are of different sizes, and a read lock on a table R is in conflict with an update of R. This isolation level, used as default, does not suffer from the problems previously seen, but it reduces the number of concurrent transactions executable per unit of time.

Note that commercial DBMSs may (a) not provide all the previous levels of isolation, (b) not have the same default level of isolation and (c) have other isolation levels (Table 13.2).[7]

Table 13.2: Isolation levels of some commercial DBMSs
(CS = CURSOR STABILITY, SN = SNAPSHOT)

DBMS	READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE	OTHER
DB2	√	√	default		CS
SQL Server	√	default	√	→	SN
Oracle		default		→	SN
PostgreSQL	→	default		→	SN
MySQL	√	√	√	default	

Some DBMS provide the **SNAPSHOT**, presented in Section 10.7, as isolation level **SERIALIZABLE**.

In addition to the right isolation level, it is important to choose an appropriate locking granularity: on tables for "long" transactions, on relation pages for "medium" transactions, on records for "short" transactions. Finally, changes to the catalog should be avoided when the database is in use, because when the catalog changes, the system will block access to the database for the entire duration of the operation.

13.2.4 Logical Schema Tuning

If the physical design is not able to provide the desired performance, it is necessary to consider a modification of the database logical schema to evaluate two types of restructuring: *partitioning* (vertical or horizontal) and *denomalization*. Then appropriate view definitions are needed for ensuring the logic independence of applications.

In the analysis of restructuring a database logical schema we consider the following example in BCNF about the exams of master degree programs:

Students(StudentNo, Name, City, BirthYear, BachelorDegree, University, Other)
Exams(PkExam, Lecturer, StudentNo, Course, Grade, Date, Other)
MasterProgram(PkMaster, Title, Coordinator, Department, Other)
MasterProgramExams(FkMaster, FkExam)

Vertical Partitioning

The goal is to reduce the volume of data transferred between permanent memory and cache by splitting a relation in order to separate frequently accessed attributes from those that are rarely referenced. A relation is partitioned with *projections that include*

7. When an isolation level is not marked this means that the DBMS does not allow the specification. When an isolation level is marked with "→" means that it can be specified but is treated as the next level.
In DB2 the level **REPEATABLE READ** is equivalent to **SERIALIZABLE**.
Oracle also provides the level **READ ONLY**.

the primary key to guarantee that each partition has a *subset of the original attributes* of *all the original records*.

Suppose that the following query is very frequently asked "*find the number of exams passed and the number of students who have done the test by course, and by academic year*", assuming that an exam is failed if the grade is "F".

A vertical partitioning of Exams that speeds up the query execution is

ExamsForAnalysis(PkExam, Course, Grade, Date)
RestOfExams(PkExam, StudentNo, Other)

The partitioning preserves data, but loses the information that (Course, StudentNo) is a key.

Horizontal Partitioning

The goal is the reduction of the cost of accessing a very large relation by splitting it into partitions, to separate frequently accessed data from data rarely used. With horizontal partitioning, each partition has a *all the original attributes* of a *disjoint subset of the original records*.

Suppose that the following query is very frequently asked "*for a study program with title X and a course with less than 5 exams passed, find the number of exams, by course, and academic year*".

A horizontal partitioning of Exams that speeds up the query execution is partitioning the table into different ones for each master programs, obtained from the join of the relations Exams, MasterProgramExams, MasterProgram.

MasterProgramXExams(PkExam, Course, StudentNo, Grade, Date, Other)

Denormalization

Normalization is the process of decomposing a table into smaller tables to reduce redundancy and avoid anomalies. Normalization can decrease performance. Denormalization is the process of adding columns to some tables to improve the query performance of read-only queries. It reverses the normalization process and results in a violation of normal form conditions.

Suppose that the following query is very frequently asked "*for a student number N, find the student name, the master program title and the grade of exams passed*".

To speed up the query the definition of the following table is useful, not in normal form, by adding attributes to Exams.

ExamsStudentsMaster(PkExam, Course, StudentNo, StudentName, Grade, Date,
MasterTitle, Other)

13.3 DBMS Tuning

If all the strategies outlined so far do not lead to the expected applications performance, the last thing to do is try to tune the database system. The aspects to consider are numerous, but in general the most important ones are at three levels, which interact with one another, and therefore they must be considered together.

1. **Transaction Management**. It is possible to intervene on parameters such as the log storage, the frequency of checkpoints and dumps. In systems that allow the size of the log to be set, the choice of this parameter can be complicated. In addition, the more frequent the checkpoints and dumps are, the faster the database can be

put online after a system failure or a disaster. The other side of the coin is that these operations subtract resources to normal activities and, therefore, if performed too frequently, can cause a significant degradation of system performance.

2. **Database Buffer Size**. The performance of the applications improves with a larger buffer size. However, when setting this value, it is also necessary to consider the amount of available memory, to avoid the buffer being placed on the disk, due to lack of space, with noticeable performance degradation.
3. **Disk Management**. Options for tuning systems at this level including adding disks or using RAID system if disk I/O is a bottleneck, adding more memory if the disk buffer is a bottleneck, or moving to a faster processor if CPU use is a bottleneck.
4. **Parallel Architectures**. This is is a complex issue that depends strictly on the system in use. The use of a parallel architecture, among other things, has impact on the way in which the data is stored and how the queries are optimized.

13.4 Summary

1. The physical design of a database requires the choice of storage structures for tables and indexes to ensure the performance expected by the workload of reference. The problem is very complex and depends on the technical features of the DBMS used, which, however, usually provide tools to assist the designer in this task.
2. In general, after a certain period of use of the database, the users report performance problems for certain applications that require a revision of the physical design, in particular the indexes used, taking into account the characteristics of the DBMS optimizer.
3. In the database tuning phase it may be necessary to review both the way in which queries were written, and the way in which the database logical schema was designed.

Bibliographic Notes

A very useful book dedicated to databases tuning is [Shasha and Bonnet, 2002]. For information on a specific DBMS, refer to the manual available online.

Exercises

Exercise 13.1 Consider the following relational schema, where the keys are underlined:

Suppliers(PkSupplier, Name, City)
Parts(PkPart, Name, PColor, Weight)
Orders(FkSupplier, FkPart, Date, Qty)

and the query:

```
SELECT      DISTINCT S.Name, SUM(Qty)
FROM        Orders O, Suppliers S
WHERE       O.FkSupplier = S.PkSupplier AND O.FkPart < 'P5' AND S.City = 'Pisa'
GROUP BY    S.Name
ORDER BY    S.Name;
```

1. Give the query logical tree.

2. Find the best physical plan to evaluate the query without indexes and estimate its cost.
3. Suggest the indexes that can be added to this database in order to improve the query performance. Indicate whether these indices should be clustered or not. Explain your reasoning briefly.
4. Find the best physical plan to evaluate the query and estimate its cost.

Exercise 13.2 Consider the following relation:

Employee(EmployeeId, Name, Age, DeptId)

For each of the following queries, estimate the cost of a physical plan for two cases: for the given query and for the query rewritten so that a better physical plan can be found.

a) A B^+–tree index on Age is available:

```
SELECT  DeptId
FROM    Employee
WHERE   Age = 30  OR  Age = 35 ;
```

b) A B^+–tree index on Age is available:

```
SELECT  DeptId
FROM    Employee
WHERE   Age < 35  AND  Age > 30 ;
```

c) A B^+–tree index on Age is available:

```
SELECT  DeptId
FROM    Employee
WHERE   2 * Age = 100;
```

d) No index is available:

```
SELECT      DeptId, AVG(Age)
FROM        Employee
GROUP BY    DeptId
HAVING      DeptId = 20;
```

e) Consider also the following relation, where the key is underlined:

Departments(DeptId, DeptName, Location, Phone, ManagerId)

ManagerId can have a null value and it is a foreign key for Employees.

```
SELECT  EmployeeId
FROM    Employees, Departments
WHERE   EmployeeId = ManagerId;
```

Exercise 13.3 Consider the schema, R(K, A), S(KE, B), with KE a foreign key for R, and the query:

```
SELECT  *
FROM    R, S
WHERE   K = KE;
```

Estimate the cost of two physical query plans with the operator **MergeJoin**, one without the use of indexes, and another one using indexes on K and on KE. For each of the following cases, explain why the indexes are useless:

1. R and S have a record for page,
2. R and S have a few records for page.

Exercise 13.4 Consider the schema

Employees(EmpID, Name, Salary, Age, DeptID)
Departments(DeptID, Budget, University, Manager)

Assume that the following attribute values are uniformly distributed: Salary in the range 10 to 100, Age in the range 20 to 70, Budget in the range 100 to 1000. Each department has on average 30 employees and there are 30 universities.

Describe the indexes that you would choose to improve the query performance in the following cases. Give the cost of the physical query plans. If the system does not support index-only plans, how do you change the solution?

a) Find the name, age and Salary of all employees.
   a1) No index.
   a2) A clustered index on (Name, Age, Salary).
   a3) An index on (Name, Age, Salary).
   a4) An index on (Name, Age).

b) Find the DeptID of the departments of the University of Pisa with a budget less than 50.
   b1) No index.
   b2) A clustered index on (University, Budget).
   b3) An index on (University, Budget).
   b4) A clustered index on (Budget).

c) Find the names of the employees who manage departments and have a salary greater than 50.
   c1) No index.
   c2) An index on DeptID.
   c3) An index on EmpID.
   c4) A clustered index on Salary.

Appendix A

FORMULARY

Selectivity Factor of Conditions

Condition (ψ)	Selectivity Factor
$A_i = c$	$s_f(\psi) = \begin{cases} 1/N_{\text{key}}(A_i) & \text{if } A_i \text{ is indexed attribute} \\ 1/10 & \text{otherwise} \end{cases}$
$A_i = A_j$	$s_f(\psi) = \begin{cases} \frac{1}{\max(N_{\text{key}}(A_i), N_{\text{key}}(A_j))} & \text{if both } A_i \text{ and } A_j \text{ are indexed attributes and } \mathrm{dom}(A_i) \subseteq \mathrm{dom}(A_j) \text{ or } \mathrm{dom}(A_j) \subseteq \mathrm{dom}(A_i). \\ 0 & \text{if } \mathrm{dom}(A_i) \textit{ and } \mathrm{dom}(A_j) \text{ are disjoint.} \\ 1/N_{key}(A_i) & \text{if only } A_i \text{ is an indexed attribute} \\ 1/10 & \text{otherwise} \end{cases}$
$A_i > c$	$s_f(\psi) = \begin{cases} \frac{\max(A_i) - c}{\max(A_i) - \min(A_i)} & \text{if } A_i \text{ is numerical and an indexed attribute} \\ 1/3 & \text{otherwise} \end{cases}$
$A_i < c$	$s_f(\psi) = \begin{cases} \frac{c - \min(A_i)}{\max(A_i) - \min(A_i)} & \text{if } A_i \text{ is numerical and an indexed attribute} \\ 1/3 & \text{otherwise} \end{cases}$
$A_i < A_j$	$s_f(\psi) = 1/3$
A_i BETWEEN c_1 AND c_2	$s_f(\psi) = \begin{cases} \frac{c_2 - c_1}{\max(A_i) - \min(A_i)} & \text{if } A_i \text{ is numerical and an indexed attribute} \\ 1/4 & \text{otherwise} \end{cases}$
A_i IN ($v_1, \ldots, v_n$)	$s_f(\psi) = \begin{cases} n \times s_f(A_i = v) & \text{if less than 1/2} \\ 1/2 & \text{otherwise} \end{cases}$
NOT ψ_1	$s_f(\psi) = \{1 - s_f(\psi_1)$
ψ_1 AND ψ_2	$s_f(\psi) = \{s_f(\psi_1) \times s_f(\psi_2)$
ψ_1 OR ψ_2	$s_f(\psi) = \{s_f(\psi_1) + s_f(\psi_2) - s_f(\psi_1) \times s_f(\psi_2)$

JRS Physical Operators

Operators marked with an asterisk are not supported by the current JRS version.

Logical Operator	Physical Operators	Description
Table R	**TableScan** (R)	Full table scan of R.
	IndexScan (R, I)	Sorted scan of R on the index I attributes.
	IndexSequentialScan* (R, I)	Sorted scan of R on the primary key with R organized *index sequential*.
	SortScan $(R, \{A_i\})$	Sorted scan of R on the attributes $\{A_i\}$.
Projection π^b	**Project** $(O, \{A_i\})$	Projection of O records without duplicate elimination.
	IndexOnlyScan $(R, I, \{A_i\})$	Projection of R records on attributes available in the index I without duplicate elimination, and without any access to R.
Duplicate elmination δ	**Distinct** (O)	Duplicate elimination from *sorted* O records.
	HashDistinct* (O)	Duplicate elimination from O records.
Sort τ	**Sort** $(O, \{A_i\})$	Sort O records on $\{A_i\}$.
Selection σ	**Filter** (O, ψ)	Selection of the O records that satisfy the condition ψ.
	IndexFilter (R, I, ψ)	Selection of the R records using the index I defined on the attributes in ψ.
	IndexSequentialFilter* (R, I, ψ)	Selection of the R records that satisfy the condition on the primary key with R organized *index sequential*.
	IndexOnlyFilter $(R, I, \{A_i\}, \psi)$	Selection of the R records attributes available in the index I and used in ψ, without any access to R. The attributes $\{A_i\}$ are a subset of those in I.
	OrIndexFilter* $(R, \{I_i, \psi_i\})$	Selection of the R records with indexes on predicates of a disjunctive condition.
	AndIndexFilter* $(R, \{I_i, \psi_i\})$	Selection of the R records with indexes on predicates of a conjunctive condition.

Logical Operator	Physical Operators	Description
Grouping γ	**GroupBy** $(O, \{A_i\}, \{f_i\})$	Grouping of O records *sorted* on the attributes $\{A_i\}$ using the aggregate functions in $\{f_i\}$. The operator returns records with attributes A_i and the aggregate functions in $\{f_i\}$, sorted on the attributes $\{A_i\}$.
	HashGroupBy $(O, \{A_i\}, \{f_i\})$	Grouping of O records using a *hash function* on the attributes $\{A_i\}$ and the *result is not sorted* on the grouping attributes $\{A_i\}$.
Join $\bowtie$	**NestedLoop** (O_E, O_I, ψ_J)	*Nested-loop* join.
	PageNestedLoop (O_E, O_I, ψ_J)	*Page nested-loop* join.
	IndexNestedLoop (O_E, O_I, ψ_J)	*Index nested-loop* join. O_I uses an index on the join attributes.
	MergeJoin (O_E, O_I, ψ_J)	*Merge join* join. The operand O_E and O_I records are sorted on join attributes. The external operand join attribute is a key.
	HashJoin* (O_E, O_I, ψ_J)	*Hash join*.
Set Operators $\cup, -, \cap$	**Union, Except, Intersect** (O_E, O_I)	Set operations with the operand records sorted and without duplicates.
	UnionAll(O_E, O_I)	Union without duplicates elimination.

Estimates for Physical Operators Cost and Result Size

Physical Operators for Relation

Operator	Cost	Result Size
TableScan(R)	$N_{\text{pag}}(R)$	$E_{\text{rec}} = N_{\text{rec}}(R)$
IndexScan(R, I)		
clustered index	$N_{\text{leaf}}(I) + N_{\text{pag}}(R)$	
index on a key of R	$N_{\text{leaf}}(I) + N_{\text{rec}}(R)$	
otherwise	$N_{\text{leaf}}(I) + N_{\text{key}}(I) \times \Phi(\lceil N_{\text{rec}}(R)/N_{\text{key}}(I) \rceil, N_{\text{pag}}(R))$	where $\Phi(k : int, n : int) : int = \lceil n(1-(1-1/n)^k) \rceil$
IndexSequentialScan(R, I)	$N_{\text{leaf}}(I)$	
SortScan$(R, \{A_i\})$	$3 \times N_{\text{pag}}(R)$	

The **SortScan** operator returns the records of a relation R sorted in ascending order using a *piped merge sort* algorithm that returns the final result without first writing it to a temporary file. For simplicity, we assume that it requires a single merge phase.

Physical Operators for Projection

Operator	Cost	Result Size
Project$(O, \{A_i\})$	$C(O)$	$E_{\text{rec}} = E_{\text{rec}}(O)$
IndexOnlyScan$(R, I, \{A_i\})$	$N_{\text{leaf}}(I)$	$E_{\text{rec}} = N_{\text{rec}}(R)$

Physical Operators for Duplicate Elimination

Operator	Cost	Result Size
Distinct(O)	$C(O)$	$E_{\text{rec}} = \lceil E_{\text{rec}}(O) \times \prod s_f(A_i) \rceil$
HashDistinct(O)	$C(O) + 2 \times N_{\text{pag}}(O)$	*where* $\{A_i\}$ *is the set of attributes of* O.
		If A_i *is an indexed attribute in* R $s_f(A_i) = N_{\text{key}}(A_i)/N_{\text{rec}}(R)$
		otherwise $s_f(A_i) = 1/10$

Physical Operator for Sort

Operator	Cost	Result Size
Sort$(O, \{A_i\})$	$C(O) + 2 \times N_{\text{pag}}(O)$	$E_{\text{rec}} = E_{\text{rec}}(O)$

As with the **SortScan**, we assume that the operator **Sort** sorts the records of the operand O with the *piped merge sort* algorithm and a single merge phase.

Physical Operators for Selection

Operator	Cost	Result Size
Filter(O, ψ)	$C(O)$	$E_{\text{rec}} = \lceil s_f(\psi) \times E_{\text{rec}}(O) \rceil$
IndexFilter(R, I, ψ)	$C_I + C_D$ $C_I = \lceil s_f(\psi) \times N_{\text{leaf}}(I) \rceil$ If the index is *clustered* $C_D = \lceil s_f(\psi) \times N_{\text{pag}}(R) \rceil$ otherwise $C_D = \lceil s_f(\psi) \times N_{\text{key}}(I) \rceil \times$ $\Phi(\lceil N_{\text{rec}}(R)/N_{\text{key}}(I) \rceil,$ $N_{\text{pag}}(R))$	$E_{\text{rec}} = \lceil s_f(\psi) \times N_{\text{rec}}(R) \rceil$ where $\Phi(k : int, n : int) : int =$ $\lceil n(1 - (1 - 1/n)^k) \rceil$
IndexSequentialFilter(R, I, ψ)	$\lceil s_f(\psi) \times N_{\text{leaf}}(I) \rceil$	
IndexOnlyFilter$(R, I, \{A_i\}, \psi)$	$\lceil s_f(\psi) \times N_{\text{leaf}}(I) \rceil$	
OrIndexFilter$(R, \{I_i, \psi_i\})$	$\lceil \sum_{k=1}^{n} C_I^k \rceil +$ $\Phi(\lceil s_f(\psi) \times N_{\text{rec}}(R) \rceil,$ $N_{\text{pag}}(R))$	
AndIndexFilter$(R, \{I_i, \psi_i\})$	$\lceil \sum_{k=1}^{n} C_I^k \rceil +$ $\Phi(\lceil s_f(\psi) \times N_{\text{rec}}(R) \rceil,$ $N_{\text{pag}}(R))$	

Physical Operators for Grouping

Operator	Cost	Result Size
GroupBy$(O, \{A_i\}, \{f_i\})$	$C(O)$	$E_{\text{rec}} = \lceil E_{\text{rec}}(O) \times \prod s_f(A_i) \rceil$
HashGroupBy$(O, \{A_i\}, \{f_i\})$	$C(O) + 2 \times N_{\text{pag}}(O)$	*If A_i is an indexed attribute in R* $s_f(A_i) = N_{\text{key}}(A_i)/N_{\text{rec}}(R)$ *otherwise* $s_f(A_i) = 1/10$

Physical Operators for Join

Operator	Cost	Result Size
NestedLoop(O_E, O_I, ψ_J)	$C(O_E) + E_{\text{rec}}(O_E) \times C(O_I)$	$E_{\text{rec}} = \lceil s_f(\psi_J) \times$ $E_{\text{rec}}(O_E) \times$ $E_{\text{rec}}(O_I) \rceil$
PageNestedLoop(O_E, O_I, ψ_J)	$C(O_E) + N_{\text{pag}}(O_E) \times C(O_I)$	
MergeJoin(O_E, O_I, ψ_J)	$C(O_E) + C(O_I)$	
HashJoin(O_E, O_I, ψ_J)	$C(O_E) + C(O_I) +$ $2 \times (N_{\text{pag}}(O_E) + N_{\text{pag}}(O_I))$	
IndexNestedLoop(O_E, O_I, ψ_J)	$C(O_E) + E_{\text{rec}}(O_E) \times C_A^I(O_I)$	*If O_I is an* **IndexFilter**(S, I, ψ_J) $E_{\text{rec}} = \lceil s_f(\psi_J) \times$ $E_{\text{rec}}(O_E) \times$ $N_{\text{rec}}(S) \rceil$ *while if O_I is a* **Filter**(**IndexFilter**$(S, I, \psi_J), \psi)$ $E_{\text{rec}} = \lceil s_f(\psi_J) \times$ $E_{\text{rec}}(O_E) \times$ $(s_f(\psi) \times$ $N_{\text{rec}}(S)) \rceil$

Physical Operators for Set Operations

Operator	Cost	Result Size
Union(O_E, O_I)	$C(O_E) + C(O_I)$	$E_{\text{rec}} = \max(E_{\text{rec}}(O_E), E_{\text{rec}}(O_I)) + \min(E_{\text{rec}}(O_E), E_{\text{rec}}(O_I))/2$
Except(O_E, O_I)	$C(O_E) + C(O_I)$	$E_{\text{rec}} = E_{\text{rec}}(O_E) - E_{\text{rec}}(O_I)/2$
Intersect(O_E, O_I)	$C(O_E) + C(O_I)$	$E_{\text{rec}} = \min(E_{\text{rec}}(O_E), E_{\text{rec}}(O_I))/2$
UnionAll(O_E, O_I)	$C(O_E) + C(O_I)$	$E_{\text{rec}} = E_{\text{rec}}(O_E) + E_{\text{rec}}(O_I)$

BIBLIOGRAPHY

Adya, J., Liskov, B., and O'Neil, P. (2000). Generalized Isolation Level Definitions. In *Proceedings of the IEEE International Conference on Data Engineering*, San Diego, California. 209

Agrawal, R. and Witt, D. D. (1985). Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564. 98

Albano, A. (1992). *Basi di dati: strutture e algoritmi*. Addison-Wesley, Milano. 62

Albano, A., Ghelli, G., and Orsini, R. (2005). *Fondamenti di basi di dati*. Zanichelli, Bologna. 2

Antoshenkov, G. and Ziauddin, M. (1996). Query processing and optimization in Oracle Rdb. *The VLDB Journal*, 5(2):229–237. 191

Astrahan, M., Kim, M., and Schkolnick, M. (1980). Evaluation of the System R access path selection mechanism. In *Proceedings of the IFIP Congress*, pages 487–491, Tokyo, Melbourne. 191

Bayer, R. and Creight, E. M. (1972). Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189. 50

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, R. (1990). The r$^*-$tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ. 74

Bernstein, P., Goodman, N., and Hadzilacos, V. (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, Menlo Park, California. 102, 106

Cahill, M. J., Rohm, U., and Fekete, A. (2008). Serializable Isolation for Snapshot Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 729–738. 118

Cardenas, A. (1975). Analysis and performance of inverted database structures. *Communications of the ACM*, 18(5):253–263. 56

Ceri, S. and Widom, J. (1991). Deriving production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–589, Barcelona, Spain. 162

Cesarini, F. and Soda, G. (1991). A dynamic hash method with signature. *ACM Transactions on Database Systems*, 16(2):309–337. 36

Chaudhuri, S. and Narasayya, V. (1997). An efficient, cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 146–155, Athens, Greece. 62

Chou, H. et al. (1985). Design and implementation of the Wisconsin storage system. *Software – Practice and Experience*, 15(1). 82

Chou, H. and Witt, D. (1985). An evaluation of buffer management strategies for relational database systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 127–141, Stockolm. 12

Chu, J. and Knott, G. (1989). An analysis of B-trees and their variants. *Information Systems*, 14(5):359–370. 50

Comer, D. (1979). The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137. 50

Enbody, R. and Du, H. (1988). Dynamic hashing schemes. *ACM Computing Surveys*, 20(2):85–113. 36

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2005). Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, 30(2):492–528. 118

Ferragina, P. and Grossi, R. (1995). A fully-dynamic data structure for external substring search. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 693–702, Las Vegas. 50

Ferragina, P. and Grossi, R. (1996). Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, Atlanta. 50

Ferragina, P. and Grossi, R. (1999). The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46:236–280. 50

Finkelstein, C., Schkolnick, M., and Tiberio, P. (1988). Physical database design for relational databases. *ACM Transactions on Database Systems*, 1(2):91–138. 62

Frazer, W. and Wong, C. (1972). Sorting by natural selection. *Communications of the ACM*, 15(10):910–913. 22

Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231. 74

Ganski, R. and Wong, H. (1987). Optimization of nested SQL queries revised. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–33, San Francisco, California. 165

Garcia-Molina, H., Ullman, J. D., and Widom, J. (1999). *Database System Implementation*. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 5, 9, 12, 152, 191

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170. 152, 191

Gray, J. and Reuter, A. (1997). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, third edition. 5, 12, 82, 102

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA. 74

Hawthorn, P. and Stonebraker, M. (1979). Performance analysis of a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, MA. 191

Kabra, N. and DeWitt, D. (1999). OPT++: An object-oriented implementation for database query optimization. *The VLDB Journal*, 8(1):55–78. 191

Kifer, M., Bernstein, A., and Lewis, P. (2005). *Database Systems: An Application Oriented Approach*. McGraw-Hill, Reading, Massachusetts, second edition. 5

Kim, W. (1982). On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469. 165

Knott, G. (1975). Hashing functions. *The Computer Journal*, 8:265–278. 36

Knuth, D. (1973). *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts. 22, 24, 61

Kossmann, D. and Stocker, K. (2000). Iterative dynamic programming: A new class of query optimization algorithms. *ACM Transactions on Database Systems*, 1(25):43–82. 175, 176, 191

Kumar, A. (1994). G-tree: A new data structure for organizing multidimensional data. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):341–347. 66, 74

Lanzelotte, R. S. G. and Valduriez, P. (1991). Extending the search strategy in a query optimizer. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 363–373, Barcelona, Spain. 175, 191

Larson, P. (1982). A single file version of linear hashing with partial expansion. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 300–309, Mexico City. 36

Litwin, W. (1978). Virtual hashing: A dynamically changing hashing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 517–523, Berlin. 30, 36

Litwin, W. (1980). Linear hashing: A new tool for file and table addressing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 212–223, Montreal, Canada. 34, 36

Lorie, R. (1977). Physical integrity in a large segmented databases. *ACM Transactions on Database Systems*, 2(1):91–104. 94

Martin, G. (1979). Spiral storage: Incrementally augmentable hash addressed storage. Technical Report 27, University of Warwick, Coventry, UK. 35

Mohan, C. (1999). Repeating history beyond ARIES. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland. 102

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162. 102

Mullin, J. (1985). Spiral storage: Efficient dynamic hashing with constant performance. *The Computer Journal*, 28(3):330–334. 36

Nievergelt, J., Hinterberger, H., and Sevcik, K. (1984). The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71. 74

O'Neil, P. and O'Neil, E. (2000). *Data Base. Principles, Programming, and Performance*. Morgan Kaufmann Publishers, San Mateo, California, second edition. 5

Paulley, G. N. and Larson, P.-Å. (1994). Exploiting uniqueness in query optimization. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 68–79, Houston, Texas, USA. 162

Pirahesh, H. and Hellerstein, J. (1992). Extensible/rule based query rewrite optimization in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 39–48, San Diego, California. 191

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, New York, third edition. 5, 9, 12, 82, 102, 152, 191

Ramamohanarao, H. (1984). Recursive linear hashing. *ACM Transactions on Database Systems*, 9(3):369–391. 36

Ramsak, F., Markl, V., Fenk, R., Zinkel, M., Elhardt, K., and Bayer, R. (2000). Integrating the UB-tree into a database system kernel. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 263–272, Cairo, Egypt. 74

Rosenberg, A. and Snyder, L. (1981). Time- and space-optimality in B-trees. *ACM Transactions on Database Systems*, 6(1):174–183. 50

Sacco, G. and Schkolnick, M. (1986). Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498. 12

Samet, H. (1990). *The design and analysis of spatial data structures*. Addison-Wesley, Reading, Massachusetts. 74

Scholl, M. (1981). New file organizations based on dynamic hashing. *ACM Transactions on Database Systems*, 6(3):194–211. 36

Selinger, P., Ashtrahan, M., Chamberlin, D., Lorie, R. A., and Price, T. (1979). Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, MA. 191

Severance, D. and Duhne, R. (1976). A practitioner's guide to addressing algorithms. *Communications of the ACM*, 19(6):314–326. 30

Shasha, D. and Bonnet, P. (2002). *Database Tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers, San Mateo, California. 206, 213

Silberschatz, A., Korth, H., and Sudarshan, S. (2010). *Database System Concepts*. McGraw-Hill, New York, sixth edition. 5, 12, 102

Steinbrunn, M., Moerkotte, G., and Kemper, A. (1997). Optimizing join orders. *The VLDB Journal*, 6(2):191–208. 175, 176

Tharp, A. (1988). *File Organization and Processing*. J. Wiley & Sons, New York. 24

Wong, E. and Youssefi, K. (1976). Decomposition - a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241. 191

Yu, C. T. and Meng, W. (1998). *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, California. 74

SUBJECT INDEX

R^*–tree, 70
B^+–tree, 46
B–tree, 40
G–tree, 66

A

Access method
  database operators, 78
  examples, 80
  Heap file operators, 78
  index operators, 79
  operators, 79
Access Methods Manager, 4, 79
ARIES algorithm, 100

B

Buffer Manager, 10
Buffer pool, 10
Buffer Manager, 3

C

Checkpoint, 91
  buffer-consistent, 92
  commit-consistent, 91
  fuzzy checkpoint, 101
Concurrency Manager, 4
Consistent state, 83

D

DBMS architecture, 2
Deadlock, 115
  detection, 115
  prevention, 115
Dynamic Hashing, 30

E

Extendible Hashing, 32
External sorting, 21

F

Failure, 87

H

Heap file page management, 17
Heap organization, 19
Histograms, 131

I

Index, 28, 53
  Bitmap, 58
  clustered, 49, 125
  composite, 61
  inverted, 54
  multi-attribute, 61
  unclustered, 125
  unclustered , 48
  with variable length keys, 50
Index selection, 199
Index sequential organization, 46
Isolation level
  READ COMMITTED, 212
  READ UNCOMMITTED, 211
  REPEATABLE READ, 212
  SERIALIZABLE, 213
  SNAPSHOT, 117

J

JRS, 4

H

Linear Hashing, 34
Locking
  multiple granularity, 118
  Strict 2PL, 113
Log, 88
Logical plan, 156
LSN, 90

M

Magnetic disk
  access time, 9
  block, 8
  rotational latency, 9
  sector, 8
  seek time, 9
  tracks, 8
  transfer time, 9
Malfunctioning
  media failures (disasters), 88
  protection, 89
  system failure, 88
Merge-sort, 21
Multidimensional data, 63

P

Permanent Memory Manager, 3, 9
Physical database design, 197
Physical operator
  for set operations, 151
  for duplicate elimination, 135
  for grouping, 141
  for join, 142
  for projection, 135
  for relation, 134
  for selection, 137
  for sorting, 137
Physical plan, *see* Physical query plan, 156
Physical query plan, 126
  execution, 126

Q

Query Manager, 4
Query optimization
  duplicate elimination, 181
  grouping, 182
  multiple-relations, 175
  pre-grouping, 186
  set operations, 190
  single-relation, 174
Query processing phases, 155
  Physical plan generation, 171
  Query analysis, 156
  Query transformation, 156
Query rewriting
  DISTINCT elimination, 159
  Equivalence rules, 157
  GROUP BY elimination, 163
  VIEW merging, 168
  WHERE-subquery elimination, 165
Query transformation, *see* Query rewriting

R

Relational Engine, 3
Relational operators implementation, 125

S

Sequential organization, 19
Serialization graph, 111
Spiral Hashing, 35
Static Hashing, 28
Storage Engine, 2
Storage Structures Manager, 4, 15

T

Table organization
  dynamic, 27
  primary, 27
  secondary, 27
  static, 27
Transaction, 84
  ACID, 84
  ARIES algorithm, 100
  histories c-equivalent, 109
  history c-serializable, 111
  NoUndo-NoRedo algorithm, 94
  NoUndo-Redo algorithm, 97
  recovery algorithms, 92
  recovery from malfunctioning, 98
  recovery operations, 96
  serial execution, 106
  serializability, 85
  serializable execution, 106
  states, 87
  Undo-NoRedo algorithm, 96
  Undo-Redo algorithm, 97
Transaction and Recovery Manager, 4, 89
Transaction recovery algorithms, 92
Tuning
  database, 208
  DBMS, 214
  index, 209
  logical schema, 213
  queries, 210
  transactions, 211

V

Virtual Hashing, 30