International Journal of Research (IJR) Vol-1, Issue-10 November 2014 **ISSN 2348-6848**

Authenticated Data Structures for Graph and Geometric Searching

Amit Saini(16763) & Akansha Marwah(16762)
Batch Year of study(2013-2017)

Department of Electronics and Computer Science Engineering
Dronacharya College of Engineering Khentawas, Farrukh Nagar – 123506
Gurgaon, Haryana
Email: amit.saini0384@gmail.com , makansha1995@gmail.com

Abstract
Following in the spirit of data structure and algorithm correctness checking, authenticated data structures provide cryptographic proofs that their answers are as accurate as the author intended, even if the data structure is being maintained by a remote host. We present techniques for authenticating data structures that represent graphs and collection of geometric objects. We use a model where a data structure maintained by a trusted source is mirrored at distributed directories, with the directories answering queries made by users. When a user queries a directory, it receives a cryptographic proof in addition to the answer, where the proof contains statements signed by the source. The user veri_es the proof trusting only the statements signed by the source. We show how to e_ciently authenticate data structures for fundamental problems on networks, such as path and connectivity queries, and on geometric objects, such as intersection and containment queries. Our work has applications to the authentication of network management systems and geographic information systems.

Introduction:-

Overview of the Functionality of a Database Management System:-Many of the previous chapters have shown that efficient strategies for complex data-structuring problems are ssential in the design of fast algorithms for a variety of applications, including combinatorial optimization, information retrieval and Web search, databases and data mining, and geometric applications. The goal of this chapter is to
provide the reader with an overview of the important data structures that are used in the implementation of a modern, general-purpose database management system (DBMS). In earlier chapters of the book the reader has already been exposed to many of the data structures employed in a DBMS context (e.g., B-trees, buffer trees, quad trees, R-trees, interval trees, hashing). Hence, we will focus mainly on their application but also introduce other important data structures to solve some of the fundamental data management problems such as *query processing and optimization*, *efficient representation of data on disk*, as well
as the *transfer of data from main memory to external storage*. However, due to space con-
straints, we cannot cover applications of data structures to solve more advanced

problems such as those related to the management of multi-dimensional data warehouses, spatial and temporal data, multimedia data, or XML.

Before we begin our treatment of how data structures are used in a DBMS, we brie°y review the basic architecture, its components, and their functionality. Unless otherwise noted, our discussion applies to a class of DBMSs that are based on the relational data model. Relational database management systems make up the majority of systems in use today and are o®ered by all major vendors including IBM, Microsoft, Oracle, and Sybase.

Most of the components described here can also be found in DBMSs based on other models such as the object-based model or XML.

Figure 60.1 depicts a conceptual overview of the main components that make up a DBMS.

Rectangles represent system components, the double-sided arrows represent input and out-

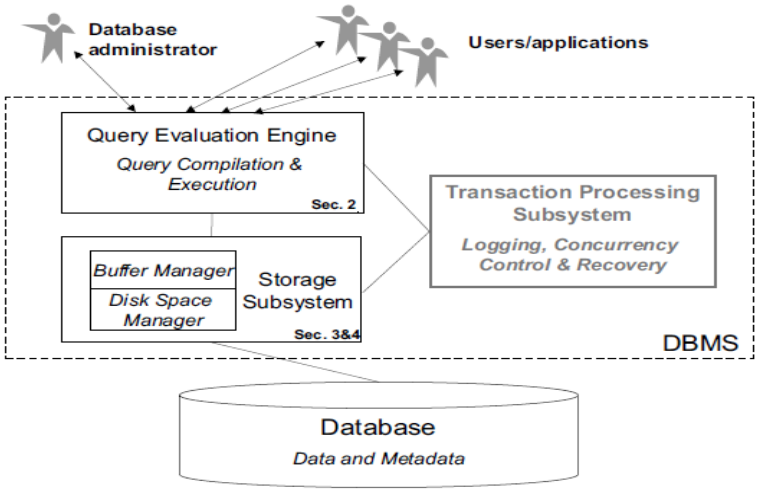

FIGURE 60.1: A simplified architecture of a database management system (DBMS)

put, and the solid connectors indicate data as well as process °between two components.

Please note that the inner workings of a DBMS are quite complex and we are not attempting to provide a detailed discussion of its implementation. For an in-depth treatment the reader should refer to one of the many excellent database books, e.g., [3, 4, 9, 10].

Starting from the top, users interact with the DBMS via commands generated from a variety of user interfaces or application programs. These commands can either retrieve or update the data that is managed by the DBMS or create or update the underlying meta data that describes the schema of the data. The former are called queries, the latter are called data definition statements. Both types of commands are processed by the *Query Evaluation Engine* which contains sub-modules for parsing the input, producing an execution plan, and executing the plan against the underlying database. In the case of queries, the parsed command is presented to a query optimizer sub-module, which uses information about how the data is stored to produce an efficient execution plan from the possibly many alternatives. An execution plan is a set of instructions for evaluating an input command, usually represented as a tree of relational operators. We discuss data

structures that are used to represent parse trees, query evaluation plans, external sorting, and histograms in Section 60.2 when we focus on the query evaluation engine.

Since databases are normally too large to ft into the main memory of a computer, the data of a database resides in secondary memory, generally on one or more magnetic disks.

However, to execute queries or mode captions on data, that data must be transferred to main memory for processing and then back to disk for persistent storage. It is the job of the *Storage Subsystem* to accomplish a sophisticated placement of data on disk, to assure an efficient localization of these persistent data, to enable their bidirectional transfer between disk and main memory, and to allow direct access to these data from higher DBMS architecture levels. It consists of two components: The *Disk Space Manager* is responsible for storing physical data items on disk, managing free regions of the disk space, hiding device properties from higher architecture levels, mapping physical blocks to tracks and sectors of a disc, and controlling the data transfer of physical data items between external and internal memory. The *Buffer Manager* organizes an assigned, limited main memory

Data Structures for Query Processing

Query evaluation is performed in several steps as outlined in Figure 60.2. Starting with the high-level input query expressed in a declarative language called SQL (see, for example, [2]) the *Parser* scans, parses, and validates the query. The goal is to check whether the query is formulated according to the syntax rules of the language supported in the DBMS. The parser also validates that all attribute and relation names are part of the database schema that is being queried.

The parser produces a *parse tree* which serves as input to the *Query Translation and Rewrite* module shown underneath the parser. Here the query is translated into an internal representation, which is based on the relational algebra notation [1]. Besides its compact form, a major advantage of using relational algebra is that there exist transformations (re-write rules) between equivalent expressions to explore alternate, more efficient forms of the same query. Differental gebraic expressions for a query are called *logical query plans* and are represented as *expression trees* or *operator trees*. Using the re-write rules, the initial logical query plan is transformed into an equivalent plan that is expected to execute faster. Query re-writing is guided by a number of heuristics which help reduce the amount of intermediary work that must be performed in order to arrive at the same result.

A particularly challenging problem is the selection of the best join ordering for queries involving the join of three or more relations. The reason is that the *order* in which the input relations are presented to a join operator (or any other binary operator for that matter) tends to have an important impact on the cost of the operation. Unfortunately, the number of candidate plans grows rapidly when the number of input relations grows1.

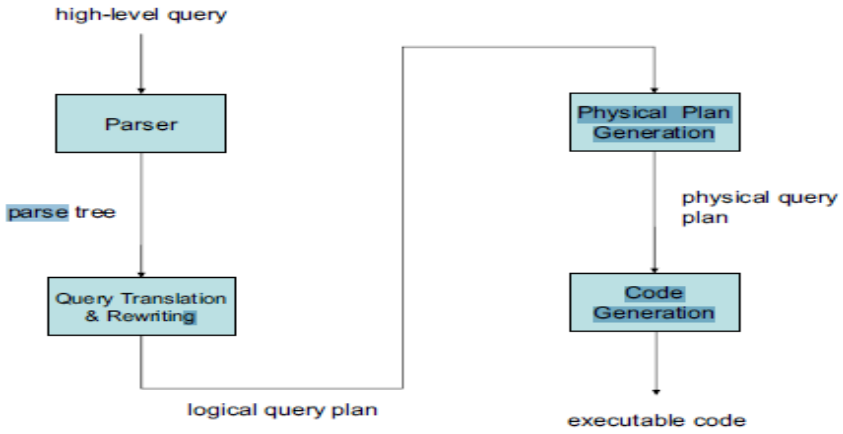

FIGURE 60.2: Outline of query evaluation

The outcome of the query translation and rewrite module is a set of \improved" logical query plans representing different execution orders or combinations of operators of the original query. The *Physical Plan Generator* converts the logical query plans into *physical query plans* which contain information about the algorithms to be used in computing the relational operators represented in the plan. In addition, physical query plans also contain information about the *access methods* available for each relation. Access methods are ways of retrieving tuples from a table and consist of either a file scan (i.e., a complete retrieval of all tuples) or an index plus a matching selection condition. Given the many different options for implementing relational operators and for accessing the data, each logical plan may lead to a large number of possible physical plans. Among the many possible plans, the physical plan generator evaluates the cost for each and chooses the one with the lowest overall cost.

Finally, the best physical plan is submitted to the *Code Generator* which produces the executable code that is either executed directly (interpreted mode) or is stored and executed later whenever needed (compiled mode). Query re-writing and physical plan generation are referred to as *query optimization*. However, the term is misleading since in most cases the chosen plan represents a reasonably efficientstrategy for executing a query.

Query evaluation engines are very complex systems and a detailed description of the underlying techniques and algorithms exceeds the scope of this chapter. More details on this topic can be found in any of the database textbooks (e.g., [3, 4, 9]). For an advanced treatment of this subject, the reader is also referred to [8, 7] as well as to some of the excellent surveys that have been published (see, for example, [6, 5]).

In the following paragraphs, we focus on several important data structures that are used during query evaluation, some of which have been mentioned above: The *parse tree* for storing the parsed and validated input query (Section 60.2.3), the *expression tree* for representing logical and

physical query plans (Section 60.2.4), and the *histogram* which is used to approximate the distribution of attribute values in the input relations (Section 60.2.5).

Index Structures An important part of the work of the physical plan generator is to choose an efficient presentation for each of the operators in the query. For each relational operator (e.g., selection, projection, join) there are several alternative algorithms available for implementation. The best choice usually depends on factors such as size of the relation, available memory in the buffer pool, sort order of the input data, and availability of index structures.

In the following, we briefly highlight some of the important index structures that are used by a modern DBMS and how they can speed up relational operations.

One-dimensional Indexes One-dimensional indexes contain a single search key, which may be composed of multiple attributes. The most frequently used data structures for one-dimensional database indexes are dynamic tree-structured indexes such as *B*/*B*+-*Trees* and *hash-based indexes* using extendible and linear hashing. In general, hash-based indexes are especially good for equality searches. For example, in the case of an equality selection operation, one can use a one dimensional hash-based index structure to examine just the tuples that satisfy the given condition. Consider the selection of students having a certain grade point average (GPA).

Assuming students are randomly distributed throughout the file, an index on the GPA value could lead us to only those records satisfying the selection condition and resulting in a lot fewer data transfers than a sequential scan of the file (if we assume the tuples satisfying the condition make up only a fraction of the entire relation).

Given their superior performance for equality searches hash-based indexes prove to be particularly useful in implementing relational operations such as joins. For example, the index-nested-loop join algorithm generates many equality selection queries, making the difference in cost between a hash-based and the slightly more expensive tree-based implementation significant.

B-Trees provide efficientsupport for range searches (all data items that fall within a range of values) and are almost as good as hash-based indexes for equality searches. Besides their excellent performance, B-Trees are \self-tuning", meaning they maintain as many levels of the index as is appropriate for the size of the file being indexed. Unlike hash-based indexes, B-Trees manage the space on the blocks they use and do not require any overflow blocks.

Indexes are also used to answer certain types of queries without having to access the data file. For example, if we need only a few attribute values from each tuple and there is an index whose search key contains all these fields, we can chose an index scan instead of examining all data tuples. This is faster since index records are smaller (and hence ft into fewer buffer pages). Note that an index scan does not make use of the search structure of the index: for example, in a B-Tree index one would examine all leaf pages in sequence.

All commercial relational database management systems support B-Trees and at least one type of hash-based index structure.

Multi-dimensional Indexes In addition to these one-dimensional index structures, many applications (e.g., geographic database, inventory and sales database for decision-support) also require data structures capable of indexing data existing in two or higher-dimensional spaces. In these domains, important database operations are selections involving partial matches (all

points within a range in each dimension), range queries (all points within a range in each dimension), nearest-neighbor queries (closest point to a given point), and so-called \where-am-I" queries (region(s) containing a point).

Some of the most important data structures that support these types of operations are:

Grid file. A multidimensional extension of one-dimensional hash tables. Grid files support range queries, partial-match queries, and nearest-neighbor queries well, as long as data is uniformly distributed.

Multiple-key index. The index on one attribute leads to indexes on another attribute for each value of the must. Multiple-key indexes are useful for range and nearest-neighbor queries.

R-tree. A B-Tree generalization suitable for collections of regions. R-Trees are used to represent a collection of regions by grouping them into a hierarchy of larger regions. They are well suited to support \where-am-I" queries as well as the other types of queries mentioned above if the atomic regions are individual points.

Quad tree. Recursively divide a multi-dimensional data set into quadrants until each quadrant contains a minimal number of points (e.g., amount of data that can ft on a disk block). Quad trees support partial-match, range, and nearest-neighbor queries well.

Bitmap index. A collection of bit vectors which encode the location of records with a given value in a given field. Bitmap indexes support range, nearest-neighbor, and partial-match queries and are often employed in data warehouses and decision support systems. Since bitmap indexes tend to get large when the underlying attributes have many values, they are often compressed using a run-length encoding.

Given the importance of database support for non-standard applications, most commercial relational database management systems support one or more of these multi-dimensional indexes, either directly as part of the core engine (e.g., bitmap indexes), or as part of an of object-relational extensions (e.g., R-trees in a spatial extender).

60.2.2 Sorting Large Files The need to sort large data files arises frequently in data management. Besides outputting the result of a query in sorted order, sorting is useful for eliminating duplicate data items during the processing of queries. In addition, a widely used algorithm for performing a join operation (sort-merge join) requires a sorting step. Since the size of databases routinely exceeds the amount of available main memory, all DBMS vendors use an external sorting technique called *merge sort* (which is based on the main-memory version with the same name). The idea behind merge sort is that a file which does not ft into main memory can be sorted by breaking it into smaller pieces (sublists), sorting the smaller sublists individually, and then merging them to produce a file that contains the original data items in sorted order.

The external merge sort is also a good example of how main memory versions of algorithms and data structures need to change in a computing environment where all data resides on secondary and perhaps even tertiary storage. We will point out more examples where the most appropriate data structure for data stored on disk is different from the data structures used for algorithms that run in memory in Section 60.4 when we describe the disk space manager.

During the phase, also called the run-generation phase, merge-sort the availablebuffer pages in main memory with blocks containing the data records from the file on disk.

Sorting is done using any of the main-memory algorithms (e.g., Heapsort, Quicksort). The sorted records are written back to new blocks on disk, forming a sorted

sublist containing as many blacks as there are available buffer pages in main memory. This process is repeated until all records in the data file are in one of the sorted sublists. Run-generation is depicted in Figure 60.3.

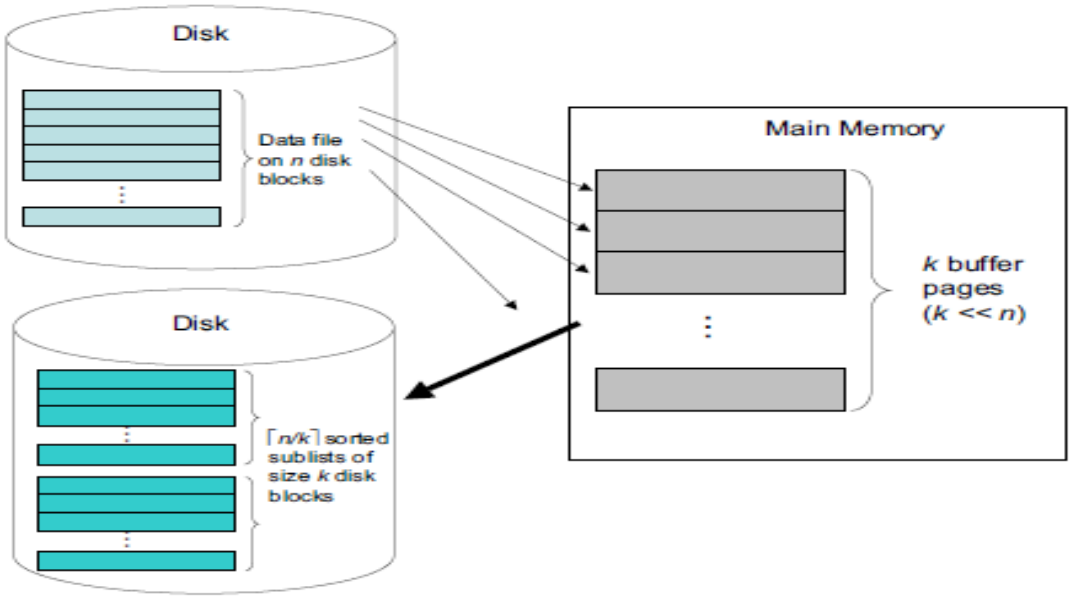

FIGURE 60.3: Run generation phase

Data Structures for Buffer Management:-

A *buffer*is partitioned into an array of *frames* each of which can keep a *page*. Usually a page of a buffer is mapped to a *block*2 of a file so that reading and writing of a page only require one disk access each. Application programs and queries make requests on the buffer manager when they need a block from disk, that contains a data item of interest. If the block is already in the buffer, the buffer manager conveys the address of the block in main memory to the requester. If the block is not in main memory, the buffer manager list allocates space in the buffer for the block, throwing out some other block if necessary, to make space for the new block. The displaced block is written back to disk if it was since the most recent time that it was written to the disk. Then, the buffer manager reads in the requested block from disk into the free frame of the buffer and passes the page address in main memory to the requester. A major goal of buffer management is to minimize the number of block transfers between the disk and the buffer.

Besides pages, so-called *segments* are provided as a counterpart of files in main memory.

This allows one to different segment types with additional attributes, which support varying requirements concerning data processing. A segment is organized as a contiguous subarea of the buffer in a virtual, linear address space with visible page borders. Thus, it consists of an ordered sequence of pages. Data items are managed so that page borders are respected. If a data

item is required, the address of the page in the buffer containing the item is returned.
An important question now is how segments are mapped to files. An appropriate mapping enables the storage system to preserve the merits of the file concept. The distribution of a segment over several files turns out to be unfavorable in the same way as the representation of a data item over several pages. Hence, a segment *Sk*is assigned to exactly one file*Fj*, and *m* segments can be stored in a file. Since block size and page size are the same, page *Pki2 Sk*is assigned to a block *Bjl2 Fj*. We distinguish four methods of realizing this mapping.

File Organization
A *file*(*segment*) can be viewed as a sequence of *blocks* (*pages*). Four fundamentalfile organizations can be distinguished, namelyfiles of unordered records (heap files), files of ordered records (sorted files), files with dispersed records (hash files), and tree-based files (index structures).
Heap files are the simplest file organization. Records are inserted and stored in their un-ordered, chronological sequence. For each heap file we have to manage their assigned pages (blocks) to support scans as well as the pages containing free space to perform insertions Efficiently. Doubly-linked lists of pages or directories of pages using both page numbers for page addressing are possible alternatives. For the first alternative, the DBMS uses a *header page* which is the first page of a heap file, contains the address of the first data page, and information about available free space on the pages. For the second alternative, the DBMS must keep the first page of the heap file in mind. The directory itself represents a collection of pages and can be organized as a linked list. Each directory entry points to a page of the heap file. The free space on each page is recorded by a counter associated with each directory entry. If a record is to be inserted, its length can be compared to the number of free bytes on a page.
Sorted files physically order their records based on the values of one (or several) of their fields, called the *ordering field*(*s*). If the ordering field is also a *key field* of the file, i.e., a field guaranteed to have a unique value in each record, then the field is called the *ordering key* for the file. If all records have the same fixed length, binary search on the ordering key can be employed resulting in faster access to records.
Hash files are a file organization based on hashing and representing an important indexing technique. They provide very fast access to records on certain search conditions. Internal hashing techniques have been discussed in different chapters of this book; here we are dealing with their external variants and will only explain their essential features. The fundamental idea of hash files is the distribution of the records of a file into so-called *buckets*, which are organized as heaps. The distribution is performed depending on the value of the *search key*.
The direct assignment of a record to a bucket is computed by a *hash function*. Each bucket consists of one or several pages of records. A *bucket directory* is used for the managemen of the buckets, which is an array of pointers. The entry for index *i* points to the first page of bucket *i*. All pages for bucket *i* are organized as a linked list. If a record has to be inserted into a bucket, this is usually done on its last page since only there space can be found. Hence, a pointer to the last page of a bucket is used to accelerate the access to this page and to avoid traversing all the pages of the bucket. If there is no space left on the last page, overflow pages are provided. This is called a *static hash file*. Unfortunately, this strategy can cause long chains of overflow pages. *Dynamic hash files* deal with this problem by allowing a variable number of buckets. *Extensible hash files* employ a directory

structure in order to support insertion and deletion efficiently without the employment of overflow pages. *Linear hash files* apply an intelligent strategy to create new buckets. Insertion and deletion are efficiently realized without using a directory structure.

Index structures are a fundamental and predominantly tree-based file organization based on the search key property of values and aiming at speeding up the access to records. They have a paramount importance in query processing. Many examples of index structures are already described in detail in this book, e.g., B-trees and variants, quad-trees and cot trees, RR-trees and variants, and other multidimensional data structures. We will not discuss them further here. Instead, we mention some basic and general organization forms for index structures that can also be combined. An index structure is called a *primary organization* if it contains search key information together with an embedding of the respective records it is named a *secondary organization* if it includes besides search key information only TIDs or TID lists to records in separate file structures (e.g., heap files or sorted files). An index is called a *dense index* if it contains (at least) one index entry for each search key value which is part of a record of the indexed file; it is named a *sparse index* (Figure 60.17) if it only contains an entry for each page of records of the indexed file. An index is called a *clustered index* (Figure 60.17) if the logical order of records is equal or almost equal to their physical order, i.e., records belonging logically together are physically stored on neighbored pages.

Otherwise, the index is named *non-clustered*. An index is called a *one-dimensional index* if a linear order is defined on the set of search key values used for organizing the index entries.

Such an order cannot be imposed on a *multi-dimensional index* where the organization of index entries is based on spatial relationships. An index is called a *single-level index* if the index only consists of a single file; otherwise, if the index is composed of several files, it is named a *multi-level index* .

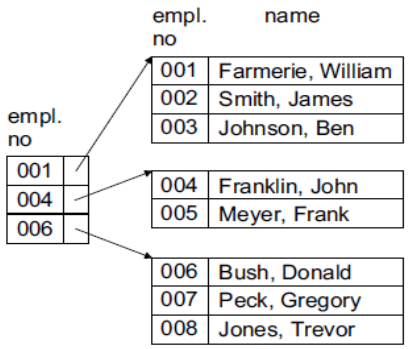

FIGURE 60.17: Example of a clustered, sparse index as a secondary organization on a sorted file

60.5 Conclusion

A modern database management system is a complex software system that leverages many So phisticated algorithms, for example, to evaluate relational operations, to provide efficient access to data, to manage the buffer pool, and to move data between disk and main memory.

In this chapter, we have shown how many of the data structures that were introduced in earlier parts of this book (e.g., B-trees, buffer trees, quad trees, R-trees, interval trees, hashing) including a few new ones such as histograms, LOBs, and disk pages, are being used in a real-world application. However, as we have noted in the introduction, our coverage of the data structures that are part of a DBMS is not meant to be exhaustive since a complete treatment would have easily exceeded the scope of this chapter. Furthermore, as the functionality of a DBMS must continuously grow in order to support new applications (e.g., GIS, federated databases, data mining), so does the set of data structures that must be designed to efficiently manage the underlying data (e.g., s patio-temporal data, XML, bio-medical data). Many of these new data structure challenges are being actively studied in the database research communities today and are likely to form a basis for tomorrow's systems.

References

[1] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377{387, 1970.

[2] Chris J. Date and Hugh Darwen. *A Guide to The SQL Standard*. Addison-Wesley Publishing Company, Inc., third edition, 1997.

[3] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, fourth edition, 2003.

[4] Hector Garcia-Molina, Je®rey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, frst edition,

[5] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2003.

[6] Abraham Silberschatz, Henry F. Korth, and S. Sudharshan. *Database System Concepts*. McGraw-Hill, fourth edition, 2002