60

Data Structures for Databases

Joachim Hammer

University of Florida

Markus Schneider

University of Florida

60.1 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 essential 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 constraints, 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 briefly 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 offered 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-
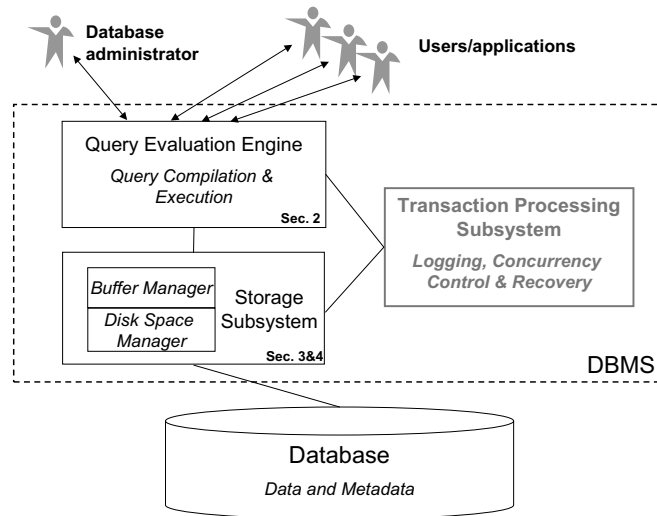
FIGURE 60.1: A simplified architecture of a database management system (DBMS)

put, and the solid connectors indicate data as well as process flow 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 fit 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 modifications on data, that data must first 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

area called *buffer*. A buffer may be a buffer pool and may comprise several smaller buffers. Components at higher architecture levels may have direct access to data items in these buffers.

In Sections 60.3 and 60.4, we discuss data structures that are used to represent both data in memory as well as on disk such as fixed and variable-length records, large binary objects (LOBs), heap, sorted, and clustered files, as well as different types of index structures. Given the fact that a database management system must manage data that is both resident in main memory as well as on disk, one has to deal with the reality that the most appropriate data structure for data stored on disk is different from the data structures used for algorithms that run in main memory. Thus when implementing the storage manager, one has to pay careful attention to selecting not only the appropriate data structures but also to map the data between them efficiently.

In addition to the above two components, today's modern DBMSs include a *Transaction Management Subsystem* to support concurrent execution of queries against the database and recovery from failure. Although transaction processing is an important and complex topic, it is less interesting for our investigation of data structures and is mentioned here only for completeness.

The rest of this chapter is organized as follows. Section 60.2 describes important data structures used during query evaluation. Data structures used for buffer management are described in Section 60.3, and data structures used by the disk space manager are described in Section 60.4. Section 60.5 concludes the chapter.

60.2 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. Different algebraic 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 grows[1].

---

[1]To be exact, for n relations there are $n!$ different join orderings.

high-level query

| Parser | | Physical Plan Generation |

parse tree

physical query plan

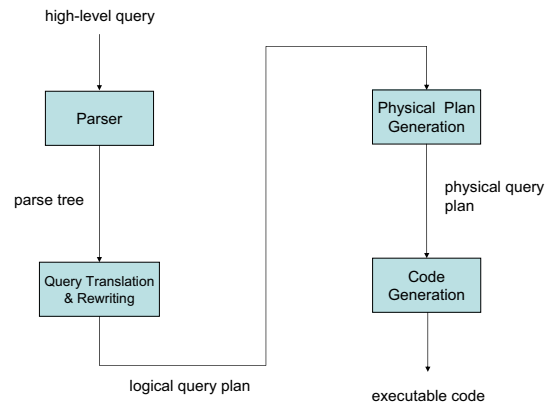| Query Translation & Rewriting | | Code Generation |

logical query plan

executable code

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 efficient strategy 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). We start with a summary of the well-known *index structures* and how they are used *to speed up the basic database operations*. Since sorting plays an important role in query processing, we

include a separate description of the data structures used to *sort large files using external memory* (Section 60.2.2).

60.2.1 Index Structures

An important part of the work of the physical plan generator is to chose an efficient implementation 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 efficient support 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 fit 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 multi-dimensional 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 first. 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 fit 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 fit 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 first phase, also called the run-generation phase, merge-sort fills the available

buffer 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.
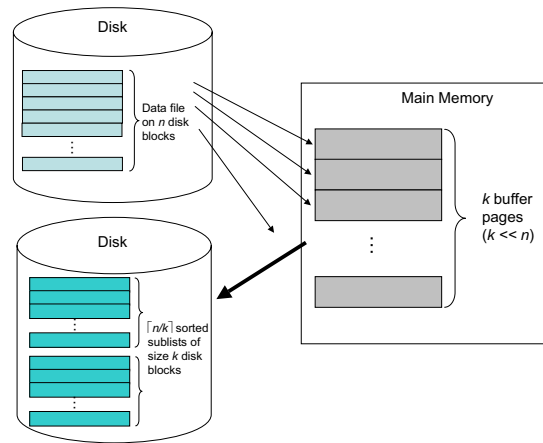

FIGURE 60.3: Run generation phase

In the second phase of the external sort merge algorithm, also called the merging phase, all but one of the main memory buffers are used to hold input data from one of the sorted sublists. In most instances, the number of sorted sublists is less than the number of buffer pages and the merging can be done in one pass. Note, this so-called *multi-way merging* is different from the main-memory version of merge sort which merges pairs of runs; hence it is also called two-way merge. A two-way merge strategy in the external version would result in reading data in and out of memory $2 * \log_2(n)$ times for n sublists (versus reading all n sublists only once). The arrangement of buffers to complete this one-pass multi-way merging step is shown in Figure 60.4.

In the rare situation when the number of sorted sublists exceeds the available buffer pages in main memory, the merging step must be performed in *several passes* as follows: assuming k buffer pages in main memory, each pass involves the repeated merging of $k - 1$ sublists until all sublists have been merged. At this point the number of runs has been reduced by a factor of $k - 1$. If the reduced number of sublists is still greater than k, the merging is repeated until the number of sublists is less than k. A final merge then generates the sorted output. In this scenario, the number of merge passes required is $\lceil \log_{k-1}(n/k) \rceil$.

60.2.3 The Parse Tree

A *parse tree* is an m-ary tree that shows the structure of a query in SQL. Each interior node of the tree is labeled with a non-terminal symbol of SQL's grammar with the goal symbol labeling the root node. The query being parsed appears at the bottom with each token of the query being a leaf in the tree. In the case of SQL, leaf nodes are lexical elements such
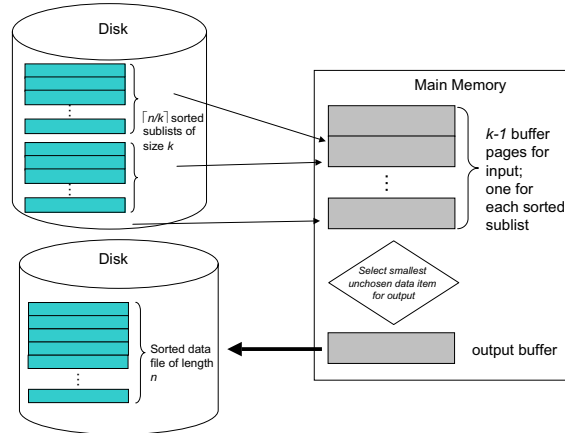
FIGURE 60.4: One pass, multi-way merging phase

as keywords of the language (e.g., $SELECT$), names of attributes or relations, operators, and other schema elements.

The parse tree for the query

```
SELECT Name
FROM Enrollment, Student
WHERE ID = SID AND GPA > 3.5;
```

is shown in Figure 60.5. This query selects all the enrolled students with a GPA higher than 3.5. We are assuming the existence of two relations called Enrollment and Student which store information about enrollment records for students in a school or University.
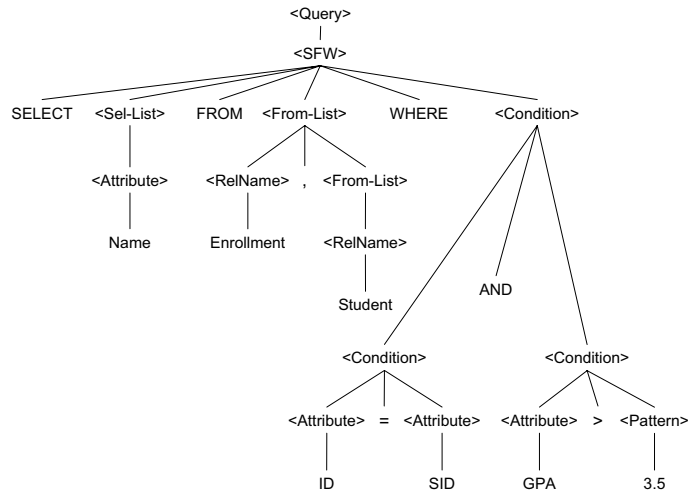
FIGURE 60.5: Sample parse tree for the SQL query

The parse tree shown in Figure 60.5 is based on the grammar for SQL as defined in [4] (which is a subset of the real grammar for SQL). Non-terminal symbols are enclosed in angled brackets. At the root is the category $< Query >$ which forms the goal for the parsed query. Descending down the tree, we see that this query is of the form SFW (select-from-where). Of interest is the $FROM$ list which is defined as a relation name followed by another $FROM$ list. In case one of the relations in the $FROM$ clause is a actually a view, it must be replaced by its own parse tree that describes the view (since a view is essentially a query). A parse tree is said to be valid if it conforms to the syntactic rules of the grammar as well as the semantic rules on the use of the schema names.

60.2.4 Expression Trees

An *expression tree* is a binary tree data structure that corresponds to a relational algebra expression. In the relational algebra we can combine several operators into one expression by applying one operator to the result(s) of either one or two other operators (since all relational operators are either unary or binary). The expression tree represents the input relations of the query as *leaf nodes* of the tree, and the relational algebra operations together with estimates for result sizes as *internal nodes*.

Figure 60.6 shows an example of three equivalent expression trees representing logical query plans for the following SQL query which selects all the students enrolled in the course 'COP 4720' during the term 'Sp04' who have a grade point average of 3.5:

```
SELECT Name FROM Enrollment, Student
WHERE Enrollment.ID = Student.SID AND Enrollment.Course = 'COP 4720'
      AND Enrollment.TermCode = 'Sp04' AND Student.GPA = 3.5;
```

An execution of a tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.
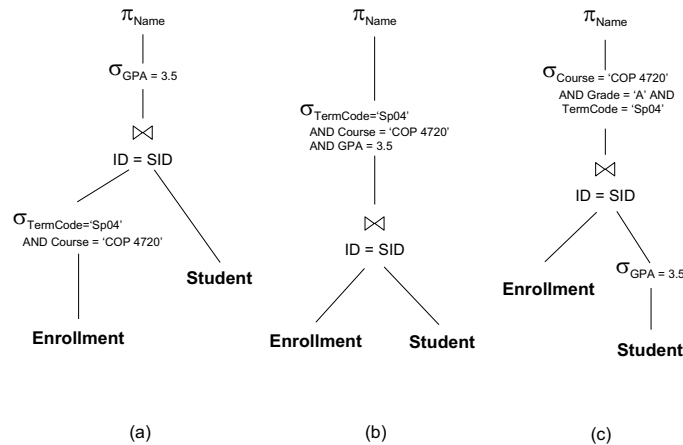

FIGURE 60.6: Sample expression trees representing three equivalent logical query plans

Note that expression trees representing physical query plans differ in the information that is stored in the nodes. For example, internal nodes contain information such as the operation being performed, any parameters if necessary, general strategy about the algorithm that is used, whether or materialization of intermediate results or pipelining is used, and the anticipated number of buffers the operation will require (rather than result size as in logical query plans). At the leaf nodes table names are replaced by scan operators such as `TableScan`, `SortScan`, `IndexScan`, etc.

However, there is an interesting twist to the types of expression trees that are actually considered by the query optimizer. As we have previously pointed out, the number of equivalent query plans (both logical and physical) for a given query can be very large. This is even more so the case, when the query involves the join of two or more relations since we need to take the join order into account when choosing the best possible plan. In general, join operations perform best when the left argument (i.e., the outer relation) is the smaller of the two relations. Today's query optimizers take advantage of this fact by pruning a large portion of the entire search space and concentrating only on the class of *left-deep trees*. A left-deep tree is an expression tree in which the right child of each join is a leaf (i.e., a base table). For example, in Figure 60.6, the tree labeled (a) is an example of a left-deep tree (the tree labeled (b) is an example of a *nonlinear* or bushy tree, tree (c) is an example of a *right-deep tree*).

There are two important advantages for considering on left-deep expression trees: (1) For a given number of leaves, the number of left-deep trees is not nearly as large as the number of all trees, enabling the query processor to consider queries with more relations than is otherwise possible. (2) Left-deep trees allow the query optimizer to generate more efficient plans by avoiding the intermediate storage (materialization) of join results. Note that in most join algorithms, the inner table must be materialized because we must examine the entire inner table for each tuple of the outer table. However, in a left-deep tree, all inner tables exist as base tables and are therefore already materialized.

IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all search for left-deep trees using a dynamic programming approach. However, Oracle also considers right-deep trees and DB2 generates some nonlinear trees. Sybase ASE goes so far as even allowing users to explicitly edit the query plan whereas IBM DB2 allow users to adjust the optimization level which influences how many plans the optimizer considers (see [9]).

60.2.5 Histograms

Whether choosing a logical query plan or constructing a physical query plan from a logical query plan, the query evaluation engine needs to have information about the expected cost of evaluating the expressions in a query. As we mentioned above, the "cost" of evaluating an expression is approximated by several parameters including the size of any intermediary results that are produced during the evaluation, the size of the output, and the types of algorithms that are chosen to evaluate the operators. Statistics include number of tuples in a relation, number of disk blocks used, available indexes, etc. Frequent computation of statistics, especially in light of many changes to the database, lead to more accurate cost estimates. However, the drawback is increased overhead since counting tuples and values is expensive. Most systems compromise by gathering statistics periodically, during query run-time, but also allow the administrator to specifically request a refresh of the statistics.

An important data structure for cost estimation is the *histogram*, which is used by the DBMS to *approximate* the distribution of values for a given attribute. Note that in all but the smallest databases, counting the exact occurrence of values is usually not an option. Having access to accurate distributions is essential in determining how many tuples satisfy a

certain selection predicate, for example, students with a `GPA` value of 3.5. This is especially important in the case of joins, which are among the most expensive operations. For example, if a value of the join attribute appears in the histograms for both relations, we can determine exactly how many tuples of the result will have this value.

Using a histogram, the data distribution is approximated by dividing the range of values, for example, `GPA` values, into subranges or *buckets*. Each bucket contains the number of tuples in the relation with GPA values within that bucket. Histograms are more accurate than assuming a uniform distribution across all values.

Depending on how one divides the range of values into the buckets, we distinguish between *equiwidth* and *equidepth* histograms [9]: In equiwidth histograms, the value range is divided into buckets of equal size. In equidepth histograms, the value range is divided so that the number of tuples in each bucket is the same (usually within a small delta). In both cases, each bucket contains the average frequency. When the number of buckets gets large, histograms can be compressed, for example, by combining buckets with similar distributions.

Consider the `Students-Enrollments` scenario from above. Figure 60.7 depicts two sample histograms for attribute `GPA` in relation `Student`. For this example we are assuming that GPA values are rounded to one decimal and that there are 50 students total. Furthermore, consider the selection $GPA = 3.5$. Using the equidepth histogram, we are led to bucket 3, which contains only the GPA value 3.5 and we arrive at the correct answer, 10 (vs. $1/2$ of $12 = 6$ in the equiwidth histogram). In general, equidepth histograms provide better estimates than equiwidth histograms. This is due to the fact that buckets with very frequently occurring values contain fewer values, and thus the uniform distribution assumption is applied to a smaller range of values, leading to a more accurate estimate. The converse is true for buckets containing infrequent values, which are better approximated by equiwidth histograms. However, in query optimization, good estimation for frequent values are more important. See [4] for a more detailed description of the usage of histograms in query optimization.

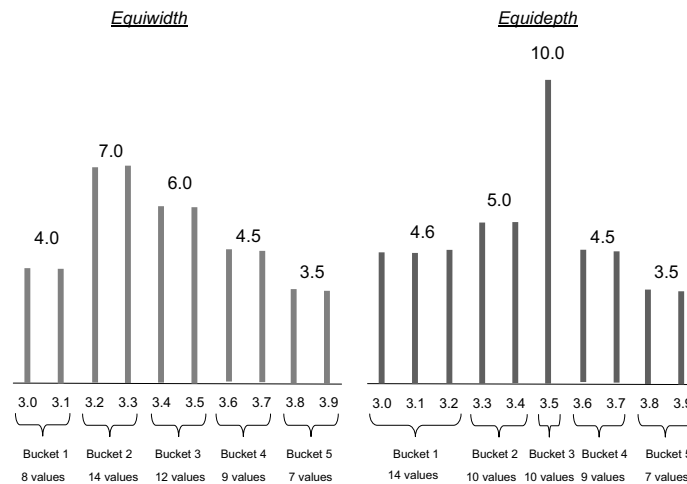Histograms are used by the query optimizers of all of the major DBMS vendors. For

FIGURE 60.7: Sample histograms approximating the distribution of grades values in relation `Student`

example, Sybase ASE, IBM DB2, Informix, and Oracle all use one-dimensional, equidepth histograms. Microsoft's SQL Server uses one-dimensional *equiarea histograms* (a combination of equiwidth and equidepth) [9].

60.3 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 first 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 modified 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 define 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 S_k is assigned to exactly one file F_j, and m segments can be stored in a file. Since block size and page size are the same, each page $P_{k_i} \in S_k$ is assigned to a block $B_{j_l} \in F_j$. We distinguish four methods of realizing this mapping.

The *direct page addressing* assumes an implicitly given mapping between the pages of a segment S_k and the blocks of a file F_j. The page P_{k_i} ($1 \leq i \leq s_k$) is stored in the block B_{j_l} ($1 \leq l \leq d_j$) so that $l = K_j - 1 + i$ and $d_j \geq K_j - 1 + s_k$ holds. K_j denotes the number of the first block reserved for S_k (Figure 60.8). Frequently, we have a restriction to a 1:1-mapping, i.e., $K_j = 1$ and $s_k = d_j$ hold. Only in this case, a dynamic extension of segments is possible. A drawback is that at the time of the segment creation the assigned file area has to be allocated so that a block is occupied for each empty page. For segments whose data stock grows slowly, the fixed block allocation leads to a low storage utilization.

The *indirect page addressing* offers a much larger flexibility for the allocation of pages to blocks and, in addition, dynamic update and extension functionality (Figure 60.9). It requires two auxiliary data structures.

---

[2]A block is a contiguous sequence of bytes and represents the unit used for both storage allocation and data transfer. It is usually a multiple of 512 Bytes and has a typical size of 1KB to 8KB. It may contain several data items. Usually, a data item does not span two or more blocks.
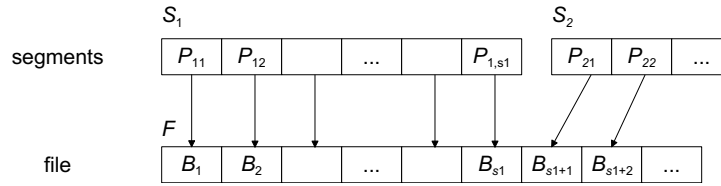
FIGURE 60.8: Direct page addressing

- Each segment S_k is associated with a *page table* T_k which for each page of the segment contains an entry indicating the block currently assigned to the page. Empty pages obtain a special null value in the page table.
- Each file F_j is associated with a *bit table* M_j which serves for free disk space management and quotes for each block whether currently a page is mapped to it or not. $M_j(l) = 1$ means that block B_{j_l} is occupied; $M_j(l) = 0$ says that block B_{j_l} is free. Hence, the bit table enables a dynamic assignment between pages and blocks.

Although this concept leads to an improved storage utilization, for large segments and files, the page tables and bit tables have to be split because of their size, transferred into main memory and managed in a special buffer. The provision of a page P_{k_i} that is not in the buffer can require two physical block accesses (and two enforced page removals), because, if necessary, the page table T_k has to be loaded first in order to find the current block address $j = T_k(i)$.
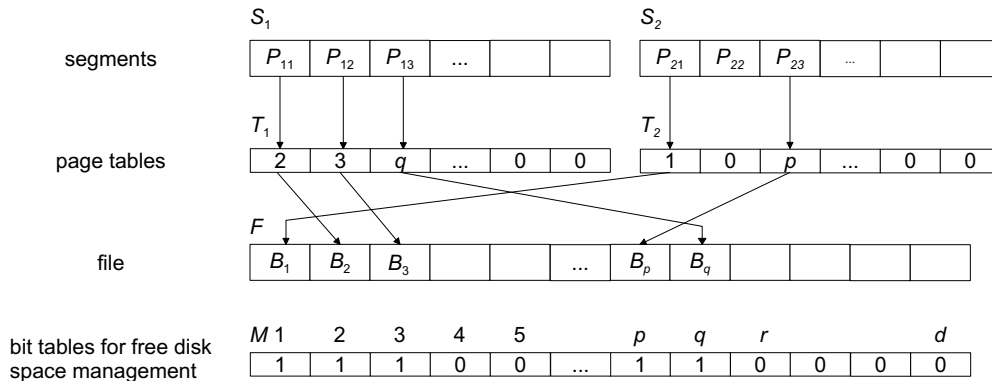
FIGURE 60.9: Indirect page addressing

The two methods described so far assume that a modified page is written back to the block that has once been assigned to it (*update in place*). If an error occurs within a transaction, as a result of the direct placement of updates, the recovery manager must provide enough log information (*undo* information) to be able to restore the old state of a page. Since the writing of large volumes of log data leads to notable effort, it is often beneficial to perform updates in a page in a manner so that the old state of the page is available until the end of the transaction. The following two methods are based on an indirect update of changes and provide extensive support for recovery.

The *twin slot method* can be regarded as a modification of the direct page addressing. It causes very low costs for recovery but partially compensates this advantage through double disk space utilization. For a page P_{k_i} of a segment S_k, two physically consecutive blocks $B_{j_{l-1}}$ and B_{j_l} of a file F_j with $l = K_j - 1 + 2 \cdot i$ are allocated. Alternately, at the beginning of a transaction, one of both block keeps the current state of the page whereas changes are written to the other block. In case of a page request, both blocks are read, and the block with the more recent state is provided as the current page in the buffer. The block with the older state then stores the changed page. By means of page locks, a transaction-oriented recovery concept can be realized without explicitly managing log data.

The *shadow paging concept* (Figure 60.10) represents an extension of the indirect page addressing method and also supports indirect updates of changes. Before the beginning of a new *save-interval* given by two *save-points*[3] the contents of all current pages of a segment are duplicated as so-called *shadow pages* and can thus be kept unchanged and consistent. This means, when a new save-point is created, all data structures belonging to the representation of a segment S_k (i.e., all occupied pages, the page table T_k, the bit table M) are stored as a consistent snapshot of the segment on disk. All modifications during a save-interval are performed on copies T_k' and M' of T_k and M. Changed pages are not written back to their original but to free blocks. At the creation of a new save-point, which must be an atomic operation, the tables T_k' and M' as well as all pages that belong to this state and have been changed are written back to disk. Further, all those blocks are released whose pages were subject to changes during the last save-interval. This just concerns those shadow pages for which a more recent version exists. At the beginning of the next save-interval the current contents of T_k' and M' has to be copied again to T_k and M. In case of an error within a save-interval, the DBMS can roll back to the previous consistent state represented by T_k and M.

As an example, Figure 60.10 shows several changes of pages in two segments S_1 and S_2. These changes are marked by so-called *shadow bits* in the page tables. Shadow bits are employed for the release of shadow pages at the creation time of new save-points. If a segment consists of s pages, the pertaining file must allocate s further blocks, because each changed page occupies two blocks within a save-interval.

The save-points orientate themselves to segments and not to transaction borders. Hence, in an error case, a *segment-oriented recovery* is executed. For a *transaction-oriented recovery* additional log data have to be collected.

60.4 Data Structures for Disk Space Management

Placing data items on disc is usually performed at different logical granularities. The most basic items found in relational or object-oriented database systems are the values of attributes. They consist of one or several bytes and are represented by *fields*. Fields, in turn, are put together in collections called *records*, which correspond to tuples or objects. Records need to be stored in physical *blocks* (see Section 60.3). A collection of records that forms a relation or the extent of a class is stored in some useful way as a collection of blocks, called a *file*.

---

[3]Transactions are usually considered as being atomic. But a limited concept of "subtransactions" allows one to establish intermediate *save-points* while the transaction is executing, and subsequently to roll back to a previously established save-point, if required, instead of having to roll back all the way to the beginning. Note that updates made at save-points are invisible to other transactions.
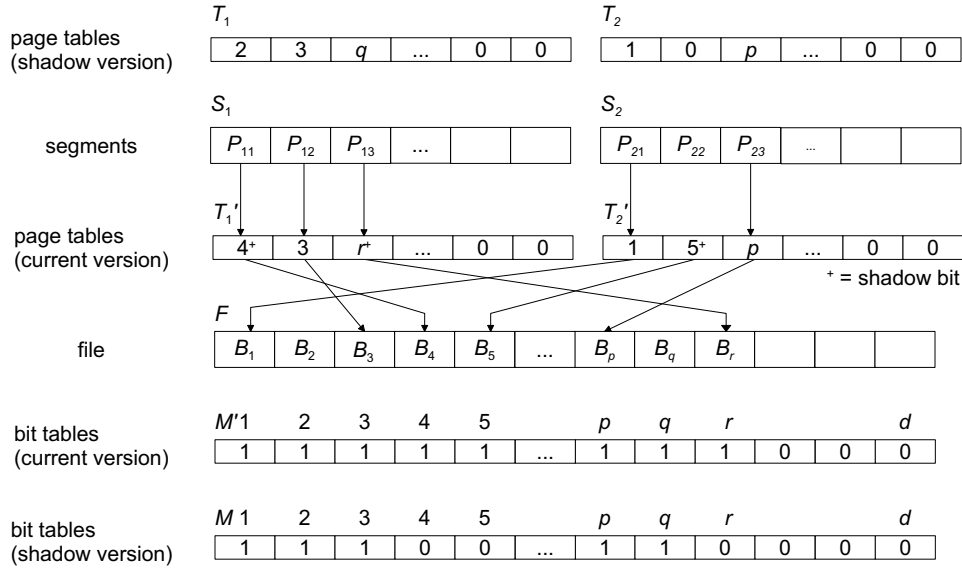
FIGURE 60.10: The shadow paging concept (segments S_1 and S_2 currently in process)

60.4.1 Record Organizations

A collection of field names and their corresponding data types constitute a *record format* or *record type*. The data type of a field is usually one of the standard data types (e.g., *integer*, *float*, *bool*, *date*, *time*). If all records in a file have the same size in bytes, we call them *fixed-length records*. The fields of a record all have a fixed length and are stored consecutively. If the *base address*, i.e., the start position, of a record is given, the address of a specific field can be calculated as the sum of the lengths of the preceding fields. The sum assigned to each field is called the *offset* of this field. Record and field information are stored in the data dictionary. Figure 60.11 illustrates this record organization.
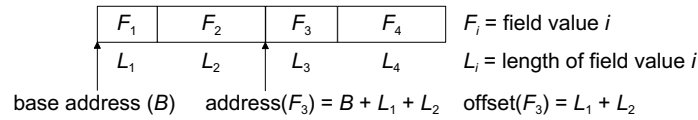
FIGURE 60.11: Organization of records with fields of fixed length

    Fixed-length records are easy to manage and allow the use of efficient search methods. But this implies that all fields have a size so that all data items that potentially are to be stored may find space. This can lead to a waste of disk space and to more unfavorable access times.

    If we assume that each record of a file has the same, fixed number of fields, a *variable-length record* can only be formed if some fields have a variable length. For example, a string representing the name of an employee can have a varying length in different records. Different data structures exist for implementing variable-length records. A first possible organization amounts to a consecutive sequence of fields which are interrupted by separators

(such as ? or % or \$). *Separators* are special symbols that do not occur in data items. A special *terminator* symbol indicates the end of the record. But this organization requires a pass (scan) of the record to be able to find a field of interest (Figure 60.12). Instead of separators, each field of variable length can also start with a counter that specifies the needed number of bytes of a field value.
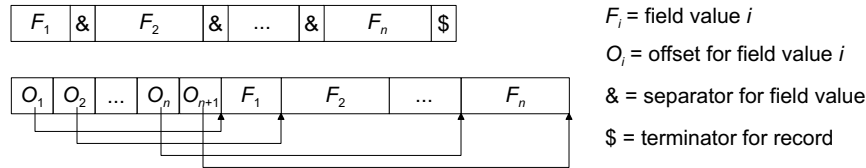

FIGURE 60.12: Alternative organizations of records with fields of variable length

Another alternative is that a header precedes the record. A *header* represents the "administrative" part of the record and can include information about integer offsets of the beginnings of the field values (Figure 60.12). The ith integer number is then the start address of the ith field value relatively to the beginning of the record. Also for the end of the record we must store an offset in order to know the end of the last field value. This alternative is usually the better one. Costs arise due to the header in terms of storage; the benefit is direct field access. Problems arise with changes. An update can let a field value grow which necessitates a "shift" of all consecutive fields. Besides, it can happen that a modified record does not fit any more on the page assigned to it and has to be moved to another page. If record identifiers contain a page number, on this page the new page number has to be left behind pointing to the new location of the record.

A further problem of variable-length records arises if such a record grows to such an extent that it does not fit on a page any more. For example, field values storing image data in various formats (e.g., GIF or JPEG), movies in formats such as MPEG, or spatial objects such as polygons can extend from the order of many kilobytes to the order of many megabytes or even gigabytes. Such truly large values for records or field values of records are called *large objects* (*lobs*) with the distinction of *binary large objects* (*blobs*) for large byte sequences and *character large objects!character* (*clobs*) for large strings.

Since, in general, lobs exceed page borders, only the non-lob fields are stored on the original page. Different data structures are conceivable for representing lobs. They all have in common that a lob is subdivided into a collection of linked pages. This organization is also called *spanned*, because records can span more than one page, in contrast to the *unspanned* organization where records are not allowed to cross page borders. The first alternative is to keep a pointer instead of the lob on the original page as attribute value. This pointer (also called *page reference*) points to the start of a linked page or block list keeping the lob (Figure 60.13(a)). Insertions, deletions, and modifications are simple but direct access to pages is impossible. The second alternative is to store a *lob directory* as attribute value (Figure 60.13(b)). Instead of a pointer, a directory is stored which includes the lob size, further administrative data, and a *page reference list* pointing to the single pages or blocks on a disk. The main benefit of this structure is the direct and sequential access to pages. The main drawback is the fixed and limited size of the lob directory and thus the lob. A lob directory can grow so much that it needs itself a lob for its storage.

The third alternative is the usage of *positional B^+-trees* (Figure 60.14). Such a B-tree
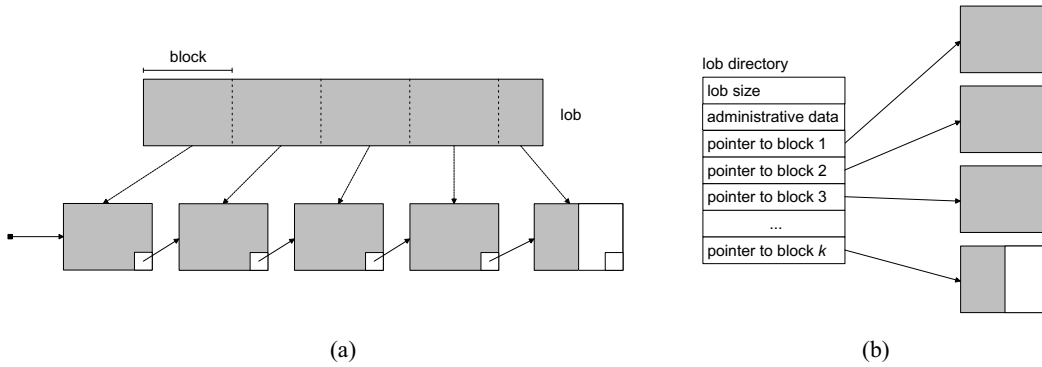
FIGURE 60.13: A lob as a linked list of pages (a), and the use of a lob directory (b)

variant stores relative byte positions in its inner nodes as separators. Its leaf nodes keep the actual data pages of the lob. The original page only stores as the field value a pointer to the root of the tree.
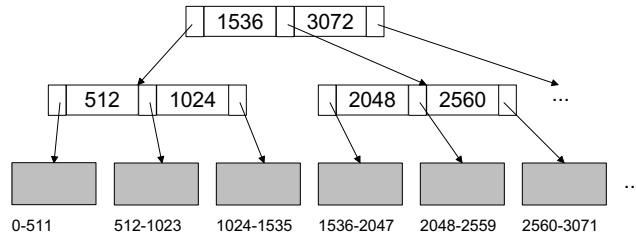

FIGURE 60.14: A lob managed by a positional B^+-tree

60.4.2 Page Organizations

Records are positioned on pages (or blocks). In order to reference a record, often a *pointer* to it suffices. Due to different requirements for storing records, the structure of pointers can vary. The most obvious pointer type is the *physical address* of a record on disk or in a virtual storage and can easily be used to compute the page to be read. The main advantage is a direct access to the searched record. But it is impossible to move a record within a page, because this requires the locating and changing of all pointers to this record. We call these pointers *physical pointers*. Due to this drawback, a pointer is often described as a pair (p, n) where p is the number of the page where the record can be found and where n is a number indicating the location of the record on the page. The parameter n can be interpreted differently, e.g., as a relative byte address on a page, as a number of a slot, or as an index of a directory in the *page header*. The entry at this index position yields the relative position of the record on the page. All pointers (s, p) remain unchanged and are named *page-related pointers*. Pointers that are completely stable against movements in main memory can be achieved if a record is associated with a *logical address* that reveals nothing about its storage. The record can be moved freely in a file without changing any pointers.

This can be realized by *indirect addressing*. If a record is moved, only the respective entry in a *translation table* has to be changed. All pointers remain unchanged, and we call them *logical pointers*. The main drawback is that each access to a record needs an additional access to the translation table. Further, the table can become so large that it does not fit completely in main memory.

A page can be considered as a collection of *slots*. Each slot can capture exactly one record. If all records have the same length, all slots have the same size and can be allocated consecutively on the page. Hence, a page contains so many records as slots fit on a page plus page information like directories and pointers to other pages. A first alternative for arranging a set of N fixed-length records is to place them in the first N slots (see Figure 60.15). If a record is deleted in slot $i < N$, the last record on the page in slot N is moved to the free slot i. However, this causes problems if the record to be moved is pinned[4] and the slot number has to be changed. Hence, this "packed" organization is problematic, although it allows one to easily compute the location of the ith record. A second alternative is to manage deletions of records on each page and thus information about free slots by means of a directory represented as a bitmap. The retrieval of the ith record as well as finding the next free slot on a page require a traversal of the directory. The search for the next free slot can be sped up if an additional, special field stores a pointer on the first slot whose deletion flag is set. The slot itself then contains a pointer to the next free slot so that a chaining of free slots is achieved.
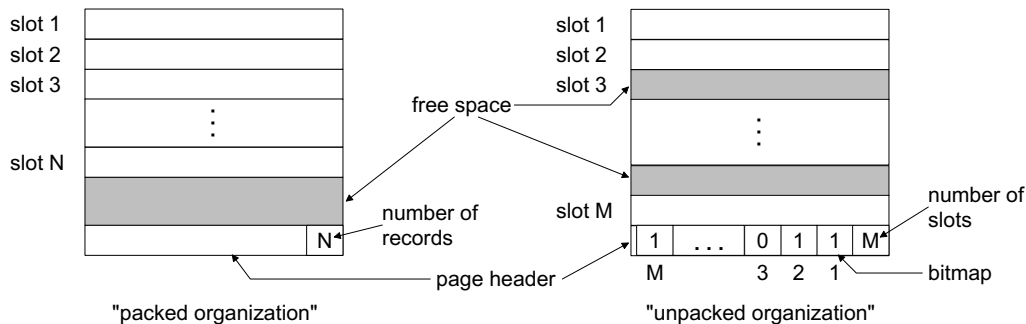

FIGURE 60.15: Alternative page organizations for fixed-length records

Also variable-length records can be positioned consecutively on a page. But deletions of records must be handled differently now because slots cannot be reused in a simple manner any more. If a new record is to be inserted, first a free slot of "the right size" has to be found. If the slot is too large, storage is wasted. If it is too small, it cannot be used. In any case, unused storage areas (*fragmentation*) at the end of slots can only be avoided if records on a page are moved and condensed. This leads to a connected, free storage area. If the records of a page are unpinned, the "packed" representation for fixed-length records can be adapted. Either a special terminator symbol marks the end of the record, or a field at the beginning of the record keeps its length. In the general case, indirect addressing is needed which permits record movements without negative effects and without further access costs.

---

[4]If pointers of unknown origin reference a record, we call the record *pinned*, otherwise *unpinned*.

The most flexible organization of variable-length records is provided by the *tuple identifier* (*TID*) *concepttuple identifier concept* (Figure 60.16). Each record is assigned a unique, stable pointer consisting of a page number and an index into a page-internal directory. The entry at index i contains the relative position, i.e., the offset, of slot i and hence a pointer to record i on the page. The length information of a record is stored either in the directory entry or at the beginning of the slot (L_i in Figure 60.16). Records which grow or shrink can be moved on the page without being forced to modify their TIDs. If a record is deleted, this is registered in the corresponding directory entry by means of a deletion flag.
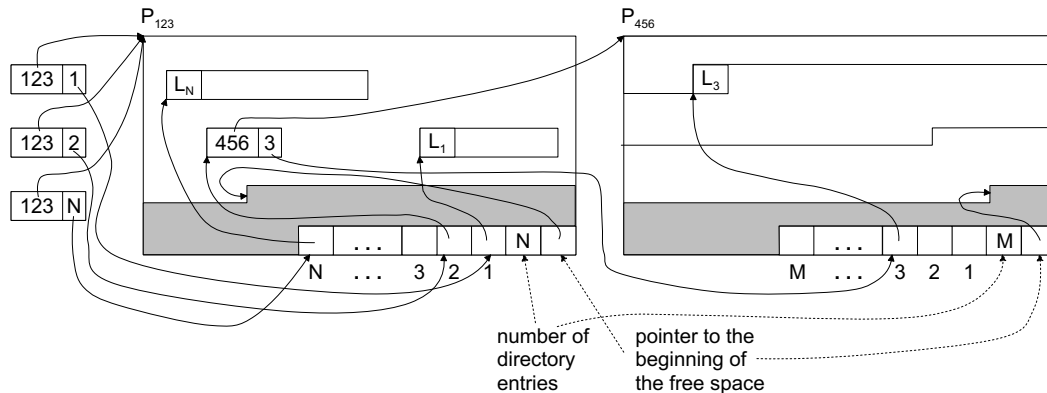
FIGURE 60.16: Page organization for variable-length records

Since a page cannot be subdivided into predefined slots, some kind of free disk space management is needed on each page. A pointer to the beginning of the free storage space on the page can be kept in the page header. If a record does not fit into the currently available free disk space, the page is compressed (i.e., defragmented) and all records are placed consecutively without gaps. The effect is that the maximally available free space is obtained and is located after the record representations.

If, despite defragmentation, a record does still not fit into the available free space, the record must be moved from its "home page" to an "overflow page". The respective TID can be kept stable by storing a "proxy TID" instead of the record on the home page. This proxy TID points to the record having been moved to the overflow page. An overflow record is not allowed to be moved to another, second overflow page. If an overflow record has to leave its overflow page, its placement on the home page is attempted. If this fails due to a lack of space, a new overflow page is determined and the overflow pointer is placed on the home page. This procedure assures that each record can be retrieved with a maximum of two page accesses.

If a record is deleted, we can only replace the corresponding entry in the directory by a deletion flag. But we cannot compress the directory since the indexes of the directory are used to identify records. If we deleted an entry and compress, the indexes of the subsequent slots in the directory would be decremented so that TIDs would point to wrong slots and thus wrong records. If a new record is inserted, the first entry of the directory containing a deletion flag is selected for determining the new TID and pointing to the new record.

If a record represents a large object, i.e., it does not fit on a single page but requires a collection of linked pages, the different data structures for blobs can be employed.

60.4.3 File Organization

A *file* (*segment*) can be viewed as a sequence of *blocks* (*pages*). Four fundamental file organizations can be distinguished, namely files 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 unordered, 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 management 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 octtrees, R-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*.
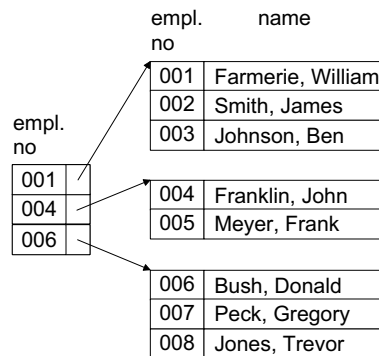
FIGURE 60.17: Example of a clustered, sparse index as a secondary organization on a sorted file

60.5 Conclusion

A modern database management system is a complex software system that leverages many sophisticated 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., spatio-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

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, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, first edition, 2002.

[5] Goetz Graefe. Query evaluation techniques for large databases. *Computing Surveys*, 25(2):73–170, 1993.

[6] Goetz ed. Graefe. Special issue on query processing in commercial database management systems. *Data Engineering Bulletin, IEEE*, 16(4), 1993.

[7] Yannis Ioannides. Query optimization. In A.B. Tucker, editor, *Handbook of Computer science*. CRC Press, 2002.

[8] Matthias Jarke and Juergen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

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

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

Index