Chapter 1

Introduction to Relational Query Processing

In 1970, Codd's classical paper "A Relational Model for Large Shared Data Bank" laid the foundation for relational database systems. Since then, many commercial relational database systems such as ORACLE, DB2 and SYBASE have been built. In fact, relational database systems have dominated the database market for many years. The remarkable success of relational database technology can be attributed to many aspects and some of the key factors are (1) it has a solid mathematical foundation; (2) it employs an easy-to-use query language, i.e., SQL (*Structured Query Language*). SQL is a *declarative* (or *non-procedural*) language in the sense that users only need to specify *what* data they want from the database but do not need to specify *how* to get the desired data; (3) it employs sophisticated query processing and optimization techniques so that user queries can be processed efficiently.

In this chapter, we provide an introduction to the processing and optimization techniques of relational queries in a centralized uniprocessor environment. Many of the terminologies introduced in this chapter are essential for the understanding of other chapters of the book. In Section 1.1, a brief review of the relational data model will be provided. An overview of problems in query processing and optimization will be given in Section 1.2. Indexing techniques commonly used in relational database systems will be briefly described in Section 1.3. The evaluation of several relational operations, including selection, projection and join, and the analysis of the evaluation cost will be presented in Section 1.4. Approaches for determining the order of processing multiple operations will be discussed in Section 1.5.

1.1 The Relational Data Model

In this section, we briefly review the relational data model, basic relational operators and the relational query language SQL.

1.1.1 Data Model

In a relational database, data are organized into table format. Each table (or *relation*) consists of a set of *attributes* describing the table. Each attribute corresponds to one column of the table. Each attribute is associated with a domain indicating the set of values the attribute can take. Each row of a table is called a *tuple* and it is usually used to describe one real-world entity and/or a relationship among several entities. For any tuple and any attribute of a relation, it is required that the value of the tuple under the attribute be atomic (e.g., no composite value or set value is allowed). For each relation, there exists an attribute or a combination of attributes such that no two tuples in the relation can have the same values under the attribute or the combination of attributes. Such an attribute or combination of attributes is called a *key* of the relation. When a key consists of multiple attributes, it is usually required that every attribute in it is needed for it to be a key. In other words, when any attribute is removed from a key, the remaining attribute(s) no longer form a key. In summary, each relation must have at least one key. But a relation may have multiple keys. In this case, one of them will be designated as the *primary key*.

Example 1.1 *Two relations,* Employee *and* Department*, are shown in Figure 1.1. The first relation has five attributes and* SSN *(Social Security Number) is the key of the relation. The second relation has three attributes and* Name *is the key of this relation. Each tuple in* Employee *represents an employee entity. In addition, it also represents a relationship,* works_in*, between an employee and a department. For example, the first tuple describes an employee as SSN = 123456789, Name = John, Age = 34 and Salary = 30000 and it also describes the relationship that John works in the Sales department. Each tuple in* Department *represents a department entity and no relationship.* ■

SSN	Name	Age	Salary	Dept
123456789	John	34	30k	Sales
234567891	Ketty	27	25k	Service
345678912	Wang	39	32k	Sales

Employee

Name	Budget	Phone
Sales	200k	4567890
Service	100k	5678900
Advertising	100k	6789123

Department

Figure 1.1: Two Sample Relations

If the value of an attribute of a tuple in a relation is *not available* or *not defined*, then a *null value*, denoted as *null*, is used to occupy its position. For example, when the budget of a relation has not been decided, it can be left as null. The *entity integrity constraint* of the relational database requires that no attribute in the primary key may take *null* values.

Two attributes of different relations (or even the same relation) may be closely related. For example, the *Dept* attribute of *Employee* and the *Name* attribute of *Department* are closely related since they both take department names as their values. Since an employee can either be without a department temporarily or work in an existing department, it makes sense to require that the *Dept* attribute takes only values currently under the *Name* attribute of *Department* or null values. If this requirement is enforced, we say that the attribute *Dept* of *Employee* references the attribute *Name* of *Department*, or that the *referential integrity constraint* between the two relations is satisfied. Often, the attribute being referenced (e.g., the *Name* attribute of *Department* in the above example) is the key of a relation. In this case, the referencing attribute (e.g., the *Dept* attribute of *Employee*) is called a *foreign key* of *Employee*.

1.1.2 Relational Algebra

The relational algebra is a collection of operations that are used to manipulate relations. Each operation takes one or two relations as the input and produces a new relation as the output. The basic relational algebra operations and their notations are described below. In the following discussion, R, R_1 and R_2 denote relations.

Selection (σ): $\sigma_C(R)$ is used to find all tuples of R that satisfy the selection conditions specified in C. A simple selection condition can be "*A op v*" or "*A op B*", where A and B are attributes of R, *op* is an operator in $\{=, \neq, <, \leq, >, \geq\}$, and v is a value in the domain of A. A more complicated condition can be constructed from simple conditions through logical operators in {and, or, not}. As an example, for the relation *Employee* in Figure 1.1, the query to find all employees who are older than 30 and who earn no more than 30k can be expressed as $\sigma_{Age>30 \ and \ Salary \leq 30k}(Employee)$.

Projection (π): $\pi_{AL}(R)$ is used to return, for each tuple in R, the values under attributes listed in AL. In other words, the columns of R which are not specified in AL are removed. It is required that all attributes in AL are from R. The query to find the names and salaries of all employees can be written as $\pi_{Name,Salary}(Employee)$. Duplicate tuples may result from projection. For example, $\pi_{Budget}(Department)$ may have two tuples with 100k. Such duplication is automatically removed by the projection operator.

Cartesian Product ($\times$): $R_1 \times R_2$ returns a new relation that contains all tuples that can be obtained by concatenating every tuple in R_1 with every tuple in R_2. Cartesian product provides a means to relate tuples from different relations. Cartesian product is a very expensive operation because it performs exhaustive tuple pair concatenation and it generates a very large result.

Union ($\cup$): $R_1 \cup R_2$ returns all tuples that belong to either R_1 or R_2. The union operator requires the union compatibility of its two operands. Two relations are *union compatible* if they have the same number of attributes and each pair of corresponding attributes have the same domain.

Set Difference ($-$): $R_1 - R_2$ returns all tuples that belong to R_1 but not R_2. This operator also requires the union compatibility of its two operands.

Set Intersection ($\cap$): $R_1 \cap R_2$ returns all tuples that belong to both R_1 and R_2. This operator also requires the union compatibility of its two operands.

Join ($\bowtie_C$): $R_1 \bowtie_C R_2$ returns all tuples in $R_1 \times R_2$ that satisfy the join condition C. C consists of one or more *basic join conditions* connected by logical operators. Each basic join condition is of the form "R_1.A op R_2.B". That is, each join condition compares attribute values of tuples from different relations. In comparison with Cartesian product, join is a more efficient alternative to relate tuples from different relations since it puts restriction on tuples in the output, i.e., a tuple in R_1 concatenates a tuple in R_2 only if the two tuples satisfy the join condition C.

If the operator in every basic join condition of a join is the equal operator, the join is called an *equi-join*. Otherwise, it is called a θ-join. Equi-joins are very frequently used because most joins are equi-joins between the foreign key of a relation and its corresponding key in another relation. Most relations are related through the key/foreign-key relationships between them. For example, to relate the employees with their corresponding departments (see Figure 1.1), the following equi-join is used: Employee $\bowtie_{Employee.Dept\ =\ Department.Name}$ Department. A relation generated by an equi-join has columns that have identical contents. For example, the above equi-join will generate a relation with two columns, Employee.Dept and Department.Name, that have identical values for each tuple in the relation. Such redundant columns are automatically removed by the *natural join*. The natural join between R1 and R2, usually denoted as $R_1 \bowtie R_2$ without any join condition explicitly specified, evaluates an equality condition between every pair of identically-named attributes from R_1 and R_2 and keeps only one of every pair of these identically-named attributes. If the Dept attribute in Employee and the Name attribute in Department are both renamed as Dept-Name, then the above equi-join with only one column of department name in the result can be replaced by the natural join: Employee $\bowtie$ Department.

Division ($\div$): This operator requires the relation to be divided (i.e., the dividend) contain all attributes of the relation that divides (i.e., the divisor). Suppose R_1 has attributes (A_1, ..., A_n, B_1, ..., B_m) and R_2 has attributes ($B_1, ..., B_m$). Let T = $\pi_{A_1,...,A_n}(R_1)$. Then $R_1 \div R_2$ returns each such tuple of T whose concatenation with every tuple of R_2 is in R_1. Intuitively, $R_1 \div R_2$ finds all the tuples of R_1 that relates with every tuple in R_2. For example, if R_1 is Participate(SSN, Proj#) representing which employee is participating in which project, and R_2 is Project(Proj#) representing a set of projects, then $R_1 \div R_2$ returns the SSNs of those employees who participate in every project.

Among the above eight relational algebra operators, only the first five are fundamental in the sense that they are necessary. Each of the last three operators can be expressed in terms of the fundamental operators. For example, the natural join between R_1(A, B, C) and R_2(B, C, D) can be expressed in terms of selection, projection and Cartesian product as follows: $R_1 \bowtie R_2 = \pi_{A,R_1.B,R_1.C,D}(\sigma_{R_1.B\ =\ R_2.B\ and\ R_1.C\ =\ R_2.C}(R_1 \times R_2))$.

1.1.3 SQL

SQL is the standard language for relational databases. SQL has many components dealing with different aspects of managing data in the database, such as the definition of data, the manipulation of data, interface with different host programming languages (embedded SQL), definition of constraints and the support for transactions. In this chapter, we are mainly interested in the aspect of SQL as a database query language to manipulate data. As a query language, SQL is mostly non-procedural. In contrast, the relational algebra is a procedural language.

A basic SQL query consists of three clauses, namely the select-clause, which specifies the target (output) attributes to be returned, the from-clause, which specifies the involved relations for the query, and the where-clause, which specifies the conditions to be satisfied by the result of the query. Using the relations in Figure 1.1, a query to find the name of each employee in the Sales department together with the department telephone number can be written in SQL as follows.

```
select Employee.Name, Department.Phone
from Employee, Department
where Employee.Dept = 'Sales' and Employee.Dept = Department.Name
```

Approximately, the select-clause corresponds to the *projection* operator, the from-clause corresponds to one or more Cartesion products and the where-clause corresponds to the *selection* operator in the relational algebra. The difference between the select-clause and the *projection* operator is that the former does not remove duplicate tuples while the latter does. To remove duplicate tuples in the result, *select distinct* instead of *select* should be used. There are two reasons for giving the users the choice on whether to keep or remove duplicate tuples. First, duplicate tuples sometimes provide additional information to the user. For example, suppose only Employee.Name is in the select-clause of the above query. If a name appears twice in the result, then we know there are two employees who have the same name. This information is lost if duplicate tuples are removed. Second, removing duplicate tuples incurs a much higher cost than keeping them since the former typically needs to sort the tuples. If *select* is replaced by *select distinct* in the above query, then it is equivalent to the following relational algebra expression: $\pi_{Employee.Name,Department.Phone}(\sigma_{Employee.Dept='Sales'\ and\ Employee.Dept=Department.Name}(Employee \times Department))$. Note that that the above SQL query is equivalent to the above relational algebra expression does not necessarily imply that the SQL query is to be evaluated based on this relational algebra expression. Query optimization techniques will be discussed in Section 1.4 and Section 1.5.

A more complicated SQL query may contain aggregate functions (i.e., *min, max, avg, sum* and *count*), up to three additional clauses (i.e., the *group by* clause, which groups tuples of the input relations based on the values of specified attribute(s) for computing aggregate values for each group, the *having* clause, which specifies conditions on aggregate values, and the *order by* clause, which sorts the output based on the values of specified attribute(s)), nested subqueries (*correlated*, in which an inner query references some attribute in the outer query, or *un-correlated*), and subqueries that are connected by set operators such as union, intersection and set difference.

It can be shown that any query that can be expressed by the relational algebra can also be expressed by SQL. Therefore, SQL is at least as powerful as the relational algebra.

1.2 An Overview of Query Processing and Optimization

When a user query (in SQL) is received, the query processor first checks whether the query has the correct syntax and whether the relations and attributes it references are in the database. Next, if the query is acceptable, then an *execution plan* for the query is generated. Recall that SQL queries are non-procedural. In contrast, an execution plan of a query defines a sequence of steps based on which the query is to be evaluated. Typically, each step in the plan corresponds to one relational operation plus the method to be used for the evaluation of the operation. Usually, for a given relational operation, there are a number of methods that can be used to evaluate it. For example, the join operation can be evaluated by *nested loop*, *sort merge* and other join methods.

Example 1.2 *Consider the following SQL query,*

```
select *
from R, S, T
where R.A > 'a' and R.B = S.B and S.C = T.C
```

where R, S and T are relations, A, B and C are attributes, and a is a constant. This query involves two joins and a selection. A possible execution plan for this query consists of the following three steps:

1. *Perform selection* $\sigma_{A>'a'}(R)$ *based on a sequential scan of the tuples of R. Let R1 denote the result of* $\sigma_{A>'a'}(R)$.

2. *Perform join* $R1 \bowtie_{R1.B=S.B} S$ *using the sort merge join algorithm. Let R2 denote the result of the join.*

3. *Perform join* $R2 \bowtie_{R2.C=T.C} T$ *using the nested loop join algorithm.*

■

For a given query, there may be different execution plans that can produce the same final result. For instance, an alternative execution plan to the one given in Example 1.2 is to modify the second and third steps as follows: (2) Perform join $S \bowtie_{S.C=T.C} T$ using the nested loop join algorithm. Let R3 denote the result. (3) Perform join $R1 \bowtie_{R1.B=R3.B} R3$ using the sort merge join algorithm. Different execution plans that can always produce the same result are said to be *equivalent*. However, different equivalent plans may be evaluated with very different costs. The goal of query optimization is to find an execution plan, among all possible equivalent plans, that can be evaluated with the minimum cost. Such a plan is called an *optimal* plan.

In a centralized database system, the cost of evaluating a query is the sum of two components, i.e., the I/O cost and the CPU cost. The I/O cost is caused by the transfer of data between the main memory and the secondary storage (typically, magnetic disk), as in most applications the data is too voluminous to be held entirely in the memory. The CPU cost is incurred when tuples in the memory are joined or checked against conditions. For most database operations, the I/O cost is the dominant cost. To reduce I/O cost, special data structures such as B^+trees are often utilized. More discussion on these data structures will be given in Section 1.3. For a single processor environment, minimizing the total cost implies the minimization of the *response time* (i.e., the real time needed to complete the query).

The number of equivalent execution plans for a query that references n relations can easily be an exponential function of n. For example, there can be as many as $\frac{n*(n-1)}{2}$ joins between the n relations and $(\frac{n*(n-1)}{2})!$ different orders exist for these joins. In general, the number of equivalent execution plans for a given query is determined by two factors, namely, the number of operations in the query and the number of methods that can be used to evaluate each operation. For example, if there are m operations in a query and each operation can be evaluated in k different ways, then there will be as many as $(m!)*k^m$ different execution plans. The set of all equivalent execution plans is the *search space* for query optimization. Due to the very large number of possible execution plans, the problem of finding an optimal execution plan for a general query becomes a very difficult problem. It has been shown that the problem is NP-hard. On the one hand, we would like to find an optimal plan so that we can minimize the query execution cost. On the other hand, we do not want to spend too much time to search for an optimal plan since ultimately the time needed to process a query will be the sum of the time used to find a plan and the time needed to execute the plan. In other words, we need to strike a balance between the two times, namely, spend a reasonable amount of time to find a sufficiently efficient execution plan. The ideas for achieving the balance can be summarized below.

1. For some special types of queries for which an optimal execution plan can be found in a reasonable amount of time, it is worthwhile to find the optimal plan. Some well-known special queries are *chain query*, in which the referenced relations can be listed in such a way that only adjacent relations have joins between them, *star query*, in which there is a *central*

relation that joins with other relations and these are the only joins, and *tree query*, which has a tree-shaped (i.e., has no cycle) *join graph*, where the join graph of a join query contains relations as vertices and joins between relations as edges. A *dynamic programming* based algorithm for processing chain queries will be discussed in Chapter 2. The technique for determining the optimal order in which the joins in a star query are evaluated is similar to that for determining the optimal order in which the child nodes of a node are traversed in processing a query in object-oriented database systems (see Chapter 4).

2. For general queries, either heuristics are used to obtain a usually reasonable but not necessarily optimal plan or a reduced search space is used so that an optimal plan based on the reduced space can be found. Neither solution guarantees the finding of a real optimal execution plan. The algebra based optimization technique uses a set of heuristic rules to guide the transformation from one execution plan to another execution plan. One such rule is to process selections as early as possible and another rule is to replace Cartesian products with selections by joins whenever possible. In most cases, these rules can lead to better execution plans. However, in some cases, they may produce worse execution plans. More discussion about algebra based optimization technique will be provided in Section 1.5. System R employs the cost estimation based optimization technique. The basic idea of this technique is to, for each query, estimate the cost of every possible execution plan, and choose the execution plan with the lowest estimated cost. Due to the large number of possible execution plans, System R only attempts to find an optimal execution plan based on a reduced search space. More specifically, only execution plans corresponding to *left-deep join trees* are considered. One of the characteristics of such a join tree is that at least one of the input relations to each join, except for the first join, is an intermediate result. More discussion on different types of join trees can be found in Chapter 3. More discussion on cost estimation based query optimization technique will be provided in Section 1.4 and Section 1.5.

1.3 Fast Access Paths

In this section, we describe two data structures, namely B^+tree and hashing, that are used very frequently in database systems for speeding up searches and for reducing I/O costs. But first, we describe a typical storage hierarchy for database applications as it is important for the understanding of I/O cost.

1.3.1 Storage Hierarchy

A typical storage hierarchy for database applications consists of two levels. At the first level is the main memory (or primary storage), which is a storage media that can be operated on directly by the CPU. At the second level is the secondary storage (typically, magnetic disk or disk pack). Data stored on secondary storage can not be directly processed by the CPU; it must be brought to the main memory first. The characteristics of the two storages can be summarized below[1].

Characteristics of main memory:

- fast access to data (0.01 μs per instruction for a processor of 100 mips).
- small storage capacity (typically a few dozen megabytes).
- volatile, i.e., contents will be lost upon power failure.
- expensive (about \$10/per megabytes).

Characteristics of secondary storage:

- slow access to data (about 25 msec. per page).
- large storage capacity (in the order of gigabytes).
- non-volatile.
- cheap (about \$200/per gigabytes).

Although the costs for both main memory and secondary storage have been falling sharply and this trend is likely to continue, the cost ratio between them has been reasonable steady, with main memory costing a few dozens of times more than the secondary storage for the same capacity. Even though main memory has become more and more affordable, using it to store all data in a database is still too expensive for most applications as database systems are being used in more and more large applications.

A typical disk pack consists of a number of disks sharing the same spindle. Each disk has two surfaces. Each surface typically has a few hundreds of information storing circles and each such circle is called a *track*. The set of tracks with the same diameter on all disk surfaces is called a *cylinder*. Each track is typically partitioned into many *pages* (*sectors* or *blocks*). The size of a page is typically 2KB (for Unix Workstations) and 4KB (for IBM Mainframes). Page is the smallest unit for transferring data between the main memory and the secondary storage. In summary, disk based secondary storage is typically organized into pages, tracks and cylinders. Each page can be addressed through a triplet (cylinder#, track#, page#).

The cost of transferring one page of data between the main memory and the disk is the sum of three components. The first component is the *seek time* which is the time needed to position the read/write head of the disk drive on the correct track. The second component is the *rotational delay* (or *latency*) which is the time needed for the beginning of the page to rotate under the read/write

[1] Note that the numbers quoted in the characteristics of main memory and secondary storage may become obsolete quickly as the technology advances quickly.

head. The third component is the *block transfer time* which is the actual time needed to transfer the data in the page. The seek time heavily depends on the relative positions of the read/write head and the desired page on the disk. If the read/write head is currently on the outermost track and the desired page in on the innermost track, then the seek time will be very long. On the other hand, if they happen to be on the same track when the transfer is needed, then the seek time will be zero. The expected seek time is the time for traversing half of the tracks on the surface and it is around 15 msec. The rotational delay depends on the relative position of the read/write head and the beginning of the desired page. In the worst case, a full rotation is needed and in the best case, no rotation is needed. On the average, half a rotation is expected. The expected rotational delay takes about 8 msec. The time needed to actually transfer one page of data is only about 1 msec. Clearly, most of the time in transferring one page of data is spent on positioning the read/write head to the right place and the actual transferring is very small. Therefore, if we need to transfer two pages of data and the second page is next to the first page on the same track, then after the data in the first page is transferred, the data in the second page can be transferred without additional seek time and rotational delay. In this case, the first page is expected to take 24 msec. to be transferred while the second page takes only 1 msec. A page I/O is *random* if transferring the data requires the re-positioning of the read/write head; otherwise, the page I/O is *sequential*. For our example, transferring the data of the first page incurs a random page I/O while transferring the data of the second page incurs a sequential page I/O. Since a sequential page I/O is much cheaper than a random page I/O, it is important to organize data on disk in such a way that the number of random page I/Os can be reduced.

Although the two-level storage architecture described above is expected to remain typical for most database applications in the foreseeable future, the following architectures are likely to be used in more and more applications.

One-level architecture: In this architecture, all data in the database are kept in the main memory. Such a database system is known as *main memory database system*. Due to the sharp decrease of main memory cost and the availability of non-volatile (battery backup) main memory, main memory database is an attractive choice for applications that require fast data accesses.

Three-level architecture: In recent years, the amount of data involved in several high profile applications is so large that the secondary storage is no longer capable of accommodating them. For example, the database in the SEQUOIA 2000 project for earth science will eventually contain as many as 10 pegabytes of data. Massive databases are also emerging in the areas of environmental study, medical research, telecommunication networks, digital information services, space exploration, etc. In this case, another level of storage, namely the *tertiary storage*, is needed. Optical disks and/or tape drives can be used as the storage media for the tertiary storage and it has the characteristics that it is much slower than the the secondary storage, much larger in capacity and non-volatile.

1.3.2 B$^+$tree

B$^+$tree is an index structure widely used in database systems. A node in the tree is either an internal node or a leaf node. Each node is stored as a page. Suppose the index is based on the values of attribute A of relation R. The internal nodes of a B$^+$tree have the format $(P_1, a_1, P_2, a_2, ..., a_{k-1}, P_k)$, where a_i's are values in attribute A satisfying $a_1 < a_2 < \cdots < a_{k-1}$, and P_i's are *tree pointers* pointing to nodes at the next lower level of the tree. Every value of A in the node pointed to by P_1 is less than or equal to a_1, every value of A in the node pointed to by P_k is greater than a_{k-1}, and every value of A in the node pointed to by P_i is greater than a_{i-1} and less than or equal to a_i, $1 < i \leq k-1$. It is required that every non-root internal node is at least half full. Let M be the maximum number of pointers that can be placed in a node (M can be computed from the size of a page, the size of A-values and the size of a pointer). Then, $\lceil \frac{M}{2} \rceil \leq k \leq M$. The leaf nodes of the tree have the format $(a_1, P_1; a_2, P_2; ...; a_m, P_m; P)$, where a_i's are A-values satisfying $a_1 < a_2 < \cdots < a_m$ and P is a *leaf node pointer* pointing to the next leaf node. Leaf nodes are ordered in ascending A-values, that is, if all A-values in node i are greater than all A-values in node j, then node j will precede node i in the order. The leaf node pointers in leaf nodes provide a way to access the tuples in an ordered manner and are useful for evaluating *range conditions* (e.g., $a \leq A < b$). Pointers P_i's in the leaf nodes are described as follows.

1. If A is a key and the tuples of R are stored in sorted A-values, then, depending on the implementation, each P_i is either a *tuple pointer* pointing to the tuple whose A value is a_i or a *page pointer* pointing to a page of R which contains the tuple whose A value is a_i. In this case, the B$^+$tree index is called a *primary index.*

2. If A is not a key but the tuples of R are stored in sorted A-values, then each P_i is a *page pointer* pointing to a page of R which contains the tuple(s) whose A value is a_i. In this case, the index is called a *clustering index.*

   Note that there can be at most one primary index or one clustering index but not both since there is only one way to store the tuples of a relation at any given time. For example, if we had a primary index on attribute A and a clustering index on attribute B, then the tuples of R would have to be stored in ascending A-values as well as in ascending B-values. This is usually impossible.

3. If the tuples of R are not stored in sorted A-values, then two cases may occur. If A is a key, then each P_i is a *tuple pointer* pointing to the tuple whose A value is a_i. If A is not a key, then each P_i is a pointer pointing to a page that contains tuple pointers pointing to tuple(s) whose A value is a_i. In either case, the index is called a *secondary index.* Notice that the

secondary index in the second case requires an additional indirection since the pointers in the leaf nodes do not point to data pages directly.

It is also required that each leaf node remain at least half full (i.e., has at least $\lceil \frac{N}{2} \rceil$ pointers if N is the maximum number of pointers that can be placed in a leaf node).

B^+tree is a *balanced tree* since all leaf nodes in the tree are at the same level. Figure 1.2 illustrates a B^+tree with the maximum number of pointers in each node (internal or leaf) being 3.

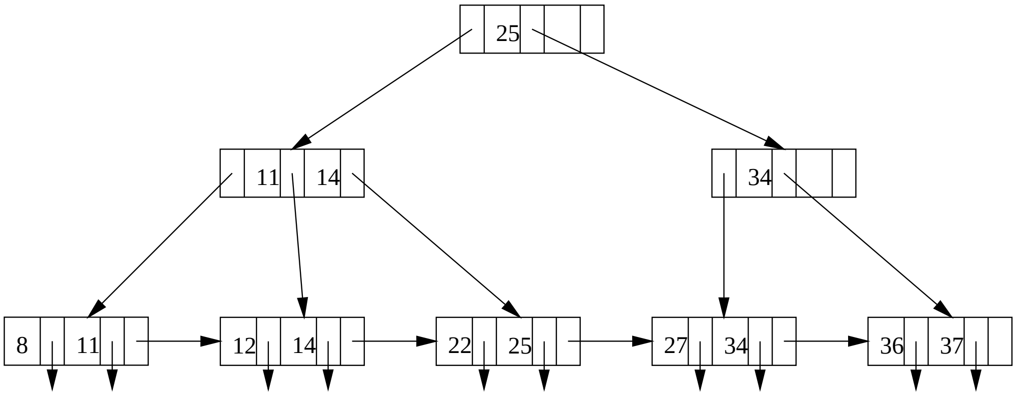

Figure 1.2: An example B^+tree

The algorithm for searching a tuple (or tuples) with A-value equal to a is described below. Let T denote the pointer pointing to the root node of the B^+tree.

Algorithm Search(a, T)

1. If T is an internal node, then compare a with the A-values in T to determine the tree pointer to follow and the next node to search. Specifically, if $a \leq a_1$, then call Search(a, P_1); if $a_{i-1} < a \leq a_i$, $1 < i \leq k-1$, then call Search(a, P_i); if $a > a_{k-1}$, then call Search(a, P_k). The comparison stops as soon as the tree pointer is determined. A binary search is often utilized to speed up the search of the right P_i.

2. If T is a leaf node, compare a with the A-values in T. If no A-value in T is equal to a, report *not found*. If a_i in T is equal to a, follow pointer P_i to fetch the tuple(s).

The above algorithm can be modified easily for searching all tuples whose A-values are in a specified range, say $[a, b]$. This is described as follows. The first step of the above algorithm is used to find a leaf node N for a. Next, the smallest value in N that is greater than or equal to a is determined. If this value is greater than b, then return *not found* and exit. If it is less than or equal to b, then the corresponding pointer is used to fetch the tuple(s). In this case, the successive A-values in N are compared with b, and if they are less than or equal to b, then their corresponding pointers are used to fetch the tuples. If the largest A value in N is less than b, then the search continues to the node next to N using the leaf node pointer in N. The process stops either when an A value greater than b is encountered or all A-values have been considered.

The performance of the search algorithm is closely related with the height of the B^+tree. In theory, if n is the number of distinct A-values in R, then the height of the B^+tree is approximately $\log_F n$, where F is the fanout of the tree. In practice, since F is usually large, a B^+tree of 3 to 4 levels can easily accommodate a very large relation. For example, consider a relation with one million distinct A-values (i.e., R has at least one million tuples). Suppose each page has 2k bytes and each (A-value, pointer) pair has 15 bytes. Suppose each page stores 100 such pairs (25% of the space is left to accommodate future insertions). Then 10,000 leaf nodes are needed. With F = 100, the next upper level needs 100 nodes and 1 node is sufficient for the next upper level (i.e., the root level). For this particular example, a 3-level B^+tree is sufficient. Therefore, in practice, a small constant number is often used to indicate the height of a B^+tree.

While searching in a B^+tree is straightforward, inserting to and deleting from a B^+tree are much more complicated. Much of the complication for insertion come from the fact that inserting a new value into a leaf node, say N, may cause the node to overflow (i.e., no space left in the node to accommodate the new value and the pointer). The overflow causes N to be split into two nodes. As a result, an additional pointer and an A-value need to be added in the parent node of N. The additional pointer and A value may cause the parent node of N to overflow and split. In the worst case, the effect may be propagated all the way to the root node. When the root node is split, the height of the tree will be increased by one. Similarly, deleting a value from a leaf node may cause it to underflow (i.e., the node becomes less than half full) and the underflow causes the merge of this node with one of its sibling nodes. Such a merge may then cause its parent node to underflow. Again, in the worst case, the effect may cascade all the way to the root node. If the root node has only two child nodes, then the merge of its two child nodes will result in the deletion of the root node (the merged child node of the old root becomes the new root of the tree) and the decrease of the height of the tree by one.

Example 1.3 *Consider the B^+tree in Figure 1.2. Suppose we need to insert a tuple with A value equal to 23 into the relation R. This will cause the insertion of the entry (23, P) into the B^+tree, where P is the address of the new tuple. Using a search procedure similar to the one described in the search algorithm, the leaf node containing A-values 22 and 25 is identified to be the node into which (23, P) should be inserted. The insertion causes the node to overflow since the maximum number of pointers allowable is 3. As a result, the node is split into two nodes, N1 with 22 and 23 and N2 with 25. The split results in the need to add a pointer and a value (23, the largest value in N1) to the parent node. This causes the parent node to be split into 2 nodes as shown in Figure 1.3. This split causes the addition of a pointer and the value 14 to the root node (14 is the largest value in the leftmost subtree). This did not cause a split of the root node since in this case, the root node still has room to accommodate the new pointer and the new value.* ■

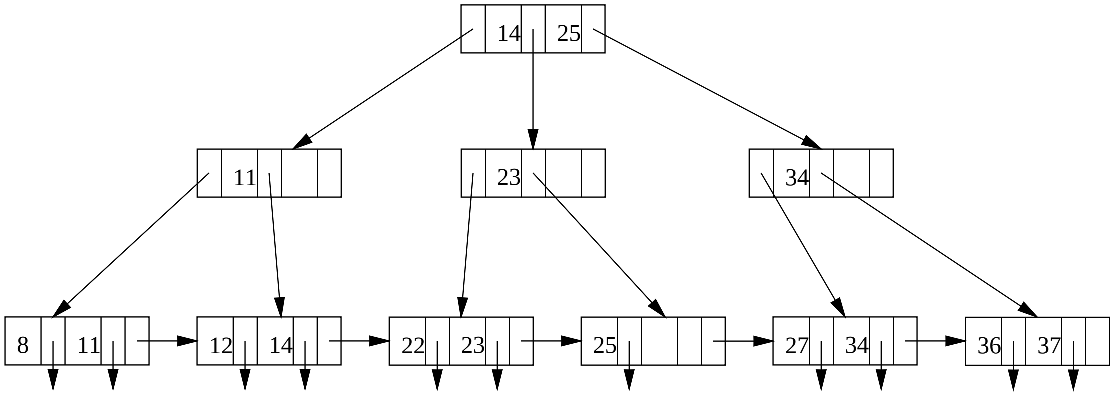

Figure 1.3: The B$^+$tree after 23 is inserted

1.3.3 Hashing

Hashing is another frequently used method for providing fast access to desired tuples in a relation. The idea is to build a *hash table* that contains an index entry for each tuple in the relation and to use a *hash function h( )* to quickly identify each entry in the hash table. The hash table consists of many *buckets*. Usually, the number of *buckets* to be used is allocated in advance based on the number of tuples in the relation. Each bucket corresponds to one to more disk pages. Suppose A is the attribute of a relation which is to be used to provide the fast access. The contents of each bucket are a number of index entries of the form (a, P), where a is the A value of some tuple and P is a *tuple pointer* pointing to the tuple on disk. A hash function $h()$ is used to map each value of A in the relation to a number, known as the *bucket number*. Buckets are usually numbered from 0 to n−1 if n is the number of buckets used. The hash function is designed in such a way that it maps any valid value of A to an integer between 0 and n−1. For example, if A takes non-negative integers, then $h(x) = x \ mod \ n$, which returns the remainder of x divided by n, is such a function. Let t be a tuple and a be the A value of t. If $h(a) = k, 0 \leq k \leq n-1$, then entry (a, P) is placed in the k-th bucket of the hash table. After an entry is created and placed in appropriate bucket for every tuple of the relation, the build of the hash table is complete.

To find the tuple(s) that satisfy the condition "A = a", the same hash function used to build the hash table is used to map a to a bucket, say B. Note that it is possible for different A-values to be mapped to the same bucket. Therefore, the entries in B must be searched to find those whose A-values are equal to a. For each correct entry found, the page that contains the corresponding tuple can be read in directly.

Hashing is considered to be the fastest method for finding tuples with a given A values. However, hashing does have several drawbacks. First, it is possible that too many A-values are mapped to the same bucket and it can not hold all the entries. This is known as *bucket overflow*. A carefully chosen hash function can relieve the problem by distributing A-values into buckets reasonably evenly. However, choosing a good hash function alone may not avoid the problem entirely. A common solution to bucket overflow problem is to place overflow entries into *overflow buckets* and link these overflow buckets to regular buckets. However, each overflow bucket in a linked list implies an additional page I/O. This slows down the access. Second, hashing is effective only when equality conditions are involved. For example, hashing is not very useful if the search condition is "A > a". Although hash functions that preserve the order of tuples exist, these functions are usually poor in distributing A-values evenly. Third, since the space of the hast table is allocated in advance, the hash table may become under utilized if too many tuples have been deleted from the relation or overflow if too many tuples have been inserted to the relation. *Dynamic hashing* and *extendible hashing* techniques can be used to overcome this problem.

1.4 Single Operation Processing

As mentioned in Section 1.2, an execution plan consists of a sequence of operations and a strategy for evaluating each operation. In this section, we discuss techniques for evaluating several relational operations — selection, projection and join. We also provide cost analysis for each strategy.

Let R and S be two relations under consideration. Let n and m be the numbers of tuples in R and S, respectively. Let N and M be the sizes of R and S in pages, respectively. We consider two types of costs in our analysis, namely, I/O cost and CPU cost. Two different methods have been used in the literature for estimating I/O cost. The first uses the total number of pages read or wrote. This method has the drawback of not differentiating random pages and sequential pages. The second method uses the number of I/O operations initiated. One I/O operation may read/write many pages. The number of I/O operations and the number of random pages are about the same. Therefore, this method essentially ignores sequential pages. The rationale is that since sequential pages are much cheaper than random pages, they can be ignored. However, if the buffer size is very large and a large number of pages can be read in per each I/O operation, then this method may also be inaccurate. In addition, in a multi-user environment, it is usually difficult to separate different I/O operations. For example, consider X pages of data that are stored in consecutive storage space. The X pages may be read in by one I/O operation when it is the only I/O request and the buffer is large enough to hold them. However, the same X pages may be read in using several I/O operations when there are concurrent I/O requests. This can also lead to inaccuracy by the second method. Our analysis will be based on the first method unless using the second method has a significant impact on the processing strategy, in the latter case, both methods will be used. A simplification we use in the analysis of the I/O cost is to ignore the cost of outputting the result of an evaluation. This is because the goal of analyzing the cost is to find out the most efficient strategy. Since the size of the result of evaluating an operation is the same regardless of the strategies used, ignoring the cost of outputting the result does not affect the finding of the most efficient strategy. For CPU cost, the number of comparisons needed and/or the number of tuples searched is used. The total cost of evaluating an operation or an execution plan is a weighted sum of the I/O cost and the CPU cost. I/O cost is usually the dominating cost.

1.4.1 Evaluating Selection

Suppose the selection under consideration is $\sigma_{A\ op\ a}$(R), where A is a single attribute, a is a constant, *op* is one of $=, \neq, <, \leq, >, \geq$. If *op* is $\neq$, then most tuples of R are likely to satisfy the condition. In this case, sequentially scanning the entire relation is likely to be a better way to evaluate the condition than using any index structure. The following discussion assumes that *op* is not $\neq$.

The cost of evaluating a selection is closely related with the number of tuples in R that satisfy the condition.

Definition 1.1 *The* selectivity *of "A op a" on R, denoted as $S_{A\ op\ a}(R)$, is the percentage of the tuples of R that satisfy "A op a".*

Clearly, $0 \leq S_{A\ op\ a}(R) \leq 1$. The selectivity factor for any condition can be estimated based on some known statistics about the relation and some assumption about the data distribution of the A values in R. In addition to the number of tuples n in R, these statistics include *dist(A)*, the number of distinct values of A in R, *min(A)*, the smallest value of A in R, and *max(A)*, the largest value of A in R. The values of A are usually assumed to satisfy the uniform distribution between min(A) and max(A). For example, based these assumptions, $S_{A=a}(R) = \frac{1}{dist(A)}$ and $S_{A>a}(R) = \frac{max(A)-a}{max(A)-min(A)}$.

Let k be the number of tuples in R that satisfy the condition "A op a". Then k is estimated to be $n * S_{A\ op\ a}$(R). The cost of evaluating $\sigma_{A\ op\ a}$(R) can be analyzed as follows.

Case 1: Fast access path is not available or not used.

Subcase 1.1: Tuples are stored in sorted A-values. In this case, binary search can be used. The CPU cost is O($\log n + k$) and the I/O cost is O($\log N + \lceil \frac{k}{n} * N \rceil$), where $\lceil \frac{k}{n} * N \rceil$ is the number of pages needed to hold the k tuples satisfying the selection condition.

Subcase 1.2: Tuples are not stored in sorted A-values. In this case, a sequential scan of all tuples is needed. The CPU cost is O(n) and the I/O cost is O(N).

Case 2: Fast access path is used.

Subcase 2.1: Tuples are stored in sorted A-values. This is the case when the fast access path is a primary or clustering index. Since all qualified tuples are stored together and it takes constant number of steps to find the first qualified tuple using the fast access path, the CPU cost is O(k) and the I/O cost is O($\lceil \frac{k}{n} * N \rceil$).

Subcase 2.2: Tuples are not stored in sorted A-values. This is the case when the fast access path is a secondary index. Since each qualified tuple can be obtained in constant number of comparisons using the index, the CPU cost is O(k). Similarly, since each qualified tuple can be obtained by fetching a constant number of pages, the I/O cost is bounded by O(k). At the same time, we never need to read in more than N pages. Therefore, the I/O cost is bounded by $\min\{O(k), N\}$.

1.4.2 Evaluating Projection

Suppose the projection under consideration is $\pi_{A_1,\ldots,A_t}$(R), where $A_1, \ldots, A_t$ are attributes of relation R. Two cases may occur.

Case 1: Duplicate rows are not removed. In SQL, duplicate rows are removed only when *select distinct* is used. In this case, the projection can be evaluated by scanning each tuple once. Therefore, the CPU cost is O(n) and the I/O cost is O(N).

Case 2: Duplicate rows are removed. This is usually accomplished in three steps. First, the relation is scanned and a projection with duplicate is performed. Second, the result of the first step is sorted. Third, the sorted result is scanned for duplication removal. The CPU cost is dominated by the sorting and the cost is O($n \log n$). The I/O cost is dominated by the first two steps. The I/O cost for the first step is O(N). The second typically requires an *external sort*. Let W be the size of the result of the first step ($W = n * (\sum_{i=1}^{t} length(A_i))$/PageSize). Then the I/O cost for the second step is O($W * \log W$).

1.4.3 Evaluating Join

Among the three most frequently used relational operations (i.e., selection, projection and join), join is the most expensive operation. Consequently, it has been studied most extensively. In this subsection, we describe several well known algorithms for evaluating the join operation. Only equijoin will be considered. Suppose the join under consideration is R $\bowtie_{R.A=S.B}$ S. Recall that n and m are the numbers of tuples in R and S, respectively, and N and M are the sizes of R and S in pages, respectively. Without the loss of generality, we assume that S is the smaller of the two relations (i.e., $M \leq N$).

Nested Loop

This algorithm compares every tuple of R with every tuple of S directly for finding the matching tuples. This is illustrated below.

```
for each tuple x in R
    for each tuple y in S
        if x[A] = y[B] then return (x, y)
```

In the above implementation, R is used in the outer loop and is therefore called the *outer relation*. S is used in the inner loop and is the *inner relation*. An alternative is to use S as the outer relation and R as the inner relation. As we will see shortly, the choice of which relation should be used as the outer/inner relation is significant for optimizing the I/O cost. However, it has little impact on the CPU cost. In fact, the CPU cost for this algorithm is always O($n * m$).

To estimate the I/O cost, the above algorithm needs to be modified to page-based. Let K be the size (in pages) of the available memory buffer for the join. Clearly, $K \geq 3$ is required since the buffer needs to hold at least one page for each of the two relations and one page for accumulating the result. In the following analysis, K is used to denote only the buffer pages available for the two join relations (i.e., output buffer page(s) are excluded from K).

We consider a special case first. In this case, K = 2. When R is the outer relation and S is the inner relation, we have the following modified algorithm.

```
for each page P of R
    for each page Q of S
        for each tuple x in P
            for each tuple y in Q
                if x[A] = y[B] then return (x, y)
```

This algorithm scans the inner relation once for each page in the outer relation. An improvement can be made by alternating the scanning order of the inner relation. That is, if the inner relation is scanned from the first page to the last page for the current iteration, then it will be scanned from the last page to the first page for the next iteration. As a result, the last page of S which has just been compared against the current page of R becomes the first page of S to be compared against the next page of R. In this way, this page of S is not re-read into the main memory, saving one page I/O. This technique is known as *rocking-scan*. With the improved algorithm, the I/O cost is $N + M + (N - 1) * (M - 1) = N * M + 1$ as relation R will be scanned once; for the first page of R, the entire S is scanned once; for each of the N–1 remaining pages of R, M–1 pages of S are read in due to rocking-scan. For this special case, it can be seen that if S is the outer relation and R is the inner relation, the I/O cost will still be $N * M + 1$.

We now consider the general case when K is any positive integer greater than or equal to 2. Without the loss of generality, we assume $K < N + M$ since otherwise both R and S can be held in the buffer in their entirety and the I/O cost will be $M + N$. Suppose R uses K1 buffer pages and S uses K2 buffer pages, where K1 and K2 are two positive integers satisfying K1 + K2 = K, K1 $\leq$ N and K2 $\leq$ M. When R is the outer relation and S is the inner relation, the following algorithm is obtained.

```
for each K1 pages P of R
        for each K2 pages Q of S
                for each tuple x in P
                        for each tuple y in Q
                                if x[A] = y[B] then return (x, y)
```

Since R will be read only once, for the first K1 pages of R, the entire S needs to be read, and for each subsequent K1 pages of R (there are $(\lceil\frac{N}{K1}\rceil - 1)$ such K1 pages of R), only M–K2 pages of S need to be read (rocking-scan is used), the I/O cost is

$$N + M + (\lceil\frac{N}{K1}\rceil - 1) * (M - K2) \tag{1.1}$$

It can be shown that expression (1.1) reaches the minimum when K1 = $\min\{N, K-1\}$. Namely, the outer relation should use as many buffer pages as needed (need no more than N buffer pages). Using similar analysis as above, it can be shown that when S is the outer relation, the I/O cost is

$$M + N + (\lceil\frac{M}{K2}\rceil - 1) * (N - K1) \tag{1.2}$$

and this expression reaches the minimum when K2 = $\min\{M, K-1\}$. It can further be shown that when the ceiling is ignored, the minimum reached by expression (1.1) is greater than or equal to the minimum reached by expression (1.2). For example, consider the case when N = 20, M = 10 and K = 15. In this case, expression (1.1) reaches the minimum when K1 = 14 and the cost is approximately (with the ceiling ignored) 34, and expression (1.2) reaches the minimum when K2 = 10 and the cost is 30. Note that when the ceiling is kept, then the result is not always true. For example, when N = 20, M = 10 and K = 5, the minimum reached by expression (1.1) is 66 while the minimum reached by expression (1.2) is 68. But the minimum reached by expression (1.1) is much more likely to be greater than the minimum reached by expression (1.2).

The above analysis is based on the number of I/O pages being the I/O cost metric. The optimization can be summarize as follows. If the optimization criterion is to minimize the number of I/O pages, then use the smaller relation as the outer relation and let the outer relation use as many buffer pages as needed (i.e., $\min\{M, K-1\}$).

However, if the optimization criterion is to minimize the number of I/O operations initiated, then a different buffer allocation strategy should be used. This is discussed below. Note that each I/O operation can read in K1 pages of R or K2 pages of S. Therefore, R can be read in with $\lceil\frac{N}{K1}\rceil$ I/O operations. Similarly, S can be read in with $\lceil\frac{M}{K2}\rceil$ I/O operations. When R is used as the outer relation, R needs to be read in once; for the first K1 pages of R, S needs to be scanned once; for each of the $(\lceil\frac{N}{K1}\rceil - 1)$ K1 pages of R, one I/O operation can be saved when reading in S due to the rocking-scan scheme. Therefore, when R is used as the outer relation, the number of I/O operations needed is

$$\lceil \frac{N}{K1} \rceil + \lceil \frac{M}{K2} \rceil + (\lceil \frac{N}{K1} \rceil - 1) * (\lceil \frac{M}{K2} \rceil - 1) = \lceil \frac{N}{K1} \rceil * \lceil \frac{M}{K2} \rceil + 1 \tag{1.3}$$

It can be shown that when the ceilings are ignored, the above expression reaches the minimum when K1 = K2 = K/2. In other words, to minimize the number of I/O operations, R and S should use the same number of buffer pages. From expression (1.3), it is easy to see that with this optimization criterion, it does not matter which of R and S is used as the outer relation.

Example 1.4 *When N = 20, M = 10 and K = 6, the number of I/O pages is minimized at 49 when S is used as the outer relation and allocated with 5 buffer pages. However, with such a buffer allocation, the number of I/O operations needed is 41. When both R and S are allocated with 3 buffer pages, the number of I/O operations is 29. Interestingly, when R is allocated with 4 buffer pages and S is allocated with 2 buffer pages, the number of I/O operations is minimized at 26. The reason that the number of I/O operations is not minimized when R and S are allocated with the same number of buffer pages is due to the effect of the ceilings in expression (1.3).* ■

Sort Merge

This join algorithm consists of the following two steps.

1. Sort the two relations in ascending order of their respective join attributes, i.e., sort R on A and sort S on B if they are not already sorted.

2. Perform a merge join. We first consider the case when the values under at least one join attribute are distinct. This is a frequently encountered case since most joins are between attributes that have the key/foreign-key relationships and the values under the attribute corresponding to the key are distinct. Without the loss of generality, we assume that A is a key of R. Initially, two pointers are used to point to the two tuples of the two relations which have the smallest values of the two join attributes. If the two values are the same, then first concatenate the two corresponding tuples to produce a result and then move the pointer pointing to a tuple of S one position lower to the next tuple of S. If the two values are different, then the pointer pointing to the tuple with smaller value is moved down one position to the next tuple. This process is repeated until all values under the two attributes are exhausted.

   We now consider the case when both attributes have repeating values. In this case, modification must be made to the above procedure to ensure that all equal values under the two attributes are exhaustively matched. This is described as follows. Suppose the pointer to tuples of R points to a tuple a1 and the pointer to tuples of S points a tuple b1. If the two tuples have the same value, say x, under A and B, advance both pointers until each pointer points to a tuple whose value under the join attribute is different from x. Let the corresponding tuples be a2 from R and b2 from S, respectively. Then a Cartesian product between the set of tuples from a1 to a2 (excluding a2) and the set of tuples from b1 to b2 (excluding b2) is taken.

The cost of this algorithm depends heavily on whether one or both relations have already been sorted on the join attribute and on how many repeating values appear under both join attributes. In the best case scenario, both relations are sorted and there is no repeating values under at least one join attribute. In this case, only step 2 is needed and one scan of each relation is sufficient to perform the merge join. Therefore, in this case, the CPU cost is O($n + m$) and the I/O cost is O($N + M$). In the worst case scenario, both relations need to be sorted and nearly all values under the two join attributes are the same (i.e., almost a full Cartesian product between the two relations is needed). In this case, the CPU cost is O($n \log n + m \log m + n * m$) and the I/O cost is O($N \log N + M \log M + C(R, S)$), where $C(R, S)$ is the cost of performing the Cartesian product between R and S. When the cost of outputting result is not included, $C(R, S)$ is the same as the I/O cost for performing a join between R and S using the nested-loop algorithm. The most frequently encountered scenario is when one relation is already sorted and there is no repeating values under at least one join attribute. This is the case when the join is between two attributes that have the key/foreign-key relationship and the relation with key has a primary index on the key. In this case, the cost (both CPU cost and I/O cost) is the sum of the cost of sorting the relation with the foreign key and the cost of scanning each relation once to merge them.

Hash Join

The basic hash join algorithm consists of the following two steps.

1. Build a hash table for the smaller relation S based on the join attribute. This process is similar to the hashing process described in Section 1.3. However, in this case, rather than putting the pointers to tuples in the buckets, the tuples themselves are placed in the buckets.

2. Use the larger relation R to probe the hash table to perform the join. The probe process is described below.

```
for each tuple x in R {
        hash on the join attribute using the same hash function used in step 1
          to find a bucket in the hash table;
        if the bucket is non-empty
              for every tuple y in the found bucket
                    if x[A] = y[B] then return (x, y)
}
```

One potential problem of hash join is that the hash table is too large to fit the memory buffer available. Building the hash table for the smaller relation helps relieving the problem. More sophisticated solutions to this problem will be discussed in Chapter 3 in the context of performing joins in parallel. Using the same hash function in both steps is essential for hash join as it guarantees that tuples with the same join attribute values from the two relations will be mapped to the same bucket. However, tuples mapped to the same bucket may have different values on the join attribute. Therefore, comparing with every entry in the found bucket is needed.

The tuples of the smaller relation are gone through once for building the hash table. For each tuple t of the large relation, every tuple in the bucket to which t is mapped needs to be examined for entries matching with t. Therefore, the CPU cost is $O(m + n * b)$, where b is the average number of tuples per bucket. If the hash table can be held in the memory, then each relation only needs to be read in once. Therefore, the I/O cost of the algorithm in this case is $O(N + M)$.

Comparison of the Join Algorithms

1. Hash join is a very efficient join algorithm when it is applicable and when the hash table can be held in the main memory. Hash join is only applicable to equijoin.

2. Sort-merge join performs better than nested-loop when both operand relations are large. This is especially true if one or both relations are already sorted on the join attributes. When both input relations are already sorted, sort-merge join is as good as hash join. Sort-merge join is also less sensitive to the size of the memory buffer available for performing the join than other algorithms.

   When there are secondary indexes on both join attributes of the two relations, a variation of the sort-merge join exists and it can be described as follows. Note that the values of each join attribute are sorted in the leaf nodes of the corresponding index. Therefore, a merge can be performed based on the sorted values of the two join attributes. When a match is found between the two values currently under consideration, the corresponding pointers in the leaf nodes are used to fetch the tuples for join. While this variation of sort-merge join can save the cost of sorting the two join attributes, it may incur two or more page I/Os for each matching values of the join attributes. This variation can be very effective if the relations are large but the join result is small (i.e., very few values of the join attributes match) since only matched values incur additional I/O cost.

3. Nested-loop join performs well when one relation is large and one relation is small. A special case is when the smaller relation can be entirely held in the main memory. In this case, both relations only need to be read in once.

   When nested-loop join is combined with the index on the join attribute of the inner relation, excellent performance can yield. With an index on the join attribute of the inner relation, for each tuple in the outer relation, the index can be utilized to find the matching tuples in the inner relation quickly. This combination is most effective when one joining relation is very large and the other is very small, and the larger relation has an index on the join attribute. For example, when S has 100 tuples and R has 1 million tuples, both the number of comparisons and the number of pages read are limited by a few hundreds if the said combination is used. Clearly, no other algorithm can deliver any better performance, not even close.

1.5 Determining the Execution Order of Operations

A typical database query consists of a number of operations. In which order these operations should be evaluated has significant impact on the cost of evaluating the query. In this section, we discuss techniques and problems on determining an optimal execution order of operations in a query.

There are primarily two approaches for determining the execution order of operations, i.e., the algebra based approach and the cost estimation based approach. Both approaches use a set of rules that can transform one execution plan (represented as a relational algebra expression) to other equivalent execution plans. We first describe the transformation rules in Section 1.5.1. We then discuss the two approaches in Sections 1.5.2 and 1.5.3, respectively.

1.5.1 Transformation Rules

There are numerous rules that can transform one relational algebra expression to other equivalent expressions. Two expressions are equivalent if they always produce the same result. In this subsection, we only present a few of these transformation rules.

Let R, S and T be three relations.

1. Cascade of selections: Let C1 and C2 be two selection conditions on R. Then

   $\sigma_{C1\ and\ C2}(R) = \sigma_{C1}(\sigma_{C2}(R))$

2. Commuting selection with join: If condition C involves only attributes of R, then

   $\sigma_C(R \bowtie S) = (\sigma_C(R)) \bowtie S$

   From rules (1) and (2), the following rule can be deduced:

   If condition C1 involves only attributes of R and condition C2 involves only attributes of S, then

   $\sigma_{C1\ and\ C2}(R \bowtie S) = (\sigma_{C1}(R)) \bowtie (\sigma_{C2}(S))$

   Note that this rule as well as the next two rules also apply to Cartesian product. That is, if $\bowtie$ is replaced by $\times$, these rules are still true.

3. Commuting projection with join: Assume AL = $\{A_1, ..., A_n, B_1, ..., B_m\}$, where A's are attributes from R and B's are attributes from S.

   (i) If the join condition C involves only attributes in AL, then

   $\pi_{AL}(R \bowtie_C S) = (\pi_{A_1,...,A_n}(R)) \bowtie_C (\pi_{B_1,...,B_m}(S))$

   (ii) If, in addition to attributes in AL, C also involves attributes $A'_1, ..., A'_u$ from R and attributes $B'_1, ..., B'_v$ from S, then

   $\pi_{AL}(R \bowtie_C S) = \pi_{AL}((\pi_{A_1,...,A_n,A'_1,...,A'_u}(R)) \bowtie_C (\pi_{B_1,...,B_m,B'_1,...,B'_v}(S)))$

4. Associativity of θ-join and natural join:

   $R \bowtie_{C1} (S \bowtie_{C2} T) = (R \bowtie_{C1} S) \bowtie_{C2} T$

   $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

   The reason to list the rules for θ-join and natural join separately is because mixing them in the same rule yields an incorrect rule. In other words, the following rule is incorrect:

   $R \bowtie_C (S \bowtie T) = (R \bowtie_C S) \bowtie T$

   As an example, suppose R has attributes (A, B), S has attributes (D, E), T has attributes (A, E), and the join condition C is "R.A = S.D". In this case, the natural join on the left hand side of the above rule will evaluate only "S.E = T.E" while the natural join on the right hand side of the above rule will evaluate "S.E = T.E" and "R.A = T.A".

5. Replacing $\times$ by $\bowtie$ and σ: If C is a selection condition of the form "R.A op S.B" or the conjunction of the form, then

   $\sigma_C(R \times S) = R \bowtie_C S$

1.5.2 Algebra Based Optimization

The basic idea of the algebra based optimization approach is to first represent each relational query in relational algebra and then transform it to an equivalent but more efficient relational algebra expression. The transformation is guided by a set of heuristic optimization rules. The following four rules are commonly used.

Rule 1: Perform selections as early as possible. This can be achieved using the transformation rules 1 and 2 ($\bowtie$ may be replaced by $\times$) presented in the last section.

Rule 2: Replace Cartesian products by joins whenever possible. This can be achieved using the transformation rule 5.

Rule 3: If there are several selections or joins, perform the most *restrictive* selections or joins first. A selection or a join is more restrictive than another selection or join if it yields a smaller result. A determination needs to be made on which selection or join is most restrictive based on selectivities and other statistical information. Algebraically, the transformation can be achieved using the transformation rules 1 and 4.

Rule 4: Project out useless attributes early. If an attribute of a relation is not needed for future operations, then it should be removed so that smaller input relations can be used by future operations. This can be achieved using the transformation rule 3.

The above heuristic rules can be illustrated graphically using the concept of *query tree* which is essentially a tree representation of a relational algebra expression. In a query tree, each input relation is represented as a leaf node and each operation is represented as an internal node. The tree also has a bottom-up execution semantics, that is an operation can be evaluated only if all its descendant operations have been evaluated.

Example 1.5 *Consider the following SQL query involving three relations Student(SSN, Name, Age, GPA, Address), Course(Course#, Title, Credit) and Take(SSN, Course#, Grade).*

```
select Name
from Student, Take, Course
where GPA > 3.5 and Title = 'Database System'
      and Student.SSN = Take.SSN and Take.Course# = Course.Course#
```

This query can be directly transformed to the following unoptimized relational algebra expression

$$\pi_{Name}(\sigma_{GPA>3.5 \text{ and } Title='Database\ System' \text{ and } Student.SSN=Take.SSN \text{ and } Take.Course\#=Course.Course\#}(Student \times Take \times Course))$$

Figure 1.4 (a) shows the query tree of the above relational algebra expression. First, optimization rule 1 is followed to push the selection conditions as far as possible down the tree. The four conditions are pushed down separately. For instance, the condition "Student.SSN = Take.SSN" is pushed over (commuted with) the first Cartesian product since it involves only attributes in the left child branch of this Cartesian product. The result of step is shown in Figure 1.4 (b). Next, optimization rule 2 is followed and the two Cartesian products together with the selection conditions immediately above them are replaced by two joins. Incidentally, natural joins can be used since for each join the join attribute is the only common attribute between the child branches. Figure 1.4 (c) shows the result of this step. There are two joins in the tree. Optimization rule 3 says that the more restrictive join should be performed first. Clearly, the join "Take.Course# = Course.Course#" is more restrictive than the join "Student.SSN = Take.SSN" since the number of students satisfying the condition "GPA > 3.5" is likely to be much greater than the number of courses satisfying the condition "Title = 'Database System' ". Therefore, the join "Take.Course# = Course.Course#"

should be performed first. The new query tree after the change is shown in Figure 1.4 (d). Finally, we apply the optimization rule 4 and see if any useless attributes can be projected out early. Consider the attributes from Course. After the selection, only Course# is needed for a later join. Therefore, other attributes can be projected out. Other relations can be considered similarly. Figure 1.4 (e) shows the final tree generated by the heuristic optimization rules.

In Figure 1.4 (e), selections are performed before projections. But in practice, projections and selections on the same relation can usually be performed using the same scan of the relation. The actual execution plan generated from Figure 1.4 (e) could have the following steps:

1. *Perform* $\pi_{Course\#}(\sigma_{Title='Database\ System'}(Course))$. *Let T1 denote the result.*
2. *Perform* $\pi_{SSN,Course\#}(Take)$. *Let T2 denote the result.*
3. *Perform T1* $\bowtie$ *T3. Let T3 denote the result.*
4. *Perform* $\pi_{SSN,Name}(\sigma_{GPA>3.5}(Student))$. *Let T4 denote the result.*
5. *Perform* $\pi_{Name}(T3 \bowtie T4)$.

■

While the algebra based heuristic optimization approach can work reasonably well for simple queries in most cases, it may result in very bad execution plans sometimes.

1.5.3 Cost Estimation Based Optimization

The basic idea of this approach can be described as follows. For each query, enumerate all possible execution plans. For each execution plan, estimate the cost of the execution plan. Finally, choose the execution plan with the lowest estimated cost.

Clearly, if cost of every execution plan can be estimated accurately, then an optimal plan can eventually be found. This is the main appeal of this approach. There are primarily two difficulties to implement this approach as explained below:

1. There may be too many possible execution plans to enumerate. In general, the number of possible execution plans is an exponential function of the number of relations referenced in a query.
2. It may be difficult to estimate the cost of each execution plan accurately. In Section 1.4, we illustrated how to estimate the cost (CPU cost and I/O cost) for several operations. In a more complex execution plan, the result of an operation, say Op1, may be used as input to another operation, say Op2. To estimate the cost of Op2, it is necessary to estimate the size of the result of Op1. For example, to estimate the cost of (R $\bowtie$ S) $\bowtie$ T, we need: (a) Estimate the cost of (R $\bowtie$ S). Let W denote the result of (R $\bowtie$ S). (b) Estimate the size of

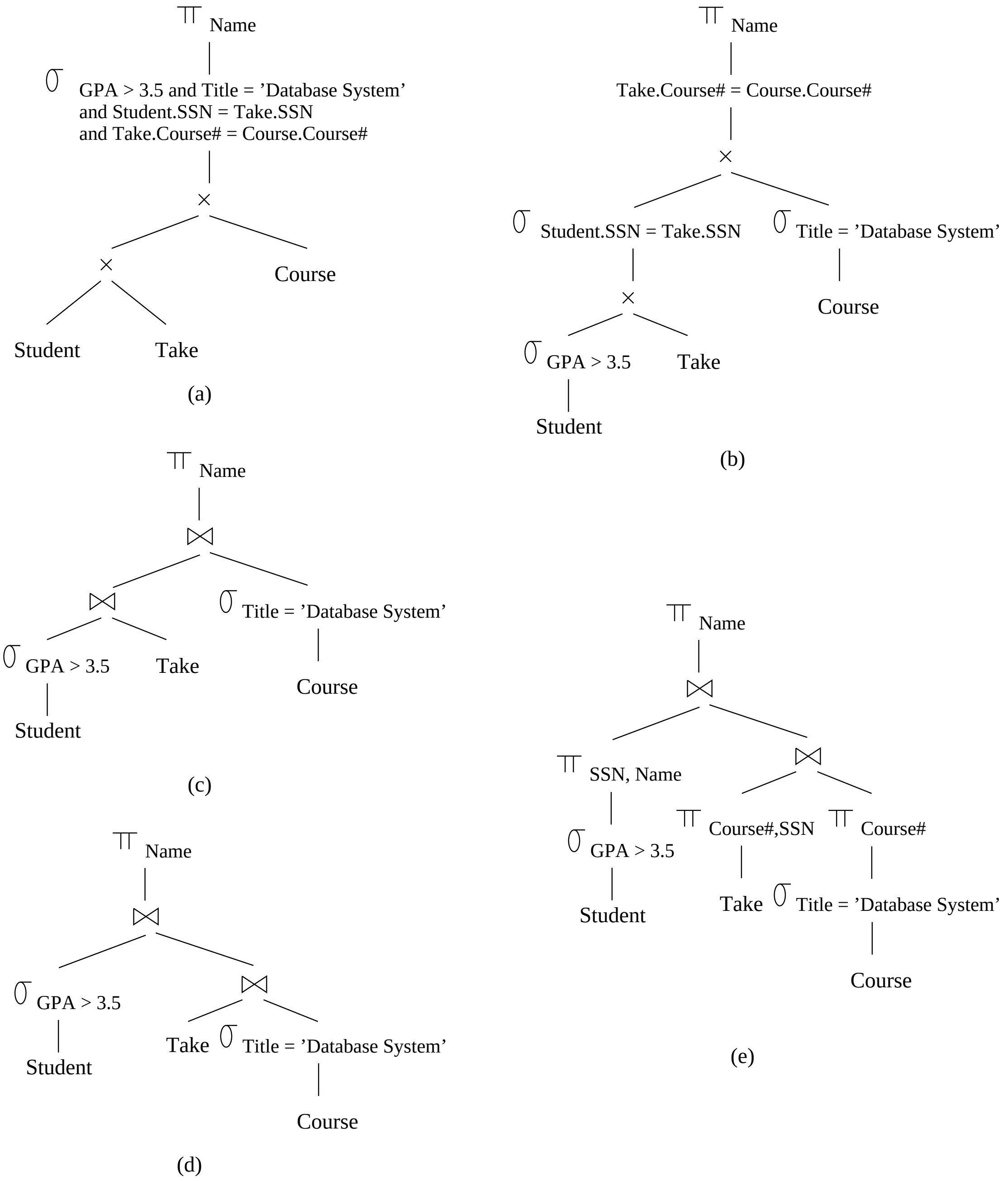

Figure 1.4: Illustrating the heuristic optimization rules

W. (c) Estimate the cost of (W $\bowtie$ T). The estimated total cost is the sum of the estimated costs obtained in steps (a) and (c). The most difficult part in accurately estimating the cost of an execution plan is to accurately estimate the sizes of intermediate results.

As mentioned in Section 1.2, commonly used approaches for dealing with the first difficulty include (1) consider special type of queries such as chain queries, star queries and tree queries, and (2) enumerate only a subset of all possible execution plans such as considering only left-deep tree plans. An example left-deep tree plan for joining five relations R_1, R_2, R_3, R_4 and R_5 is (((($R_1 \bowtie R_2$) $\bowtie R_3$) $\bowtie R_4$) $\bowtie R_5$). It is so named because its query tree goes left deep. Some of the issues will be revisited in more detail in later chapters of the book. More discussions can also be found from the references listed at the end of the chapter.

For the rest of this section, we briefly discuss techniques for estimating the result size of the three operations — selection, projection and join.

Selection σ_C(R): Let S_C(R) denote the selectivity of the condition C on the relation R. Then the number of tuples in σ_C(R) is estimated to be $n * S_C$(R), where n is the number of tuples in R, and the size of σ_C(R) in pages is estimated to be $N * S_C$(R), where N is the size of R in page.

Clearly, the key to ensure that the above estimation is accurate is to have an accurate estimation of the selectivity. When the condition C is a *simple condition* of the form "A op a", the selectivity depends largely on the distribution of the values of A in R. A common assumption is that the distribution is uniform. In this case, selectivities can be derived straightforwardly. If the actual distribution deviates from the assumed distribution severely, then the estimation will not be accurate. Sometimes, the selectivity can also be derived from the characteristics of the attribute A. For example, if A is a key, then $S_{A=a}$(R) can be estimated as 1/n. When C is a conjunction or disjunction of several simple conditions, then the selectivity also depends on the dependencies among the involved attributes. For example, if the condition is "A op a and B op b" and the two attributes A and B are independent, then the selectivity can be estimated as $S_{A\ op\ a}$(R)*$S_{B\ op\ b}$(R).

Projection $\pi_{A_1,...,A_t}$(R): We consider the case when duplicates are removed. In general, it is extremely difficult to estimate the result size of a projection. Several special cases exist in which the estimation can be done reasonably accurately. The first case is when $\{A_1, ..., A_t\}$ forms a superkey. In this case, no duplicate is possible. Therefore, the number of tuples in $\pi_{A_1,...,A_t}$(R) is still n and the size of $\pi_{A_1,...,A_t}$(R) in pages is $\frac{n*(\sum_{i=1}^{t} length(A_i))}{PageSize}$, where $length(A_i)$ denotes the number of bytes of the values of A_i. The second case if when $t = 1$, i.e., the result contains only one attribute. In this case, the number of tuples in the result is the number of distinct A-values in R and the size of the result in pages is $\frac{dist(A_1)*length(A_1)}{PageSize}$. The third case is when one of the attributes, say A_j, $1 \leq j \leq t$, *functionally determines* other attributes, that is, for any two tuples in R, if they have the same value under A_j, then they also have the same value under each of the other attributes. In this case, the number of tuples in the result is $dist(A_j)$ and the size of the result in pages is $\frac{dist(A_j)*(\sum_{i=1}^{t} length(A_i))}{PageSize}$. The fourth case is when all A_i's are independent. In this case, the number of tuples in the result can be estimated easily. For example, when $t = 2$, the number is $n * (\frac{dist(A_1)}{n} + \frac{dist(A_2)}{n} - \frac{dist(A_1)}{n} * \frac{dist(A_2)}{n}) = dist(A_1) + dist(A_2) - \frac{dist(A_1)*dist(A_2)}{n}$. After the number of tuples is estimated, the size can also be estimated easily since each tuple in the result has length $\sum_{i=1}^{t} length(A_i)$.

Join R $\bowtie_{R.A\ op\ S.B}$ S: We consider the case when *op* is =. If A and B are arbitrary attributes of R and S, then estimating the number of tuples in the result of the join is very difficult. Fortunately, most joins in practice are between a foreign key and the referenced key. For this most frequently encountered case, the estimation can be reasonably made. Suppose A is the primary key of R and B is a foreign key of S that references A. Then the number of tuples in the result of the join is m — the number of tuples of S if there is no null value under B. This is because every tuple of S will join with exactly one tuple of R. In this case, the size of the result in pages is $\frac{m*(length(R)+length(S))}{PageSize}$. When null values exist in B, m can be used as an approximation for the number of tuples in the result. One complication is that often R and S are not the *base relations*, i.e., R and S could be the intermediate results with selections having been performed. In this case, it can no longer be guaranteed that every tuple of S will match some tuple of R although it is still true that every tuple of S can join with at most one tuple of R. In this case, the number of tuples in the result can be estimated to be $m' * r$, where m' is number of tuples in S after the selection and r is the ratio of the number of the tuples in R after the selection over that in R before the selection.

How to accurately estimate the sizes of intermediate results has been studied extensively in the literature. The methods proposed can be roughly classified into three categories and they are *sampling methods*, which estimate the sizes based on the information collected from a small fraction of instances from the relations, *histogram methods*, which use pre-stored detailed information about the relations to estimate the sizes, and *parametric methods*, which use analytical and/or statistical techniques for size estimation. Parametric methods typically make assumptions about the distribution of data values (e.g., uniform distribution) and about the correlation between the values of different attributes (e.g., independent). These assumptions may be inaccurate, affecting the effectiveness of this type of methods. Nevertheless, parametric methods are still the most commonly used method for estimating the sizes of intermediate results in commercial database systems. Histogram methods need to store and maintain detailed statistics about the data in the database. Maintaining these statistics could be very expensive in terms of both space and computation. Sampling methods do not require any assumptions about data distribution nor the storage of any detailed statistics in advance. Sampling is performed based on the current data instances. Sampling methods have been shown to be quite accurate in estimation. However, the cost of this type of methods is substantially higher than that of parametric methods. Details of these methods can be found from the references provided at the end of the chapter.

Bibliographic Notes

Processing and optimization of relational queries is one of the most researched areas in relational database systems. Most database textbooks have one or more chapters dealing with indexing and query optimization techniques [10, 11, 25, 35, 44, 45]. An earlier survey on query optimization techniques can be found in [21]. A more recent survey can be found in [14].

B^+tree is discussed in [2, 9, 24]. Hashing is extensively discussed in [24]. Extendible hashing is discussed in [12, 29] and dynamic hashing is discussed in [26].

Optimization of relational algebra expressions is discussed in [18, 31, 40]. More detailed presentation and discussion on the transformation rules of relational algebra expressions and the algebra based optimization technique can be found in [44]. [46] discusses a query optimization strategy for the INGRES system. Cost estimation based optimization is discussed in [1, 39, 4]. Nested loop join is first discussed in [36, 37]. Combining nested loop with index on the join attribute of the inner relation is from [15]. Extending the nested loop join to page based is discussed in [22]. Rocking-scan is also from this paper. That allocating equal buffer pages to both operand relations minimizes the number of I/O operations is from [17]. Sort merge join is first discussed in [3]. A recent survey on join processing techniques can be found in [32]. Optimization of SPJ queries is discussed in [47]. The optimization of conjunctive queries is discussed in [5]. This paper also proved that finding an optimal execution plan for a general conjunctive query is NP-complete. Transformation of nested queries to unnested queries is discussed in [23].

There are many studies on estimating the selectivities and the sizes of intermediate results. Sampling techniques can be found in [16, 19, 20, 27, 28]. Histogram method is discussed in [7, 34, 38]. Discussions on parametric methods can be found in [6, 7, 30, 33, 39].

Bibliography

[1] M. Astrahan, et al, System R, A Relational Approach to Data Base Management. ACM TODS, 1:2, June 1976, pp. 97-137.

[2] R. Bayer, and K. Unterauer, Prefix B-trees. ACM TODS, 2:1, March 1977, pp. 11-26.

[3] M. Blasgen, and K. Eswaran, On the Evaluation of Queries in a Relational Database System. IBM System Journal, 16:1, 1977, pp. 363-377.

[4] K. Bratbergsengen, Hashing Methods and Relational Algebra Operations. VLDB, 1984, pp. 323-333.

[5] A. Chandra, and P. Merlin, Optimal Implementation of Conjunctive Queries in Relational Databases. 9th ACM Symposium on the Theory of Computing, pp. 77-90.

[6] S. Christodoulakis, Estimating Record Selectivities. Information System, 8:2, 1983, pp. 105-115.

[7] S. Christodoulakis, Estimating Block Transfers and Join Sizes. ACM SIGMOD, 1983, pp. 40-54.

[8] E. Codd, A Relational Model for Shared Large Data Bank. Communications of the ACM, 13:6, June 1970.

[9] D. Comer, The Ubiquitous B-tree. ACM Computing Surveys, 11:2, June 1979, pp. 121-137.

[10] C. J. Date, An Introduction to Database Systems. Addison-Wesley, 1995.

[11] R. Elmasri, and S. Navathe, Fundamentals of Database Systems. Benjamin/Cummings, 1994.

[12] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, Extendible Hashing — A Fast Access Method for Dynamic Files. ACM TODS, 4:3, September 1979.

[13] L. Gotlieb, Computing Joins of Relations. ACM SIGMOD, 1975.

[14] G. Graefe, Query Evaluation Techniques for Large Databases. ACM Computing Surveys, June 1993.

[15] N. Griffeth, Nonprocedural Query Processing for Databases with Access Paths. ACM SIGMOD, 1978, pp. 160-168.

[16] P. Haas, and A. Swami, Sequential Sampling Procedures for Query Size Estimation. ACM SIGMOD, 1992, pp. 341-350.

[17] R. Hagmann, An Observation on Database Buffering Performance Metrics. VLDB, 1986, pp. 289-293.

[18] P. Hall, Optimization of a Single Relational Expression in a Relational Database. IBM J. of Research and Development, 20:3, 1976, pp. 244-257.

[19] W. Hou, G. Ozsoyoglu, and B. Taneja, Statistical Estimators for Relational Algebra Expressions. ACM PODS, 1988, pp. 276-287.

[20] W. Hou, G. Ozsoyoglu, and E. Dogdu, Error-constrained COUNT Query Evaluation in Relational Databases. ACM SIGMOD, 1991, pp. 278-287.

[21] M. Jarke, and J. Koch, Query Optimization in Database Systems. ACM Computing Surveys, 16, 1984, pp. 111-152.

[22] W. Kim, A New Way to Compute the Product and Join. ACM SIGMOD, 1980, pp. 179-187.

[23] W. Kim, On Optimizing an SQL-like Query. ACM TODS, 3:3, September 1982.

[24] D. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, 1973.

[25] H. Korth, and A. Silberschatz, Database System Concepts. McGraw-Hill, 1991.

[26] P. Larson, Dynamic Hashing. BIT, 18, 1978.

[27] R. Lipton, and J. Naughton, Query Size Estimation by Adaptive Sampling. ACM PODS, 1990, pp. 40-46.

[28] R. Lipton, J. Naughton, and D. Schneider, Practical Selectivity Estimation through Adaptive Sampling. ACM SIGMOD, 1990, pp. 1-11.

[29] W. Litwin, Trie Hashing. ACM SIGMOD, 1981.

[30] C. Lynch, Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values. VLDB, 1988, pp. 240-251.

[31] J. Minker, Search Strategy and Selection Function for an Inferential Relational System. ACM TODS, 3:1, pp. 1-31.

[32] P. Mishra, and M. Eich, Join Processing in Relational databases. ACM Computing Surveys, March 1992.

[33] T. Mostardi, Estimating the Size of Relational SPJ Operation Results: An Analytic Approach. Information System, 15, 1990, pp. 591-601.

[34] M. Muralikrishna, and D. DeWitt, Equi-depth Histograms for Estimating Selectivity Factors for Multi-dimensional Queries. ACM SIGMOD, 1988, pp. 28-36.

[35] P. O'Neil, Data Base: Principles, Programming, Performance. Morgan Kaufmann, 1994.

[36] R. Pecherer, Efficient Evaluation of Expressions in a Relational Algebra. ACM Pacific 75 Conference, 1975, pp. 44-49.

[37] R. Pecherer, Efficient Exploration of Product Spaces. ACM SIGMOD, 1976, pp. 169-177.

[38] G. Piatetsky-Shapiro, and C. Connell, Accurate Estimation of the Number of Tuples Satisfying a Condition. ACM SIGMOD, 1984, pp. 256-276.

[39] P. Selinger, et al, Access Path Selection in a Relational Database Management System. SIGMOD, 1979.

[40] J. Smith, and P. Chang, Optimizing the Performance of a Relational Algebra Interface. CACM, 18:10, October 1975.

[41] M. Stonebraker, *Managing Persistent Objects in a Multi-level Store*. SIGMOD, 1991.

[42] M. Stonebraker, J. Frew, K. Gardels and J. Meredith, *The SEQUOIA 2000 Storage Benchmark*, SIGMOD, 1993.

[43] W. Sun, Y. Ling, N. Rishe, and Y. Deng, An Instant and Accurate Size estimation Method for Joins and Selection in a Retrieval-Intensive Environment. ACM SIGMOD, 1993, pp. 79-88.

[44] J. Ullman, Principles of Database Systems. Pitman Publishing, 1982.

[45] J. Ullman, Database and Knowledge-Base Systems. Vol.1 and Vol 2, Computer Science Press, 1988.

[46] E. Wong, and K. Youssefi, Decomposition — A Strategy for Query Processing. ACM TODS, 1:3, 1976, pp. 223-241.

[47] S. Yao, Optimization of Query Evaluation Algorithms. ACM TODS, 4:2, pp. 133-155.