Data Modelling and Databases - Chapter 15 (Book)

- Author: Ruben Schenk
- Date: 24.05.2021
- Contact: ruben.schenk@inf.ethz.ch

15. Query Execution

The `query processor` is the group of components of a DBMS that turns user queries and data-modification commands into a sequence of database operations and executes those operations.

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter:

1. `Parsing`. A `parse tree` for the query is constructed.
2. `Query Rewrite`. The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query.
3. `Physical Plan Generation`. The abstract query plan from (2), often called a `logical query plan`, is turned into a `physical query plan` by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators.

Parts (2) and (3) are often called the `query optimizer`, and these are the hard parts of the query compilation.

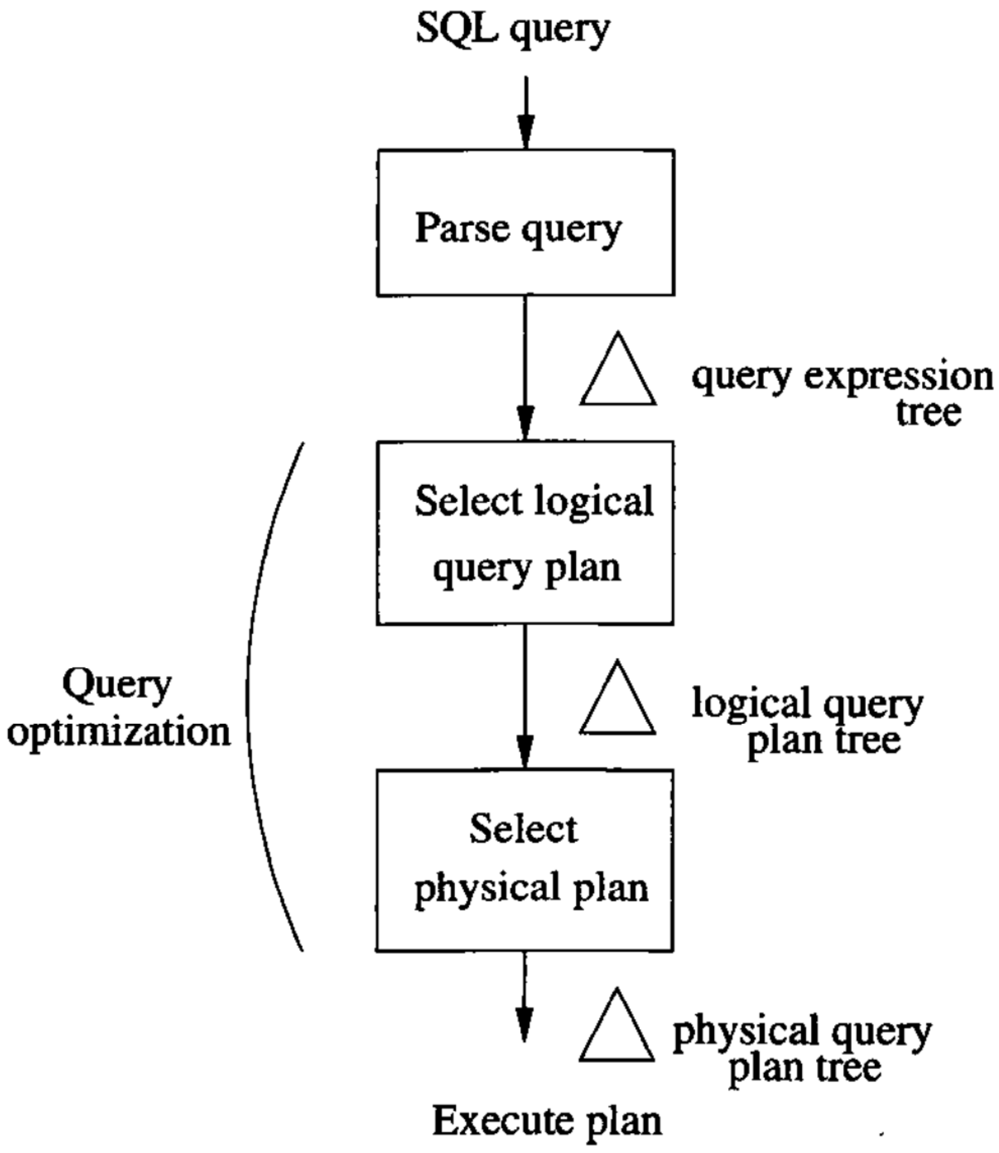


Figure 15.2: Outline of query compilation.

15.1 Introduction to Physical-Query-Plan Operators

15.1.1 Scanning Tables

Perhaps the most basic thing we can do in a physical query plan is to read the entire contents of a relation R. There are two basic approaches to locating the tuples of a relation R:

1. In many cases, the relation R is stored in an area of secondary memory, with its tuples arranged in blocks. The blocks containing the tuples of R are known to the system, and it is possible to get the blocks one by one. This operation is called `table-scan`.
2. If there is an index on any attribute of R, we may be able to use this index to get all the tuples of R. This operation is called `index-scan`.

15.1.2 Sorting While Scanning Tables

The physical-query-plan operator `sort-scan` takes a relation R and a specification of the attributes on which the sort is to be made, and produces R in that sorted order. There are several ways that sort-scan can be implemented, for example if there is a B-tree index on the attribute to be sorted by.

15.1.3 The Computation Model for Physical Operators

We shall use the number of disk I/O's as our measure of cost for an operation. When comparing algorithms for the same operations, we shall make an assumption that may be surprising at first:

- We assume that the arguments of any operator are found on disk, but the result of the operator is left in main memory.

15.1.4 Parameters for Measuring Costs

Now, let us introduce the parameters (sometimes called statistics) that we use to express the cost of an operator.

M will denote the number of main-memory buffers available to an execution of a particular operator.
We shall furthermore make the simplifying assumption that data is accessed one block at a time from disk.
There are three parameter families, B, T, and V:

- When describing the size of a relation R, we most often are concerned with the number of blocks that are needed to hold all the tuples of R. This number of blocks will be denoted $B(R)$. Usually, we assume that R is `clustered`, that is, it is stored in B blocks or in approximately B blocks.
- Sometimes, we also need to know the number of tuples in R, and we denote this quantity by $T(R)$.
- Finally, we shall sometimes want to refer to the number of distinct values that appear in a column of a relation. If R is a relation, and one of its attributes is a, then $V(R,\, a)$ is the number of distinct values of the column for a in R.

15.1.5 I/O Cost for Scan Operators

IF relation R is clustered, then the number of disk I/O's for the table-scan operator is approximately B. Likewise, if R fits in main-memory, we can implement sort-scan by reading R into memory and performing an in-memory sort, again requiring only B disk I/O's.

However, if R is distributed among tuples of other relations, then a table-scan for R may require reading as many blocks as there are tuples of R, that is, the I/O cost is T.

15.1.6 Iterators for Implementation of Physical Operators

Many physical operators can be implemented as an `iterator`, which is a group of three methods that allows a consumer of the result of the physical operator to get the result one tuple at a time. The three methods forming the iterator for an operation are:

1. `Open()`. This method starts the process of getting tuples, but does not get a tuple.
2. `GetNext()`. This method returns the next tuple in the result and adjusts data structures as necessary to allow subsequent tuples to be obtained.
3. `Close()`. This method ends the iteration after all tuples, or all tuples that the consumer wanted, have been obtained.

Example: Perhaps the simplest iterator is the one that implements the table-scan operator. Fig. 15.3 sketches the three methods for this iterator.

```
Open() {
    b := the first block of R;
    t := the first tuple of block b;
}

GetNext() {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block) {
            RETURN NotFound;
        } ELSE /* b is a new block */ {
            t := first tuple on block b;
        }
    } /* now we are ready to return t and increment */
    oldt := t;
    increment t to the next tuple of b;
    RETURN oldt;
}

CLOSE() {

}
```

Figure 15.3: Iterator methods for the table-scan operator over relation R.

Example: Let us consider another example of how iterators can be combined by calling other iterators. The operation is the bag union $R \cup S$, in which we produce first all the tuples of R and then all the tuples of S, without regard for the existence of duplicates. The iterator methods for the union are sketched in Fig. 15.4.

```
Open() {
    R.Open();
    CurRel := R;
}

GetNext() {
    IF (CurRel = R) {
        t := R.GetNext();
        IF (t <> NotFound) /* R is not exhausted */ {
            RETURN T;
        } ELSE /* R is exhausted */ {
            S.Open;
            CurRel = S;
        }
    }
    /* here, we must read from S */
    RETURN S.GetNext();
    /* notice that if S is exhausted, S.GetNext()
    will return NotFound, which is the correct
    action for our GetNext as well */
}

Close() {
    R.Close();
    S.Close();
}
```

Figure 15.4: Building a union iterator from iterators $\mathcal{R}$ and $\mathcal{S}$.

15.2 One-Pass Algorithms

The choice of algorithm for each operator is an essential part of the process of transforming a logical query plan into a physical query plan. While many algorithms for operators have been proposed, they largely fall into three classes:

1. Sorting-based methods
2. Hash-based methods
3. Index-based methods

In addition, we can divide algorithms for operators into three "degrees" of difficulty and cost:

1. Some methods involve reading the data only once from disk. These are the `one-pass` algorithms.
2. Some methods work for data that is too large to fit in available main memory bur not for the largest imaginable data sets. These `two-pass` algorithms are characterized by reading data a first time from disk, processing it in some way, writing it to disk, and then reading it a second time for further processing during the second pass.
3. Some methods work without a limit on the size of the data. These methods use three or more passes to do their jobs, and are natural, recursive generalizations of the two-pass algorithms.

In this section, we shall concentrate on the one-pass methods. HEre and subsequently, we shall classify operators into three broad groups:

1. `Tuple-at-a-time, unary operations`. These operations - selection and projection - do not require an entire relation, or even a large part of it, in memory at once.
2. `Full-relation, unary operations`. These one-argument operations require seeing all or most of the tuples in memory at once, so one-pass algorithms are limited to relations that are approximately of size M or less. The operations of this class are γ (the grouping operator) and δ (the duplicate-elimination operator).
3. `Full-relation, binary operations`. All other operations are in this class: set and bag versions of union, intersection, difference, joins, and products.

15.2.1 One-Pass Algorithms for Tuple-at-a-Time Operations

The tuple-at-a-time operations $\sigma(R)$ and $\pi(R)$ have obvious algorithms, regardless of whether the relation fits in main memory. WE read the blocks of R one at a time into an input buffer, perform the operation on each tuple, and move the selected tuple or the projected tuples to the output buffer.

15.2.2 One-Pass Algorithms for Unary, Full-Relation Operations

Now, let us consider the unary operations that apply to relations as a whole, rather than to one tuple at a time: duplicate elimination (σ) and grouping (γ).

Duplicate Elimination

To eliminate duplicates, we can read each block of R one at a time, but for each tuple we need to make a decision as to whether we have already seen this tuple before or not. To support this decision, we need to keep in memory one copy of every tuple we have seen. One memory buffer holds one block of R's tuples, and the remaining $M-1$ buffers can be used to hold a single copy of every tuple seen so far.
When a new tuple from R is considered, we compare it with all tuples seen so far, and if it is not equal to any of these tuples we both copy it to the output and add it to the in-memory list of tuples we have seen.

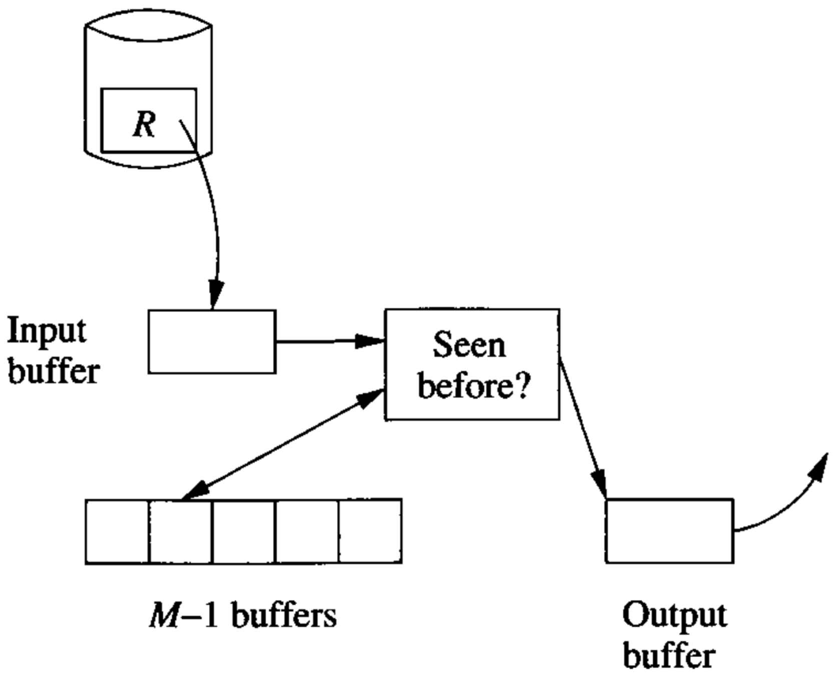


Figure 15.6: Managing memory for a one-pass duplicate-elimination.

Grouping

A grouping operation γ_L gives us zero or more grouping attributes and presumably one or more aggregated attributes. If we create in main memory one entry for each group, then we can scan the tuples of R, one block at a time. The `entry` for a group consists of values for the grouping attributes and an accumulated value or values for each aggregations.
When all tuples of R have been read into the input buffer and contributed to the aggregations for their group, we can produce the output by writing the tuple for each group.

15.2.3 One-Pass Algorithms for Binary Operations

Left out.

15.3 Nested-Loop Joins

15.3.1 Tuple-Based Nested-Loop Join

The simplest variation of nested-loop join has loops that range over individual tuples of the relations involved. In this algorithm, which we call `tuple-based nested-loop join`, we compute the join $R(X, Y) \bowtie S(Y, Z)$ as follows:

```
FOR each tuple s in S DO
    FOR each tuple r in R DO
        IF r and s join to make a tuple t THEN
            output t;
```

If we are careless about how we buffer the blocks of relations R and S, then this algorithm could require as many as $T(R)T(S)$ disk I/O's.

15.3.2 An Iterator for Tuple-Based Nested-Loop Join

One advantage of a nested-loop join is that it fits well into an iterator framework, and thus allows us to avoid storing intermediate relations on disk in some situations.

15.3.3 Block-Based Nested-Loop Join Algorithm

We can improve on the tuple-based nested-loop join of Section 15.3.1 if we compute $R \bowtie S$ by:

1. Organizing access to both argument relations by blocks, and
2. Using as much main memory as we can store tuples belonging to the relation S, the relation of the outer loop.

The algorithm of Fig. 15.8 is sometimes called "nested-block join". We shall continue to call it simply `nested-loop join`, since it is the variant of the nested-loop idea most commonly implemented in practice.

```
FOR each chunk of M-1 block of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;
```

Figure 15.8: The nested-loop join algorithm.

15.3.4 Analysis of Nested-Loop Join

Left out.

15.3.5 Summary of Algorithms so Far

The main-memory and disk I/O's requirements for the algorithms we have discussed in Sections 15.2 and 15.3 are shown in Fig. 15.9 below:

Operators	Approximate M required	Disk I/O	Section
σ, π	1	B	15.2.1
γ, δ	B	B	15.2.2
$\cup$, $\cap$, $-$, $\times$. $\bowtie$	$\min(B(R),\, B(S))$	$B(R) + B(S)$	15.2.3
$\bowtie$	any $M \geq 2$	$B(R)B(S)/M$	15.3.3

Figure 15.9: Main memory and disk I/O requirements for one-pass and nested-loop algorithms.

15.4 Two-Pass Algorithms Based on Sorting

We shall now concentrate on `two-pass algorithms`, where data from the operand relations is read into main memory, processed in some way, written out to disk again, and then reread from disk to complete the operation.

15.4.1 Two-Phase, Multiway Merge-Sort

It is possible to sort very large relations in two passes using an algorithm called `Two-Phase, Multiway Merge-Sort (TPMMS)`. Suppose we have M main-memory buffers to use for the sort. TPMMS sorts a relation R as follows:

- *Phase 1:* Repeatedly fill the M buffers with new tuples from R and sort them, using any main-memory sorting algorithm. Write out each *sorted sublist* to secondary storage.
- *Phase 2:* Merge the sorted sublists. For this phase to work, there can be at most $M-1$ sorted sublists, which limits the size of R.

15.4.2 Duplicate Elimination Using Sorting

To perform the $\delta(R)$ operation in two passes, we sort the tuples of R in sublists as in TPMMS. In the second pass, we use the available main memory to hold one block from each sorted sublist and one output block, as we did for TPMMS. However, instead of sorting on the second pass, we repeatedly select the first unconsidered tuple t among all the sorted sublists. We write one copy of t to the output and eliminate from the input blocks all occurrences of t. Thus, the output will consist of exactly one copy of any tuple in R. They will in fact be produced in sorted order.

15.4.3 Grouping and Aggregation Using Sorting

Left out.

15.4.4 A Sort-Based Union Algorithm

Left out.

15.4.5 Sort-Based Intersection and Difference

Left out.

15.4.6 A Simple Sort-Based Join Algorithm

There are several ways that sorting can be used to join large relations. Given relations $R(X, Y)$ and $S(Y, /, Z)$ to join, and given M blocks of main memory for buffers,w e do the following:

1. Sort R, using TPMMS with Y as the sort key.
2. Sort S similarly.
3. Merge the sorted R and S. We use only two buffers: one for the current block of R and the other for the current block of S. The following steps are done repeatedly:
    1. Find the least value y of the join attributes Y that is currently at the front of the blocks of R and S.
    2. If y does not appear at the front of the other relation, then remove the tuples with sort key y.
    3. Otherwise, identify all the tuples from both relations having sort key y. If necessary, read blocks from the sorted R and/or S, until we ware sure there are no more y's in either relation.
    4. Output all the tuples that can be formed by joining tuples from R and S that have a common Y-value y.
    5. If either relation has no more unconsidered tuples in main memory, reload the buffer for that relation.

15.4.7 Analysis of Simple Sort-Join

Left out.

15.4.8 A More Efficient Sort-Based Join

Left out.

15.4.9 Summary of Sort-Based Algorithms

In Fig. 15.11 is a table of the analysis of the algorithms we have discussed in Section 15.4.

Operators	Approximate M required	Disk I/O	Section
$\tau,\ \gamma,\ \delta$	$\sqrt{B}$	$3B$	15.4.1, 15.4.2, 15.4.3
$\cup,\ \cap,\ -$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.4, 15.4.5
$\bowtie$	$\sqrt{\max(B(R),\ B(S))}$	$5(B(R)+B(S))$	15.4.6
$\bowtie$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.8

Figure 15.11: Main memory and disk I/O requirements for sort-based algorithms.

15.5 Two-Pass Algorithms Based on Hashing

There is a family of hash-based algorithms that attack the same problems as in Section 15.4. The essential idea behind all these algorithms is as follows.
If the data is too big to store in main-memory buffers, hash all the tuples of the argument to arguments using an appropriate hash key. For all the common operations, there is a way to select the hash key so all the tuples that need to be considered together when we perform the operation fall into the same bucket. We then perform the operation by working on one bucket at a time.

In effect, we have reduced the size of the operands by a factor equal to the number of buckets, which is roughly M.

15.5.1 Partitioning Relations by Hashing

To begin, let us review the way we would take a relation R and, using M buffers, partition R into $M-1$ buckets of roughly equal size. We associate one buffer with each bucket. Each tuple t in the block is hashed to bucket $h(t)$ and copied to the appropriate buffer. If that buffer is full, we write it out to disk, and initialize another block for the same bucket.

15.5.2 A Hash-Based Algorithm for Duplicate Elimination

Consider duplicate elimination, that is, the operation $\delta(R)$. We hash R to $M-1$ buckets. Note that two copies of the same tuple t will hash to the same bucket. Thus, we can examine one bucket at a time, perform δ on that bucket in isolation, and take as the answer the union of $\delta(R_i)$, where R_i is the portion of R that hashes to the ith bucket. The one-pass algorithm of Section 15.2.2 can be used to eliminate duplicates from each R_i in turn and write out the resulting unique tuples.

15.5.3 Hash-Based Grouping and Aggregation

To perform the $\gamma_L(R)$ operation, we again start by hashing all the tuples of R to $M-1$ buckets. However, in order to make sure that all tuples of the same group wind up in the same bucket, we must choose a hash function that depends only on the grouping attributes of the list L.

Having partitioned R into buckets, we can then use the one-pass algorithm for γ from Section 15.2.2 to process each bucket in turn.

15.5.4 Hash-Based Union, Intersection, and Difference

Left out.

15.5.5 The Hash-Join Algorithm

To compute $R(X, Y) \bowtie S(Y, Z)$ using a two-pass, hash-based algorithm, we act almost as for the other binary operations discussed in Section 15.5.4. The only difference is that we must use as the hash key just the join attributes Y. Then we can be sure that if tuples of R and S join, they will wind up in corresponding buckets R_i and S_i for sone i. A one-pass join of all pairs of corresponding buckets completes this algorithm, which we call `hash-join`.

15.5.6 Saving Some Disk I/O's

Left out.

15.5.7 Summary of Hash-Based Algorithms

Fig. 15.13 gives the memory requirements and disk I/O's needed by each of the algorithms discussed in this section.

Operators	Approximate M required	Disk I/O	Section
γ, δ	$\sqrt{B}$	$3B$	15.5.2, 15.5.3
$\cup, \cap, -$	$\sqrt{B(S)}$	$3(B(R)+B(S))$	15.5.4
$\bowtie$	$\sqrt{B(S)}$	$3(B(R)+B(S))$	15.5.5
$\bowtie$	$\sqrt{B(S)}$	$(3-2M/B(S))(B(R)+B(S))$	15.5.6

Notice that the requirements for sort-based and the corresponding hash-based algorithms are almost the same. The significant differences between the two approaches are:

1. Hash-based algorithms for binary operations have a size requirement that depends only on the smaller of two arguments rather than on the sum of the argument sizes.
2. Sort-based algorithms sometimes allow us to produce a result in sorted order and take advantage of that sort later.
3. Hash-based algorithms depend on the buckets being of equal size.
4. In sort-based algorithms, the sorted sublists may be written to consecutive blocks of the disk if we organize the disk properly. Thus, one of the three disk I/O's per block my require little rotational latency or seek time.

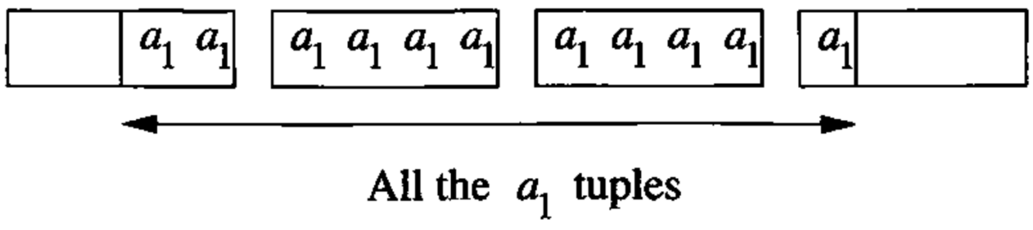
15.6 Index-Based Algorithms

The existence of an index on one or more attributes of a relation make available some algorithms that would not be feasible without the index.

15.6.1 Clustering and Nonclustering Indexes

Recall from Section 15.1.3 that a relation is "clustered" if its tuples are packed into roughly as few blocks as can possibly hold those tuples.

We may also speak of `clustering indexes`, which are indexes on an attribute or attributes such that all the tuples with a fixed value for the search key of this index appear on roughly as few blocks can hold them.



15.6.2 Index-Based Selection

Suppose that the condition C is of the form $a = v$, where a is an attribute for which an index exists, and v is a value. Then one can search the index with value v and get pointers to exactly those tuples of R that have a-value v. These tuples constitute the result $\sigma_{a=v}(R)$, so all we have to do is retrieve them.
If the index on $R.a$ is a clustering index, then the number of disk I/O's to retrieve the set $\sigma_{a=v}(R)$ will average $B(R)/V(R,\ a)$. Let us consider what happens when the index on $R.a$ is nonclustering. To a first approximation, each tuple we retrieve will be on a different block, and we must access $T(R)/V(R,\ a)$ tuples. Thus, $T(R)/V(R,\ a)$ is an estimate of the number of disk I/O's we need.

15.6.3 Joining by Using an Index

Let us examine the natural join $R(X,\ Y) \bowtie S(Y,\ Z)$.
For our first index-based join algorithm, suppose that S has an index on the attributes Y. Then one way to compute the join is to examine each block of R, and within each block consider each tuple t. Let t_Y be the component or components of t corresponding to the attributes Y. Use the index to find all those tuples of S that have t_Y in their Y-components. These are exactly the tuples of S that join with tuple t of R, so we output the join of each of these tuples with t.

15.6.4 Joins USing a Sorted Index

When the index is a B-tree, or any other structure from which we easily can extract the tuples of a relation in sorted order, we have a number of other opportunities to use the index. Perhaps the simplest is when we want to compute $R(X,\ Y) \bowtie S(Y,\ Z)$, and we have such an index on Y for either R or S. WE can then perform an ordinary sort-join, but we do not have to perform the intermediate step of sorting one of the relations on Y.

As an extreme case, if we have sorting indexes on Y for both R and S, then we need to perform only the final step of the simple sort-based join of Section 15.4.6. This method is sometimes called `zig-zag join`.

15.7 Buffer Management

We have assumed that operators on relations have available some number M of main-memory buffers that they can use to store needed data.
The central task of making main-memory buffers available to processes, such as queries, that act on the database is given to the `buffer manager`. It is the responsibility of the buffer manager to allow processes to get the memory they need, while minimizing the delay and unsatisfiable requests.

15.7.1 Buffer Management

There are two broad architectures for a buffer manager:

1. The buffer manager controls main memory directly, as in many relational DBMS's, or
2. The buffer manager allocates buffers in virtual memory, allowing the operating system to decide which buffers are actually in main memory at any time and which are in the "swap space" on disk that the operating system manages.

Whichever approach a DBMS uses, the same problem arises: the buffer manager should limit the number of buffers in use so they fit in the available main memory.
Normally, the number of buffers is a parameter set when the DBMS is initialized. We would expect that this number is set so that the buffers occupy the available main memory, regardless of whether the buffers are allocated in main or virtual memory. In what follows, we shall not concern ourselves with which mode of buffering is used, and simply assume that there is a fixed-size `buffer pool`, a set of buffers available to queries and other database actions.

15.7.2 Buffer Management Strategies

The critical choice that the buffer manager must make is what block to throw out of the buffer pool when a buffer is needed for a newly requested block. The `buffer-replacement strategies` in common use may be familiar to you from other applications of scheduling policies, such as in operating systems. These include:

Least-Recently Used (LRU)

The `LRU` rule is to throw out the block that has not been read or written for the longest time. This method requires that the buffer manager maintain a table indicating the last time the block in each buffer was accessed.

First-In-First-Out (FIFO)

When a buffer is needed, under the FIFO policy the buffer that has been occupied the longest by the same block is emptied and used for the new block. In this approach, the buffer manager needs to know only the time at which the block currently occupying a buffer was loaded into that buffer.

The "Clock" Algorithm ("Second Chance")

This algorithm is a commonly implemented, efficient approximation to LRU. Think of the buffers as arranged in a circle. A "hand" points to one of the buffers, and will rotate clockwise if it needs to find a buffer in which to place a disk block. Each buffer has an associated "flag", which is either 0 or 1. Buffers with a 0 flag are vulnerable to having their contents sent back to disk, buffers with a 1 are not. When a block us read into a buffer, its flag is set to 1. Likewise, when the contents of a buffer is accessed, its flag is set to 1. When the buffer manager needs a buffer for a enw block, it looks for the first 0 it can find, rotating clockwise. If it passes 1's,it sets them to 0.

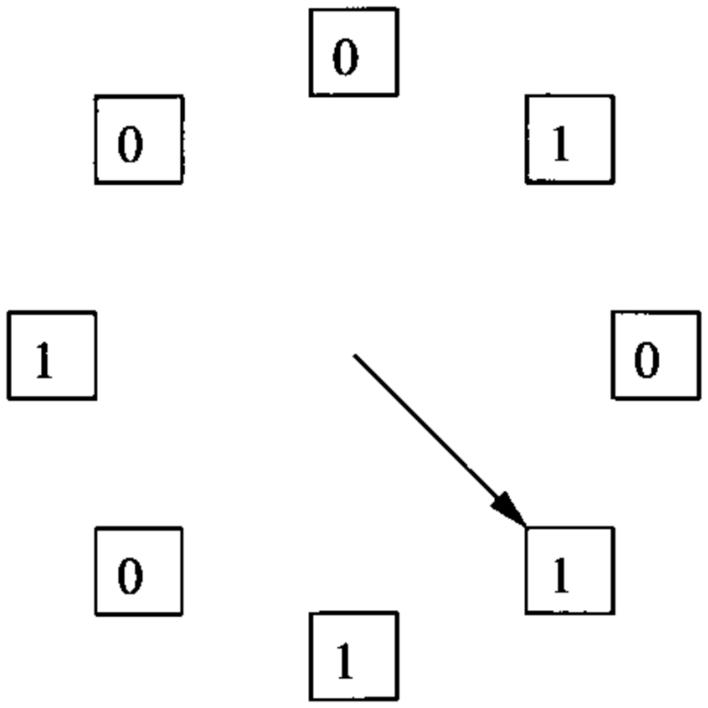


*Figure 15.17: The clock algorithm visits buffers in a round-robin fashion and replaces $01 \cdots 1$ with $10 \cdots 0$.

15.7.3 The Relationship Between Physical Operator Selection and Buffer Management

The query optimizer will eventually select a set of physical operators that will be used to execute a given query. This selection of operators may assume that a certain number of buffers M is available for execution of each of these operators. However, as we have seen, the buffer manager may not be willing or able to guarantee the availability of these M buffers when the query is executed.

15.8 Algorithms Using More Than Two Passes

While two passes are enough for operations on all but the largest relations, we should observe that the principal techniques discussed in Section 15.4 and 15.5 generalize to algorithms that, by using an many passes as necessary, can process relations of arbitrary size.

15.8.1 Multipass Sort-Based Algorithms

Suppose we have M main-memory buffers available to sort a relation R, which we shall assume is stored clustered. Then do the following:

BASIS: If R fits in M blocks, then read R into main memory, sort it using any main-memory sorting algorithm, and write the sorted relation to disk.

INDUCTION: If R does not fit into main memory, partition the blocks holding R into M groups, which we shall call $R_1,\ R_2, \dots,\ R_M$. Recursively sort R_i for each $i = 1,\ 2, \dots,\ M$. Then, merge the M sorted sublists, as in Section 15.4.1.

If we are not merely sorting R, but performing a unary operation such as γ or δ on R, then we modify the above in the following way:

- For δ, output one copy of each distinct tuple, and skip over copies of the tuple.
- For a γ, sort on the grouping attributes only, and combine the tuples with a given value of these grouping attributes in

the appropriate manner.

15.8.2 Performance of Multipass, Sort-Based Algorithms

Left out.

15.8.3 Multipass Hash-Based Algorithms

There is a corresponding recursive approach to using hashing for operations on large relations. We hash the relation or relations into $M - 1$ buckets. WE then apply the operation to each bucket individually. We can describe this approach recursively as:

BASIS: For a unary operation, if the relation fits in M buffers, read it into memory and perform the operation. FOr a binary operation, if either relation fits in $M - 1$ buffers, perform the operation by reading this relation into main memory and then read the second relation, one block at a time, into the Mth buffer.

INDUCTION: If no relation fits in main memory, then hash each relation into $M - 1$ buckets. Recursively perform the operation on each bucket or corresponding pair of buckets, and accumulate the output from each bucket or pair.

15.8.4 Performance of Multipass Hash-Based Algorithms

Left out.

15.9 Summary of Chapter 15

Query Processing

Queries are compiled, which involves extensive optimization, and then executed.

Query Plans

Queries are compiled first into `logical query plans`, which are often like expressions of relational algebra, and then converted to a `physical query plan` by selecting an implementation for each operation, ordering joins and making other discussions.

Table Scanning

To access the tuples of a relation, there are several possible physical operators. The `table-scan` operator simply reads each block holding tuples of the relation. `Index-scan` uses an index to find tuples, and `sort-scan` produces the tuples in sorted order.

Iterators

Several operations involved in the execution of a query can be meshed conveniently if we think of their execution as performed by an iterator. This mechanism consists of three methods, to open the construction of a relation, to produce the next tuple of the relation, and to close the construction.

One-Pass Algorithms

As long as one of the arguments of a relational-algebra operator can fit in main memory, we can execute the operator by reading the smaller relation to memory, and reading the other argument one block at a time.

Nested-Loop Join

This simple join algorithm works even when neither argument fits in main memory. It reads as much as it can of the smaller relation into memory, and compares that with the entire argument. This process is repeated until all of the smaller relation has had its run in memory.

Two-Pass Algorithms

Except for nested-loop join, most algorithms for arguments that are too large to fit into memory are either sort-based, hash-based, or index-based.

Sort-Based Algorithms

These partition their arguments into main-memory-sized, sorted sublists. The sorted sublists are then merged appropriately to produce the desired result.

Hash-Based Algorithms

These use a hash function to partition the arguments into buckets. The operation is then applied to the buckets individually or in pairs.

Index-Based Algorithms

The use of an index is an excellent way to speed up a selection whose condition equates the indexed attribute to a constant.

The Buffer Manager

The availability of blocks of memory is controlled by the buffer manager. When a new buffer is needed in memory, the buffer manager uses one ot the familiar replacement policies to decide which buffer is returned to disk.

Multipass Algorithms

The two-pass algorithms based on sorting or hashing have natural recursive analogs that take three or more passes and will work for larger amounts of data.