

Chapter 10

Query Processing Basics

An application designer has to understand some basic principle and methods of query processing to produce a more efficient system.

In this chapter, we discuss some of the methods used to evaluate the basic relational operators and discuss their impact on physical database design.

SQL queries, after being submitted, will first be parsed by the DBMS parser, which verifies the syntax of the query and, using the system catalog, determines if the attribute references such as names and types are correct.

Then what?

Since an SQL query is declarative, it does not suggest any specific implementation. To achieve possible improvement, an SQL query will be converted to a relational algebraic expression.

For example, the following query

```
Select S.Name From Transcript T, Student S
Where T.Semester='F2004' And S.Id=T.StudId
      And S.Name='John Doe'
```

will be put into the following:

$$\pi_{Name}(\sigma_{C_1}(Transcript \times Student)),$$

where C_1 refers to the condition in the Where clause.

This expression can then be directly, but naively, evaluated.

It works, but...

The problem for this naive evaluation is that the involved Cartesian product can lead to a huge intermediate table, although at the end, we only have a small result.

For example, in observing that the above expression only cares about “John Doe”, The *optimizer* will generate the following equivalent, but a lot “simpler”, expression:

$$\pi_{Name}(\sigma_{Semester='F2004} (Transcript \bowtie_{StudId=Id} (\sigma_{Name='JohnDoe'}(Student))))$$

Once such a simple expression is reached, DBMS will choose algorithms to carry out such operations as projection, selection, and join.

External sorting

Sorting is at the very core of all the algorithms that perform relational operations. It is, for example, one of the more efficient ways to get rid of duplicates (?), and is also the basis for the join algorithms, as we will see.

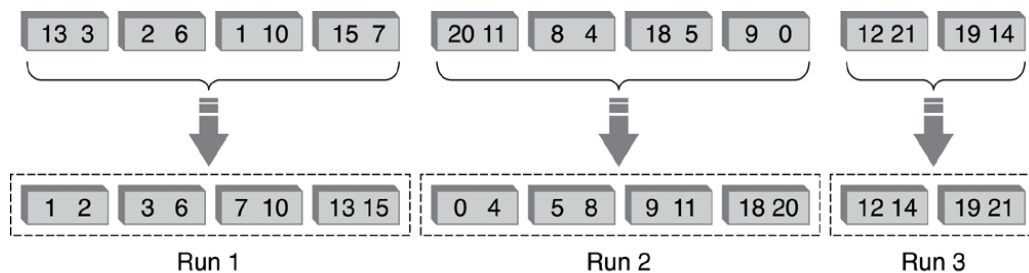
We once studied various sorting algorithms, in e.g., *CS3220 Data structures and algorithm analysis*, but we can't use any of them, since the data we have to sort is too big to fit in the main memory.

When the files are large and are kept in the external memory, such as a disk, we use *external sorting* algorithms to sort them out.

The essential idea is...

to bring in a portion of the data into the main memory and apply the traditional sorting algorithms to get them sorted and then put the sorted data back to the disk.

Eventually, we will have a bunch of partially sorted data, often called *runs*.



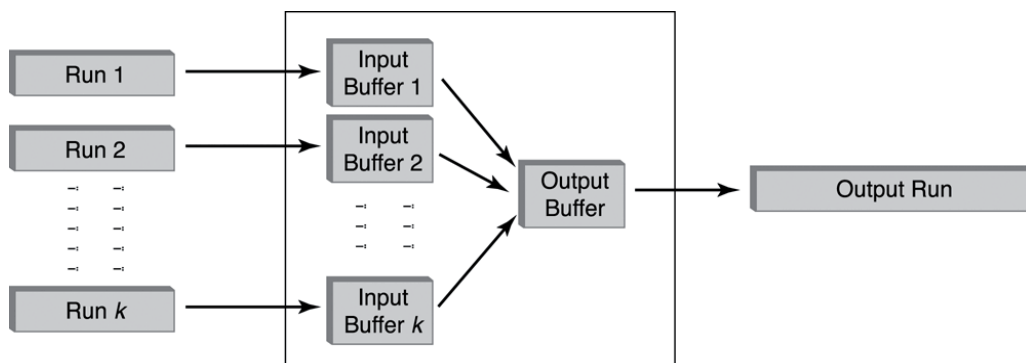
As the next and final step, we then *merge* all the runs into a whole table of sorted data, just as what we did in *mergesort*.

k -way merge

The merge algorithm is pretty simple

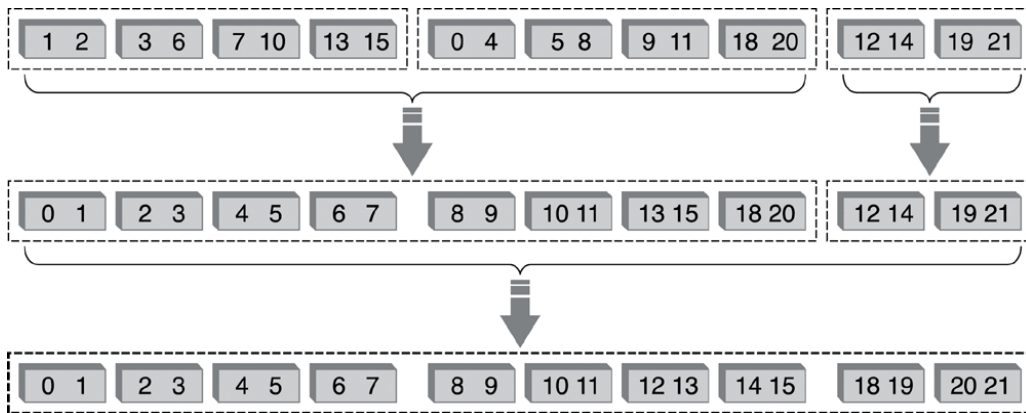
While (there are still runs) do
 choose a smallest tuple in each run
 and output the smallest among them
 delete the chosen tuple from its run.

This process is demonstrated with the following figure

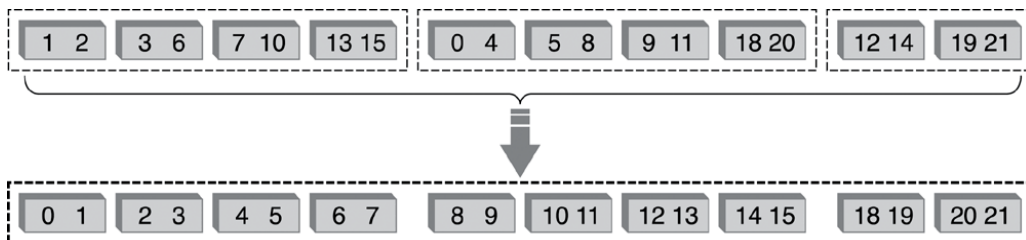


Two merging examples

Below shows a 2-way merging.



Now we show a 3-way merging.



A bit technical...

When the portion of the data is brought into the main memory, it is placed in a buffer. Assume that the buffer can handle M pages, and the file has F pages, which is usually much larger than M (?). Thus, the run generation part generates $\lceil F/M \rceil$ runs, at the cost of $2F$ disk I/Os: For each page, it takes one operation to bring it into the memory, and another to send it back to the disk.

It take a bit more analysis for the merging part, which is made in pp. 383, and leads to the result of $2F(\log_{M-1} F - 1)$..

Thus, the total cost of the external mergesort is $2F \log_{M-1} F$.

Projection, Union and Difference

They all sound like pretty easy. For projection of a table on some attributes, we just scan the table and delete the unwanted columns. However, if the user puts on a `distinct` restriction, the DBMS must take away the duplicates. So, we have to find some efficient way to remove them.

The same thing happens to the union operation, which does not allow duplicates. To calculate the difference $r - s$, we have to identify those in s that also appear in r , and take them out.

Thus, they all come down to duplicate identification.

Which way do you prefer?

There are two ways to identify the duplicates: sorting or hashing. When applying a sort-based projection, we scan the whole table, remove those tuple components that are projected out, and then write the result back to the disk, assuming that there is not enough space for us to keep them internally. This step costs $2F$.

We then sort the result in $2F \log_{M-1} F$, and scan them again, again in $2F$, to remove the duplicates which must sit next to each other.

We can also improve this process a bit by combining the sorting and scanning. We can delete those unwanted attributes during the sorting stage since we have to do a scanning there anyway, and the duplicates can be deleted when we produce the runs during the sorting process too, since they have to be there next to each other.

Hash based projection

Another way is to use a hash function, which maps tuples with the same values into the same bucket. Thus, all the duplicates must be thrown into the same bucket, which might contain other stuff as well.

More specifically, we scan the original table, during which we delete the unwanted columns; we then hash the rest of the tuples on the remaining attributes, and whenever a page in the hash table becomes full, it is transferred to a bucket in the disk.

We can then search for duplicates in each bucket.

For the other two

The calculation for union and difference is similar except that we don't need to throw out the unwanted attributes.

When we calculate $r - s$, we sort them separately, then scan them in parallel, just like what we do in the merging process. Whenever we discover a tuple $t \in r$ is also in s , we don't add it to the result.

For the union, we only add in one of the duplicates into the result, when identified.

Computing selection

This could be much more complex, since it depends on what indices we have kept for the involved tables. There are a bunch of techniques available, depending on the type of the selection condition and on the physical organization of the tables in question.

Let's have a look at some of the heuristics a DBMS might use so that we can request a physical organization that is most favorable to the selection types that occur most frequently in our application.

Simple conditions

Given $\sigma_{attri\ op\ value}(r)$, an obvious way is to scan the relation r and check the condition for all the rows, and output those that satisfy the condition. However, this process would cost us too much if only a small portion of the tuples satisfy this condition; and it is not necessary if there are more information for the structure of r . We discuss several cases:

1. In case there is no index on $attri$, we have no choice but scan r at the cost of F page transfers. But, if r is sorted on $attri$, we can do a binary search to find the page that contains the first tuple that meets the cut, then follow it through, with the cost of about $\log_2 F$.

2. If there is a B^+ tree on *attri*, we can use an algorithm that is similar for a sorted file. We can use the tree to find all the indices for the tuples that satisfy this search condition. The cost is essentially equal to the depth of the B^+ tree, which again is about $\log_2 F$. The time to get all the tuples depends also on whether the index is clustered or not. If it is, then all the tuples are in the same neighborhood; otherwise, they are all over the place, which needs many I/O operations.

3. If there is a hash index, we can use the hash function to find all the buckets for the tuples for which the condition is met. The time to find a bucket, assuming an efficient hash function, is constant. But, the actual cost of getting all the tuples depend on the number of qualifying tuples, and thus on where the index is clustered or not.

Access paths

The aforementioned algorithms all assume that certain indices are available for the relations being processed. These indices, together with the associated algorithms, are called *access paths*.

For example, a *file scan* can always be used, no matter an index exists or not; in case a file is sorted on attributes specified in the *Where* clause, a *binary search* can be used; a *hash index* or a B^+ tree can be used if the index has a search key that involves the specified attributes.

An example

Consider the `Transcript` table and assume that we have a hash index on the search key (`StudId`, `Semester`).

Such an index is useful in computing the query $\pi_{StudId, Semester}(Transcript)$, since we can be certain that any duplicates created by this query will be hashed in the same bucket.

But, it will not be helpful if what we want is $\pi_{StudId, CrsCode}(Transcript)$, since the involved `Semester` value can be different, thus it is unlikely that the tuples with same values for `StudId` and `CrsCode` will be mapped into the same bucket under the hash function for (`StudId`, `Semester`).

.

Another example

This index is also useful when we calculate $\sigma_{StudId=9999 \wedge Grade='A' \wedge Semester='F1994'}(Transcript)$ since we can use the hash index to get all the tuples that satisfy the condition $StudId = 9999 \wedge Semester = 'F1994'$ from the Transcript table, and then scan the result to pick up those who got an 'A'.

This index is of no help in calculating the query $\sigma_{StudId=9999}(Transcript)$ since this hash index needs values for both attributes.

A hash index on (Grade, StudId) does not help us in computing $\sigma_{Grade > 'C'}(Transcript)$, since hash function needs specific values; while a B^+ tree on the pair does help, since it works for range searches.

Finally, a tree on (StudId, Grade) won't, since in this case Grade is not a prefix.

A notion of coverage

The above example motivates the following notion:

Consider an expression $\sigma_{(a_1 \text{ op}_1 v_1) \wedge \dots \wedge (a_n \text{ op}_n v_n)}(R)$, we say this expression is *covered* by an access path iff one of the following holds:

1. The access path is a file scan.
2. the access path is a hash index whose search key is a subset of a_1, a_2, \dots, a_n and all the operators op_i 's are equality operators.
3. The access path is a B^+ tree with the search key sk_1, \dots, sk_n such that some of the prefix sk_1, \dots, sk_i is a subset of the attribute set.
4. The access path is a binary search, and the relations instance is sorted on a_1, a_2, \dots, a_n .

We use the term *selectivity for an access path* to refer to the number of pages that path will retrieve.

Complex conditions

If the complex conditions are constructed with conjunctions, i.e., logic ands, we can use the most selective access paths to retrieve the corresponding tuples. Such an access path tries to use as many attributes as possible thus getting as small a set as possible, then we can scan this set to find the tuples that satisfy the whole condition.

Given $\sigma_{Grade > 'C' \wedge Semester = '1994'}(Transcript)$, and a B^+ tree with the search key (Grade, StudId). Since this tree covers the first part of the condition, we can use this tree to compute a set for $\sigma_{Grade > 'C'}(Transcript)$, and then scan the result to get the answer.

If we have a tree on (Semester, Grade), it will be even better since it covers both parts.

We can also use several access paths that cover the expression. For example, we might have two secondary indices that might do better than a plain file scan. We can then use them to generate a candidate sets and then compute their intersection, and test the remaining for satisfaction of the original condition.

Given $\sigma_{StudId=9999 \wedge Grade='A' \wedge Semester='F1994'}(Transcript)$, if we have two hash indices on `Semester` and `Grade`. Using the first access path, we can find the students who took courses in Fall 1994, and then the second access path leads to the students who got 'A' in any course any time. We then form an intersection of these two sets to get those who got an 'A' in Fall 1994.

We finally do a scan in this much smaller set to find out the tuple for the one who has an id of 9999.

How about disjunction?

It will take some more efforts. We first have to convert it into a *disjunctive normal form*, which is guaranteed to exist for any expression.

For example, given $(G = 'A' \vee G = 'B') \wedge (S = 'F1994' \vee S = 'F1995')$, we will get its equivalent disjunctive normal form as follows:

$$(G = 'A' \wedge S = 'F1994') \vee (G = 'A' \wedge S = 'F1995') \\ \vee (G = 'B' \wedge S = 'F1994') \vee (G = 'B' \wedge S = 'F1995')$$

The DBMS then has to check out the access paths for all the individual disjuncts, and decides what to do.

Then what?

If we find out that one of the disjunctive term C_i must be evaluated using a file scan, we then might just evaluate the whole thing using this file scan.

On the other hand, if we find out that every disjunctive term C_i has an access path that is better than a plain file scan, but the sum of their selectivity is close to that of the file scan, we might just do a file scan to avoid the overheads that those separate access paths will lead.

If the sum of their selectivity is much less than that of the file scan, we will then computer $\sigma_{C_i}(R)$ separately, then apply a union of the result.

Homework: 10.7.

Computing joins

We once went through a process of computing natural join, which can be generalized to any join. We will be now more serious about it.

The work we just did in computing projection, selection is just a warm-up. A join can be implemented as a combination of all these, starting with a Cartesian product, which is the worst, since the size of a Cartesian table is the product of the respective sizes, or *quadratic* in the size of the input. For example, if we join two tables with 1000 pages each, we might have to do a million I/O operations, which is just not practical.

There are essentially three ways to do a join: nested loops; sort-merge; and hash based.

Using nested loops

One way to evaluate a join $r \bowtie_{A=B} s$ is to use the following loop:

```
foreach  $t \in r$  do
  foreach  $t' \in s$  do
    if  $t.A = t'.b$  then output  $\langle t, t' \rangle$ 
```

Let F_r and F_s be the number of pages for r and s , and let τ_r and τ_s be the number of tuples in r and s , respectively. Clearly, we have to scan every tuple in s for every tuple in r , thus resulting in $\tau_r \times F_s$ page transfers. Moreover, r must be scanned once in the outside loop.

Thus the total transfer for this algorithm is $F_r + \tau_r \times F_s$.

So what?

1. Lots of page transfers have to be done.

Let $F_r = 1000$, $F_s = 100$, and $\tau_r = 10,000$. The previous estimation tells us that the computation of $r \bowtie s$ takes $1000 + 10000 \times 100 = 1,001,000$ page transfers, about 166 minutes if one such transfer takes 10ms.

2. The order of the join matters.

If we switch the order of the previous join, $s \bowtie r$, and assume that $\tau_s = 1000$, we have that the number of transfers becomes $100 + 1000 \times 1000 = 1,000,100$. It is not much in this case, but might lead to a significant reduction in other cases.

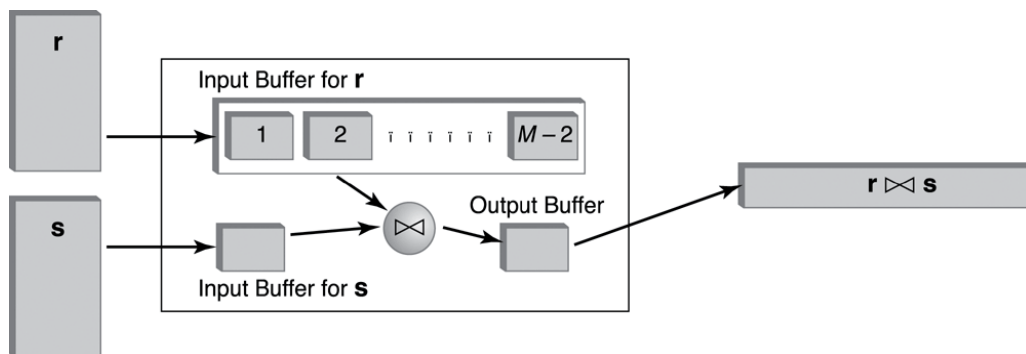
Since it leads to too high a cost, simple nested loop is never used to compute joins.

Using block-nested loop

The cost of the nested-loop join can be cut drastically if, instead of scanning s for every tuple of r , we scan it for every page for r . This will cut the cost to $F_r + F_r \times F_s$. We can achieve this by sending out the join of a page of r and a page of s at each time as follows:

```
foreach page  $p_r$  of  $r$  do
  foreach page  $p_s$  of  $s$  do
    output  $p_r \bowtie_{A=B} p_s$ .
```

We can do more: assume that the query processor has an M page buffer to do the join, we can allocate $M - 2$ pages of r and one s page and let the last page be the output buffer area.



How much will we save?

The outside loop does a complete scan of r , thus F_r transfers, while s is scanned for every group of $M - 2$ pages of r , thus, the total cost becomes

$$F_r + F_s \left\lceil \frac{F_r}{M - 2} \right\rceil.$$

For our concrete example, if $M = 102$, we will have that the total number of page transfers to do $r \bowtie s$ will be $1000 + 100 \times 10 = 2000$; and if we do $s \bowtie r$ instead, it will be $1000 + 100 \times 1 = 1,100$ transfers, or about 11 seconds, assuming each transfer takes 10 ms.

Thus, order again matters; while the cost comes down a lot.

Using index nested loops

We love to use indices. It works particularly well if the number of tuples in r and s that do match under the condition is smaller compared with the size of r and s .

Assume s has an index on the attribute B , then we can do the following:

```
foreach  $t \in r$  do
  use the index on  $B$  to find all tuples  $t' \in s$ 
  such that  $t.A = t'.B$ 
  Output  $\langle t, t' \rangle$  for each such  $t'$ 
```

The cost depends on what sort of index we have on B and whether it is clustered or not. If the index is a B^+ tree, the cost of finding the first node for the matching tuple in s is about 2 to 4 I/O operations. In a hash-based index, it is about 1.2.

Get the tuples

Once the indices are in our hands, we also need to get the tuples themselves, which depends on how many tuples satisfying the condition and whether the index is clustered or not.

In case of a clustered index, the cost is

$$F_r + (\rho + 1) \times \tau_r,$$

where ρ is the number of transfers it takes to find the index.

In the case of an unclustered index, the cost is

$$F_r + (\rho + \mu) \times \tau_r,$$

where μ is the average number of tuples that satisfy the condition.

Notice that the estimate has nothing to do with s .

Go back to the example

Assume that $\rho = 2$ since we have a small table, we get the cost for a clustered index is $1000 + 3 \times 10,000 = 31,000$ transfers; but it comes down to $100 + 3 \times 1000 = 3100$ transfers if we switch the order for the join.

Although index based join seems not as efficient as block-nested join, it does have one nice feature: the cost does not go up as dramatically when the size of the inner table changes.

In our example, when calculating $r \bowtie s$, and the size of s grows to 10,000 pages with 100,000 tuples, the cost for a block-nested join will go from 2,000 to $1000 + 10000 \times 10 = 101,000$ transfers; while for the index-based approach, it stays at 31,000.

Sort-merge join

The idea is to sort each relation on the join attributes, and then to find matching tuples using the same merging technique, i.e., scan both sorted relations in parallel and compare the join attributes. When a matching between two tuples on the specified attributes is found, all possible combination between the respective tuples via this matched value are added to the result.

An algorithm is given in Figure 10.7 in pp. 396.

It is estimated that with this sort of join, it has to make 4,200 page transfers.

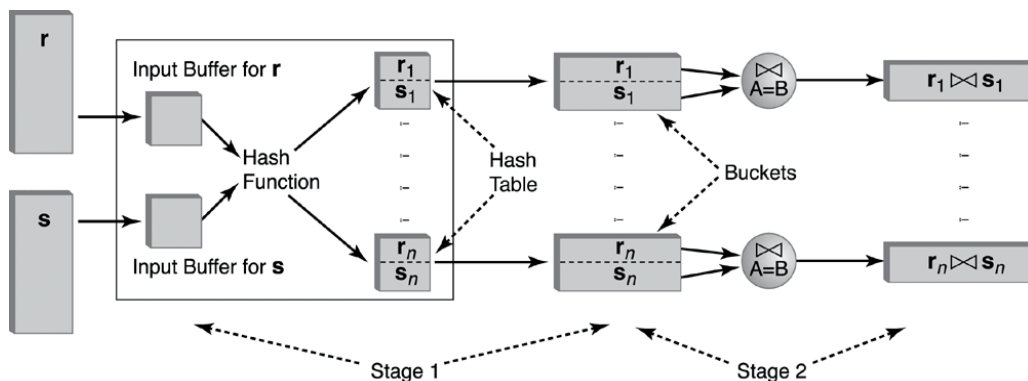
In general, the average behavior of the sorted-merge join is better than the block-nested join.

Homework: 10.8.

What's the point?

The key idea behind the sorting-merge join is to do a preprocessing of the involved relations so that the tuples that possibly match will be placed in the same or adjacent pages. We then need not scan for matching tuples.

We can also use some sort of hashing functions to place these matching tuples next to each other in their buckets.



Here, r is hashed on A , and s is hashed on B , thus those pairs with matching values will be placed in the same bucket, and will then be joined.

Computing aggregates

In general, to calculate aggregates such as `AVG`, `Count`, etc., in a query needs to scan the whole table.

The only issue is what happens when we also have to deal with `Group By` statement. We again have to find out efficient methods to partition tuples according to certain values. We can use a combination of the usual methods such as sorting, hashing and indexing to access the tuples that are identified by the `Group` clause.

Once this is done, we can then apply the aggregates to the groups of tuples via a scan.