Advances in Data Management
Query Processing and Query Optimisation
A.Poulovassilis

1 General approach to Query Processing and Optimisation in DBMS

1. Parse and translate queries expressed in a high-level query language into a small set of lower-level *algebraic operators*.
2. Provide *efficient implementations* of these operators.
3. Devise a *cost model* for these operators.
4. Use the *algebraic laws* governing the behaviour of the operators to generate a set of *candidate query plans* for a given query.
5. Use the *cost model* and the *statistics* maintained by the DBMS to estimate a cost for each candidate query plan. Select the cheapest plan for execution.

The general architecture of a DBMS's Query Processor is illustrated in Figure 1.

2 Translation into algebraic form

Before it is optimised and evaluated, an SQL query is first parsed and translated into an expression in the **extended relational algebra**.

This algebra consists of the classical relational algebra operators, but operating on *bags* — i.e. collections that may contain duplicates — rather than sets. Also, the classical operators are extended with GROUP BY and HAVING operators, and aggregation operators (SUM,COUNT,AVG,MIN,MAX) are allowed to appear in the argument list of the projection operator, π.

If an SQL query references a view, then the view definition is first substituted into the query before the query is translated into the extended relational algebra.

Each "block" of an SQL query, i.e. each SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... clause, is translated into a separate expression in the extended relational algebra.

The extended relational algebra expression for each block consists of an inner σ, π, $\times$ expression, with possibly additional GROUP BY, HAVING and π operators applied to it (in that order, from the "inside-out").

If a subquery is a correlated query, its parent query is considered as providing a selection condition within the subquery.

Translation Example. Translate the following SQL query (which returns the earliest reservation of a red boat by each sailor of the highest rating) into the extended relational algebra. There are two query blocks, and therefore two extended relational algebra expressions will be produced:

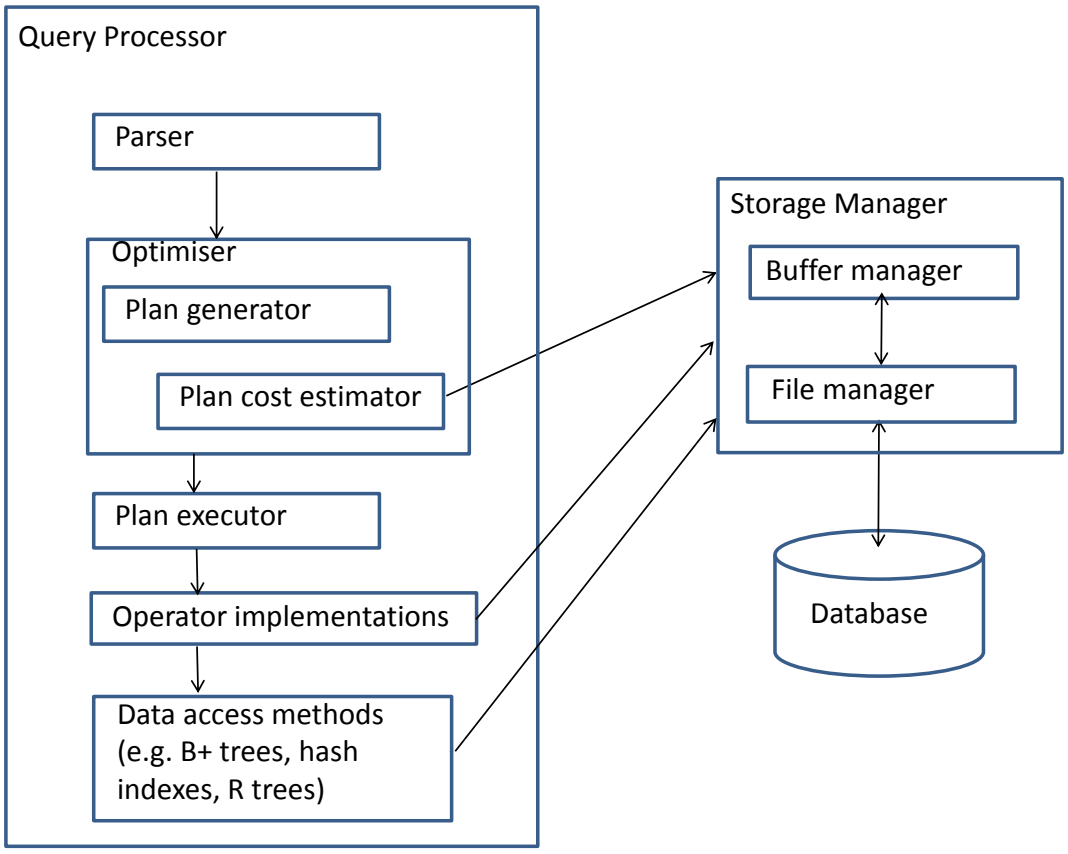

Figure 1: Query Processor Architecture

$$
\begin{array}{ll}
SELECT & S.sid, MIN(R.day) \\
FROM & Sailors\ S, Reserves\ R, Boats\ B \\
WHERE & S.sid = R.sid\ AND\ R.bid = B.bid\ AND\ B.colour =' red'\ AND \\
 & NOTEXISTS \\
 & (SELECT * \\
 & \ FROM\ Sailors\ S1 \\
 & \ WHERE\ S1.rating > S.rating) \\
GROUP\ BY & S.sid
\end{array}
$$

3 Relational Query Optimisation

Generally, each query block is optimised separately (more sophisticated optimisers also "flatten" correlated queries into equivalent join queries and hence consider these alternative forms of the query as well in their search for a good evaluation strategy for the query; similarly, optimisers may covert non-correlated queries into equivalent correlated queries and join queries).

The inner σ, π, $\times$ expression of a query block is optimised first; and then the cheapest method of evaluating the outer GROUP BY, HAVING and π operators (if there are any) is determined.

Optimisation of each σ, π, $\times$ expression consists of three steps:

1. Generate a set of **candidate query plans** for this expression. A query plan is a tree such that:

   - each leaf of the tree is a stored database relation;
   - each inner node of the tree is a relational algebra operator which is applied to the relations that are produced by its chidren;
   - each inner node is annotated with a choice of algorithm for this operator.

   Thus, when a query plan is executed, data flows upwards, from the leaves of the tree, through the inner nodes, all the way up to the root of the tree; the root of the tree is the final query operator and it produces the query result.

2. Estimate the cost of each candidate query plan.

3. Select the cheapest plan.

Generating candidate query plans

Several algebraic laws relating to the relational algebra operators are used to generate candidate query plans in step 1 above, for example the following laws:

(i) Commutativity and associativity of binary operators: natural join, theta join, product, union and intersection are all both **commutative** and **associative**.

(ii) Cascade of projections: if $\{A_1, \ldots, A_n\} \subseteq \{B_1, \ldots, B_n\}$ then

$$\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(E)) = \pi_{A_1,\ldots,A_n}(E)$$

(iii) Cascade of selections, and commutativity of selections:

$$\sigma_{C_1}(\sigma_{C_2}(E)) = \sigma_{C_1 \ AND \ C_2}(E) = \sigma_{C_2 \ AND \ C_1}(E) = \sigma_{C_2}(\sigma_{C_1}(E))$$

(iv) Commuting selection and projection: If C involves only attributes in $A_1, \ldots, A_n$ then

$$\pi_{A_1,\ldots,A_n}(\sigma_C(E)) = \sigma_C(\pi_{A_1,\ldots,A_n}(E))$$

(v) Commuting selection with product: If C_1 involves only attributes of E_1 and C_2 only attributes of E_2 then

$$\sigma_{C_1 \ AND \ C_2}(E_1 \times E_2) = \sigma_{C_1}(E_1) \times \sigma_{C_2}(E_2)$$

(vi) Converting a selection over a product into a join: if C involves comparisons of attributes from E_1 and E_2 then

$$\sigma_C(E_1 \times E_2) = E_1 \bowtie_C E_2$$

Two heuristics are generally applied during the generation of candidate query plans:

(a) Convert selections over products into *joins* whenever possible. This is in order to eliminate the very costly product operation.

(b) Consider only **left-deep join orders**; this is in order to reduce the total number of possible query plans that need to be generated and costed: left-deep join orders take the form $(\ldots((R_1 \bowtie R_2) \bowtie R_3) \ldots \bowtie R_n)$.

(other possible join orders are "right-deep" e.g. $R_1 \bowtie (R_2 \bowtie (R_3 \bowtie R_4))$, and "bushy" e.g. $(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$).

Rule-based vs cost-based optimisation

There are two main approaches to generating and selecting query plans: **rule-based** and **cost-based**. (In this course we focus on the latter.)

Rule-based optimisation uses a predefined ranked order of all possible access paths and algorithms supported by the DBMS, to guide the generation and selection of candidate query plans.

Cost-based optimisation considers all the candidate query plans and uses statistics maintained by the DBMS relating to the current database state in order to estimate the cost of each candidate query plan.

Cost-based optimisation is generally at least as effective as rule-based optimisation (and generally better as it makes use of database statistics). However, it is more expensive, and becomes prohibitively so for queries involving a large number of relations, for which rule-based optisation is used.

How is cost-based optimisation done ?

In our discussion below of how the relational operators are implemented, we will see how an **analytical cost formula** can be derived for each algorithm and access path supported by the DBMS.

The DBMS can also dynamically maintain statistics regarding the volume and distribution of data in the database. These statistics include:

- for each relation, the number of records and the number of pages in the physical file storing its data;
- for each index, the number of key values and the number of pages; the structure of the index e.g. hash file or B+ tree (in which case its height is also recorded); whether the index is clustered or not; the minimum and maximum key values present in the index; and possibly a histogram recording the actual distribution of key values for the index's search key.

These numbers are substituted into the analytic cost formulae to estimate the cost of each operation in the query plan, starting from the "leaves" of the plan and moving upwards. Also estimated are the *size* of the relation output by the operation, and the *sort order* of this relation (if it is produced in a sorted order).

These costs, sizes, and sort orders are recursively fed into operations higher up the query plan, starting at the leaves of the plan and terminating at the root; and in this way the cost of the whole plan is estimated.

Note that it may be possible to *pipeline* the output of one operation into the next operation upwards in the query tree, without storing intermediate results in a temporary table on disk. Pipelining has a lower cost than storing (*materialising*) intermediate results and it is therefore used whenever possible.

4 Implementing the Relational Algebra Operators

In what follows, suppose we have two relations R and S. There are N_R pages in relation R, with p_R rows per page; and N_S pages in relation S, with p_S rows per page.

(a) union : $R \cup S$

If duplicates do not need to be removed from the result, then this operation can be implemented by:

(i) reading and outputing R;

(ii) reading and outputing S.

By convention, query **cost formulae** do not include the cost of writing out the output to disk, so the cost of the above is just the cost of reading R plus the cost of reading S.

For this course, we assume a simple query cost model which considers the cost of a query to comprise just the number of I/O operations required for its evaluation.

This is a reasonable assumption for database systems where the data does not all fit into memory, since disk accesses are much slower than in-memory operations.

Thus, the cost of (i) + (ii) above is $N_R + N_S$ I/Os.

(**Aside**: Real DBMSs do generally take memory access costs and CPU costs into account as well in their query cost models. Also, they may break down an I/O operation into the disk seek time and the block transfer time for finer-grain modelling of I/O costs.

Main-memory DBMSs do not consider I/O costs at all in their query cost models; and they use fine-grained modelling of memory access operations, simple comparison/arithmetic operations, evaluation of more complex user-defined functions etc.)

If duplicates *do* need to be removed from the result of a union then this can be implemented by:

(i) sorting R (on the combination of all its attributes);

(ii) sorting S similarly;

(iii) reading the sorted R and S in parallel, and merging them.

(**Aside**: Appendix A discusses how relations are sorted in DBMSs — you don't need to remember all the details discussed there. The volumes of data involved mean that the data to be sorted may not all fit into main memory, in which case conventional, main-memory oriented, sorting algorithms are not adequate. The *external merge sort* algorithm described in Appendix A uses the DBMS's buffer (in main memory) to incrementally sort a file over several *passes*. On each pass, the partially sorted file is read into B buffer pages, and 1 buffer page is used to construct each sorted output page, which is written back to disk, ready to be read in again in the next pass of the algorithm.

For a file of N pages, $\lceil log_B(N) \rceil$ passes are required in order to completely sort the file. On each pass, the whole file is read in and written out again. So the total cost of sorting the file is

$$2N(\lceil log_B(N) \rceil)$$

I/O operations.)

The cost of steps (i)-(iii) above is therefore

$$2N_R(\lceil log_B(N_R) \rceil) + 2N_S(\lceil log_B(N_S) \rceil)$$

to sort R and S, plus

$$N_R + N_S$$

to read the sorted relations and merge them (again note that the cost of actually writing out the final output to disk is not included).

If one or both of R and S are already appropriately sorted, then of course they do not need to be sorted in (i) and (ii) above, and this cost is not incurred.

(b) difference : $R - S$

(i) sort R;

(ii) sort S;

(iii) read the sorted R and S and output the rows in R that are not in S.

Assuming B input buffer pages, the cost is

$$2N_R(\lceil log_B(N_R) \rceil) + 2N_S(\lceil log_B(N_S) \rceil) + N_R + N_S$$

(c) projection, $\pi_{A_1,\ldots,A_n}R$

If duplicates do not need to be removed from the result, then R can be read and the $A_1, \ldots, A_n$ fields of each of its records can be output. The cost is just

$$N_R$$

Suppose duplicates *do* need to be removed. If there is a B+ tree index on R whose search key contains $A_1, \ldots, A_n$ as a **prefix** (i.e. the search key is $A_1, \ldots, A_n, B_1, \ldots, B_m$ for 0 or more further attributes $B_1, \ldots, B_m$), then

(i) retrieve the records of R in the order of the search key,

(ii) check consecutive records for duplicates over the attributes $A_1, \ldots, A_n$,

(iii) output each distinct n-tuple of values of $A_1, \ldots, A_n$

If the B+ tree is a clustered index, the cost is just

$$N_R$$

to read the N_R leaf pages.

If the B+ tree is an unclustered index, the cost is again just

$$N_R$$

to read the N_R leaf pages, because the necessary attributes $A_1, \ldots, A_n$ are part of the key values stored in the leaf pages (and the remaining attributes of the records are not needed in order to compute this projection operation). This is an example of an **index-only** evaluation, where an unclustered index is just as efficient as a clustered index.

If such an index is not available, a variation of the external merge sort algorithm can be used:

- in the first pass of the algorithm, all record fields other than $A_1, \ldots, A_n$ are eliminated from the records being read in;
- in subsequent passes of the algorithm, duplicates are removed as sorted pages of records are written out.

The cost is similar to that of sorting R, except that the number of pages being generated at each pass of the sort may be fewer than N_R due to the elimination of record fields other than $A_1, \ldots, A_n$ in the first pass, and the subsequent elimiation of duplicates over these fields.

(d) selection, $\sigma_C(R)$

(d.i) simple selection conditions

Suppose C is of the form *attr op value* e.g.

$$SELECT\ *\ FROM\ Sailors\ WHERE\ Sailors.rating = 10$$

If there is a **B+ tree index** on *attr* (e.g. on 'rating' in the above example) then this index can be used to find the qualifying rows.

The cost of a selection $\sigma_{A=v}R$ assuming a **clustered** B+ tree index of height h_R on attribute A of R is approximately:

$$(h_R - 1) + \lceil selectivity(A = v, R) \times N_R \rceil$$

where $h_R - 1$ is the number of inner nodes that have to be read before reaching the leaf nodes of the index, and

$$selectivity(A = v, R)$$

denotes the percentage of rows of R whose A attribute has value v.

If the B+ tree index is **unclustered**, the cost is:

$$(h_R - 1) + \lceil selectivity(A = v, R) \times N_R \rceil + \lceil selectivity(A = v, R) \times N_R \times p_R \rceil$$

where p_R is the number of records per leaf page of the index.

More generally, the cost of a selection C of the form A *op* v, where *op* is $=$, $<$, $>$, $\leq$ or $\geq$, assuming a **clustered B+ tree index** of height h_R on attribute A of R is:

$$(h_R - 1) + \lceil selectivity(C, R) \times N_R \rceil$$

With an **unclustered B+ tree index** the cost is:

$$(h_R - 1) + \lceil selectivity(C, R) \times N_R \rceil + \lceil selectivity(C, R) \times N_R \times p_R \rceil$$

If *op* is the equality operation, $=$, then a **hash index** on attribute A of R could also be used to support this operation, if available:

The cost of a selection $\sigma_{A=v}R$ assuming a (clustered) hash index on attribute A of R is:

$$\lceil selectivity(A = v, R) \times N_R \rceil$$

If a suitable index is not available, then the whole of R has to be read and the qualifying rows output, at a cost of

$$N_R$$

(d.ii) conjunctions of simple selection conditions (AND)

For example, $SELECT * FROM\ Reserves\ R\ WHERE\ R.day < 8/9/2015\ AND\ R.bid = 5\ AND\ R.sid = 20$.

The general approach to evaluating such a selection is to:

1. determine the **cheapest access path** i.e. the index or file scan that is estimated to require the fewest I/Os; and
2. retrieve records using this access path, and then check each of them for the remaining selection conditions.

Example. Consider the condition $day < 8/9/2015\ AND\ bid = 5\ AND\ sid = 20$. Suppose there is a clustered B+ tree index of height 2 on *Reserves* over *day*, and an unclustered B+ tree index of height 2 over (bid, sid). Suppose also that the *Reserves* data consists of 200 pages each containing 100 tuples.

The clustered B+ tree index over *day* can be used to retrieve the rows satisfying $day < 8/9/2015$, and those rows that also satisfy $bid = 5\ AND\ sid = 20$ can be output.

Alternatively, the unclustered B+ tree index over (bid, sid) can be used to retrieve the rows satisfying $bid = 5\ AND\ sid = 20$, and those rows also satisfying $day < 8/9/2015$ can be output.

How does the DBMS determine which of these two alternatives is the cheapest?

It needs to know the selectivity of the respective indexes i.e. what percentage of the rows of the *Reserves* relation satisfy $day < 8/9/2015$ and what percentage satisfy $bid = 5\ AND\ sid = 20$.

How are selectivities estimated?

Simple approach:

$$selectivity\ =\ \frac{|retrieval\ range|}{|max\ value - min\ value|}$$

For example, suppose that in the current *Reserves* relation

- the latest reservation date in the *day* column is 31/12/2015,
- the earliest reservation date in the *day* column is 1/9/2015,
- the *bid* values are in the range $1 \ldots 100$
- the *sid* values are in the range $1 \ldots 50$

For the condition $day < 8/9/2015\ AND\ bid = 5\ AND\ sid = 20$:

- The B+ tree index on *day* is estimated to select 7/120 of the *Reserves* relation (assuming months are 30 days long).

  Since this index is a clustered one, the cost of the retrieval will be approximately

  $$1 + \lceil (7/120) \times 200 \rceil$$

  $= 1 + 12 = 13$ I/Os.

- The index on (bid, sid) is estimated to select $1/(50 \times 100) = 1/5000$ of the *Reserves* relation.

  Since this index is an unclustered one, the cost of the retrieval will be approximately

  $$1 + \lceil (1/5000) \times 200 \rceil + \lceil (1/5000) \times 200 \times 100 \rceil$$

  $= 1 + 1 + 4 = 6$ I/Os.

  So this access path is cheaper than the clustered B+ tree index on *day*.

This simple approach to estimating the selectivity of a condition works well when the values of a column are evenly distributed over the domain of that column, but it may not work so well if the distribution is uneven. For example, what if there is actually only one row in *Reserves* with a date earlier than 8/9/2015 ?

Then the clustered B+ tree index on *day* would actually only require 2 I/Os, which is cheaper than using the unclustered index on (bid, sid).

A more sophisticated approach to estimating selectivities is therefore to maintain a **histogram** for each column of a relation. This divides the domain of that column into a number of ranges and keeps a count of the number of rows that currently fall within each range. These histograms are updated periodically (not every time an update on the relation occurs, as this would be too costly); some DBMSs update histograms automatically, while others do so in response to an explicit user request.

For example, for the *day* column of the *Reserves* relation, suppose each range in the histogram is for 7 days. Thus, the count for the first range $1/9/2015 - 7/9/2015$ would be 1, and the cost of the B+ tree index on *day* would be estimated more accurately as 2 I/Os.

(d.iii) disjunctions of conditions (OR)

For example $SELECT * FROM \ Reserves \ R \ WHERE \ R.day < 8/9/2015 \ OR \ (R.bid = 5 \ AND \ R.sid = 20)$.

If every sub-condition of the disjunction has a matching index, then the query can be transformed into two subqueries whose results are unioned together. For the above example, the transformed query would be:

$$
\begin{aligned}
&(SELECT * FROM \ Reserves \ R \\
&WHERE \ R.day < 8/9/2015) \\
&UNION \\
&(SELECT * FROM \ Reserves \ R \\
&WHERE \ (R.bid = 5 \ AND \ R.sid = 20))
\end{aligned}
$$

This allows the each subquery to be efficiently evaluated using the appropriate matching index.

If there are some sub-conditions of the disjunction for which matching indexes are not available, then it is cheaper to evaluate the original query by one full scan of the table, checking each record against the overall condition (because a full table scan would need to be done in any case for those sub-conditions that do not have matching indexes).

(e) natural join / equi-join of R and S over one join column A

We discuss four alternative join algorithms commonly used in relational DBMS:

(e.i) Nested Loops Join (NLJ):

This algorithm can be used to join any two tables:

for each page, RP, of R do
    for each page, SP, of S do
        output all pairs of records r, s such that
            r is in RP and s is in SP and $r.A = s.A$

The cost of this is $N_R + (N_R \times N_S)$ (N_R to read each page of R and, for each page of R, N_S to read the whole of S).

Thus, note that it is cheaper to make the smaller relation the "outer" one.

How does the DBMS know which relation is smaller? The DBMS keeps a count of the number of pages and records in each relation as part of its database statistics. This information is updated periodically.

(e.ii) Index Nested Loops Join (INLJ):

If there is an index on the A attribute of one of the relations (say S), this can be made the "inner" relation and the index can be exploited:

for each page RP of R do
    for each record r in RP do
        output all pairs of records r, s such that s is in S and $r.A = s.A$

The cost of this is:

$$N_R + (N_R \times p_R \times \textit{cost of finding matching records in } S \textit{ for a record of } R)$$

where for each record r of R, the cost of finding the records in S matching r (i.e. of "probing" S) depends on whether the index on S is a hash or B+tree index, and whether it is clustered or unclustered. The cost formulae are the same as those for a selection with an equality condition (see (d)(i) above).

(e.iii) Sort-Merge Join (SMJ):

This first sorts R and S on the join column A, and then reads the sorted files in parallel in order to combine records that agree on A. The second phase works as follows:

while not (eof(R) or eof(S)) do
    if R-record.$A < S$-record.A then
        get next R-record
    elsif S-record.$A < R$-record.A then
        get next S-record
    else /* R-record.$A = S$-record.A */
        $currentVal$:= R-record.A;
        output all pairs of records r, s with r in R and
            s in S such that $r.A = s.A = currentVal$;
        R-record := the first record in R with $A > currentVal$;
        S-record := the first record in S with $A > currentVal$;
    endif
endwhile

The overall cost of the sort-merge join consists of
(a) the cost of the two original sorts, plus
(b) N_R to scan R, plus
(c) the cost of scanning the matching records of S for the records of R i.e.

$$2N_R(\lceil log_B(N_R) \rceil) + 2N_S(\lceil log_B(N_S) \rceil) + N_R + C$$

Cost C is theoretically anywhere between N_S and $N_R * N_S$, but is likely to be near to N_S in practice.

If there are clustered B+ tree indexes available on either or both of R and S that have the join column A as a prefix of their search key, then there is no need for the original sort since the data records are already in the right sort order.

(e.iv) Hash Join (HJ):

This first partitions R and S using the same hash function h on A. Thus, records in the i^{th} partition of R can have the same A value only as records in the i^{th} partition of S.

The algorithm then reads into memory the first partition P_R of R, creating a hash table T_{P_R} for it using some other hash function, $h2$, also on A.

It then scans the corresponding partition P_S of S on disk and "probes" the hash table T_{P_R} to find the matching records in P_R for each record of P_S.

This process is repeated for the second partition of R and S, the third partition, and so on, to the last partition.

The overall cost is thus $3(N_R + N_S)$, since R and S are both read and written during the partitioning phase, and are then both read again during the matching phase.

(f) equi-join / natural join over more than one join attribute

NLJ can be used for this.

INLJ can be used if there are matching indexes on S over one or more of the join attributes.

SMJ can be used, and requires sorting on all the join attributes.

HJ can be used, and requires hashing on all the join attributes.

(g) theta join

NLJ can be used for this.

INLJ can be used for simple comparisons if there is a clustered B+ tree index on S. (Why not a hash index? Why not an unclustered B+ tree index?)

SMJ and HJ are not applicable (why not ?)

(h) intersection and product

These are special cases of equi-join.

For intersection, the set of join attributes is the whole of the scheme of R and S. All four join algorithms are applicable, as in (f) above.

For product, the set of join attributes is empty, so all rows of R match all rows of S. So which of the 4 join algorithms are suitable for evaluating a product ?

(i) aggregation operations (SUM, MAX, MIN, COUNT, AVG)

If there is no GROUP BY clause in the query, then the relation output by the SELECT ... WHERE ... clause of the query needs to be scanned to compute the aggregation operation. In practice this can be done "on-the-fly" during the last phase of the computation of the SELECT ... WHERE ... clause, not incurring any additional I/O operations. For each aggregation function, some "running information" needs to be maintained during the scan:

COUNT: number of values scanned so far

SUM: sum of the values scanned so far

AVG: number of values scanned so far, and their sum

MIN: minimum value scanned so far

MAX: maximum value scanned so far

If there is a GROUP BY clause in the query, then the relation output by the SELECT ... WHERE ... clause first needs to be sorted on the GROUP BY attributes, and then scanned to compute the aggregation operation for each group. The computation of the aggregation operation can be done "on-the-fly" during the last phase of the sort operation, not incurring any additional I/O operations. The same running information as above is maintained, but in this case it is re-initialised at the beginning of each group.

5 Query Optimisation Exercise

Consider the following query:

$SELECT \quad R.day$
$FROM \quad BoatsB,\ Reserves\ R$
$WHERE \quad B.colour =' red'\ AND\ R.bid = B.bid$

(i) Translate this query into the relational algebra, using the operators π, σ and $\times$.

Answer: $\pi_{R.day}(\sigma_{Reserves.bid=Boats.bid\ AND\ Boats.colour='red'}(Boats \times Reserves))$

(ii) Write down 4 possible query plans for this query, assuming that:

- there is a clustered hash index on *bid* of Boats, and
- there is a clustered B+ tree index of height 3 on (bid, sid) of Reserves.

Answer (there are other possibilities as well):

(a) INLJ of Boats as the outer relation and Reservers as the inner relation, using the index on (bid,sid) of Reserves; followed by selection of red boats and projection on day 'on-the-fly'.

(b) Selection of red boats from Boats, pipelined into INLJ with Reserves as the inner relation, using the index on (bid,sid) of Reserves; followed by projection on day 'on-the-fly'.

(c) INLJ of Reserves as the outer relation and Boats as the inner relation, using the index on bid of Boats; followed by selection of red boats and projection on day 'on-the-fly'.

(d) Selection of red boats from Boats, stored as a Temp file sorted on bid. SMJ of Reserves and Temp (both now sorted on bid), scanning them and doing the projection on day 'on-the-fly'

(iii) Estimate the cost of each of your four query plans, assuming that:

- Boats consists of 4 pages of 25 records per page
- Reserves consists of 200 pages of 100 records per page
- *bid* values are in the range $1 \ldots 100$
- One quarter of boats are red
- There is a uniform distribution of *bid* values in Reserves, and one quarter of boat reservations are for red boats.

Answer:

(a)

```
Read the Boats relation => 4 I/Os

For each of its 100 Boats records,
    look-up the matching Reservations records in Reserves using the
    clustered B+ tree index on (bid,sid).

    There will be 20000/100 = 200 Reservation records per boat,
    stored on about 2 leaf pages of the B+ tree (since this is
    clustered).

    Hence a cost of 2 I/Os [to read the inner nodes - recall that
                            the height of the B+ tree is 3]
               and  2 I/Os [to read the 2 leaf nodes]
    for each of the 100 boats.
    => 100 x 4 = 400 I/Os

The selection and projection are done on-the-fly as the rows
from this join are produced, and hence incur no I/O cost.

Total cost: 404 I/Os
```

(b) Read the Boats relation => 4 I/Os

For each record, check the selection condition colour='red'.

There will be approximately 100/4 = 25 such records.

For each such record,
look-up the matching Reservations records in Reserves using the clustered B+ tree index on (bid,sid).

There will be 20000/100 = 200 Reservation records per boat, stored on about 2 leaf pages of the B+ tree (since this is clustered).

Hence a cost of 2 I/Os [to read the inner nodes]
and 2 I/Os [to read the 2 leaf nodes]
for each of the 25 red boats.
=> 25 x 4 = 100 I/Os

Do the final projection on-the-fly as the rows of this join are produced.

Total cost: 104 I/Os

(c) Read the Reserves relation => 200 I/Os

For each of its 20,000 Reservation records,
look-up the matching Boat record in Boats using the hash index on bid.

Boats is clustered on bid, and Reserves is ordered on (bid,sid).
So the relevant page of Boats will be in the buffer for each of the 100 different bid values that are successively encountered in the leaves of Reserves.

=> 100 I/Os

The selection and projection are done on-the-fly as the rows from this join are produced, and hence incur no I/O cost

Total cost: 300 I/Os

(d)
```
Read Boats => 4 I/Os

Write the approximately 25 red boat records into Temp file
after first sorting these records (in memory) => 1 I/O

Now do the SMJ: scan Reserves and Temp concurrently
(both will be sorted in bid order), joining pairs of
rows that match on their bid value
=> 200 I/Os to read Reserves
=> 1 I/O to read Temp

Do the final projection on-the-fly as the rows of this
join are produced.

Total cost: 206 I/Os
```

Exercise for class: What would be the costs of plans (a)-(d) using different join algorithms e.g. HJ and SMJ for plans (a), (b), (c), HJ for (d).

6 Optimising Nested queries

Subqueries are typically optimised independently from their parent queries. Non-correlated subqueries need only be evaluated once. For correlated subqueries, their parent query provides a selection condition within the subquery; and correlated subqueries need to be evaluated once for each different value passed to them from their parent query.

It is sometimes possible to "flatten" nested queries into equivalent join queries, and some optimisers try to do that. This is advantageous because there may be more efficient query plans generated by the optimiser than if it considers only the nested form of the query, in which relations in the parent query are always treated as the "outer" relation when being joined with relations in the inner query.

7 Plan Execution

The implementations of the relational operators forming the nodes of a query plan each support an Iterator interface. This encapsulates the implementation details of each operator, algorithm and access method.

The Iterator interface includes the functions `open`, `get_next` and `close`:

- `open` initialises the state of the Iterator, allocating buffers for its input and its output, and passing parameteters to the Iterator such as selection and join conditions, projection attributes, group-by attributes etc.

- `get_next` is called by an operator node whenever it requires more input tuples from one of its children nodes; the operator then executes operator-specific code to process these input tuples; and the output tuples it generates are placed in its output buffer.

  The decision to pipeline or materialise the input tuples is encapsulated in the operator-specific code that processes input tuples: if the operator can completely process the input tuples as it receives them, then it doesn't need to materialise them; if it needs to examine the same input tuples several times, then they need to be materialised.

- `close` is called by an operator when all input from a child node has been consumed. Similarly, its own `close` is called by its parent node it when it can produce no more tuples.

Appendix A - External Sorting

Why is sorting important in databases ?

- Users frequently require data to be presented in a sorted order c.f. ORDER BY clause in SQL.
- Sorting is useful for removing duplicate rows from a collection c.f. DISTINCT in SQL.
- The sort-merge algorithm for implementing the join operation requires sorting. Several of the other relational algebra operations also require sorting.

The volumes of data involved mean that conventional "internal" sorting algorithms are not suitable because they do not use information about the amount of main memory available for the sort and hence may cause a lot of I/O in cases where not all of the data fits into main memory.

External merge sort is a standard sorting technique used in DBMS. We give a simple version of it first, that uses 2 input buffer pages and one output buffer page. The general version for B input buffer pages is given next.

Two-way external merge sort

The problem: sort a file of N pages using 2 input buffer pages and 1 output buffer page.

The algorithm is as follows:

Pass 0: Read two pages of the file at a time into the 2 input buffer pages. Sort each pair of pages (using an internal sorting algorithm such as quicksort, bubble sort etc.) and output the result.

This produces $\lceil N/2 \rceil$ sorted groups of 2 pages each[1].

Pass 1: Read the pairs of pages output by Pass 0 two at a time into the input buffer pages. Merge them into 4 new sorted pages using the output buffer page and output the result.

This produces $\lceil N/2^2 \rceil$ sorted groups of 4 pages each.

Pass 2: Read the groups of 4 pages output by Pass 1 two at a time into the input buffer pages. Merge them into 8 new sorted pages using the output buffer page and output the result.

[1]The notation $\lceil e \rceil$ denotes the smallest integer greater than or equal to e.

This produces $\lceil N/2^3 \rceil$ sorted groups of 8 pages each.

...

Pass i: Read the groups of 2^i pages output by Pass $i-1$ two at a time into the input buffer pages. Merge them into 2^{i+1} new sorted pages using the output buffer page and output the result.

This produces $\lceil N/2^{i+1} \rceil$ sorted groups of 2^{i+1} pages each.

...

The sort terminates when $2^{i+1} \geq N$ i.e. when $i = \lceil log_2 \lceil N/2 \rceil \rceil$.

Thus, the total number of passes is $1 + \lceil log_2 \lceil N/2 \rceil \rceil = \lceil log_2(N) \rceil$.

On each pass, each page of the file is read in and written out. Thus the total cost (in terms of I/O operations) is

$$2N(\lceil log_2(N) \rceil)$$

For example, the cost of sorting a file of 1,000,000 pages is

$$2,000,000 \times (\lceil log_2(1,000,000) \rceil) = 2,000,000 \times 20 = 40,000,000$$

I/O operations.

General external merge sort

The problem: sort a file of N pages, using B input buffer pages and 1 output buffer page.

The algorithm is as follows:

Pass 0: Read B pages of the file at a time into the B input buffer pages. Sort each group of pages (using an internal sorting algorithm) and output the result.

This produces $\lceil N/B \rceil$ sorted groups of B pages each.

Pass 1: Read the groups of B pages output by Pass 0 B at a time into the input buffer pages. Merge each group into B^2 new sorted pages using the output buffer page and output the result.

This produces $\lceil N/B^2 \rceil$ sorted groups of B^2 pages each.

Pass 2: Read the groups of B^2 pages output by Pass 1 B at a time into the input buffer pages. Merge each group into B^3 new sorted pages using the output buffer page and output the result.

This produces $\lceil N/B^3 \rceil$ sorted groups of B^3 pages each.

...

Pass i: Read the groups of B^i pages output by Pass $i-1$ B at a time into the input buffer pages. Merge each group into B^{i+1} new sorted pages using the output buffer page and output the result.

This produces $\lceil N/B^{i+1} \rceil$ sorted groups of B^{i+1} pages each.

...

The sort terminates when $B^{i+1} \geq N$ i.e. when $i = \lceil log_B \lceil N/B \rceil \rceil$.

Thus, the total number of passes is $1 + \lceil log_B \lceil N/B \rceil \rceil = \lceil log_B(N) \rceil$.

On each pass, each page of the file is read in and written out. Thus the total cost is

$$2N(\lceil log_B(N) \rceil)$$

For example, the cost of sorting a file of 1,000,000 pages using 100 input buffer pages is

$$2,000,000 \times (\lceil log_{100}(1,000,000) \rceil) = 2,000,000 \times 3 =$$

6,000,000 I/O operations (compare with the cost using 2 input buffers).

Using 1000 input buffer pages it would be:

$$2,000,000 \times (\lceil log_{1000}(1,000,000) \rceil) = 2,000,000 \times 2 = 4,000,000$$

Appendix B - Query Optimisation in Oracle

Oracle has extensive Query Optimisation capabilities. Information about these can be found in the "Oracle Database Performance Tuning Guide"

Oracle uses the syntax "EXPLAIN PLAN FOR SQL_statement" to display the selected execution plan for an SQL statement. The selected plan is stored in a global table called PLAN_TABLE, by default.

The query
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY());
displays the stored plan, as well as the estimated cost of the plan.

Oracle can store query execution plans for SQL statements with trusted performance characteristics — these are termed baseline plans. When the query optimiser generates an execution plan for an SQL statement, it checks whether there exist any baseline plans for that statement. If the new plan is different from all the baseline plans, the optimiser selects what it considers to be the best plan; it also adds the new plan to the plan history for that statement; subsequently this may be added to the set of baseline plans for the query.

Automatic gathering of database statistics for the optimiser can be enabled or disabled. It is also possible to use the DBMS_STATS package to gather statistics, setting specific preferences for the statistics gathering (e.g. sampling level, when to update the statistics, which columns to maintain histograms for, the bucket-size of the histograms). The current statistics held about tables, indexes and columns can be viewed, including the content of the histograms.

Homework Reading (optional)

Relational DBMSs generate cadidate query plans using an approach based on *dynamic programming* that was pioneered by the System R optimiser, an early and very influential relational DBMS prototype developed by IBM.

The principle of dynamic programming is that optimal solutions to sub-problems can be used to find good solutions to an overall problem.

In the context of query optimisation, this is done "bottom-up", starting with single-table query plans for each of the n individual tables being joined in the query.

A second table is added to each of these single-table plans and the best two-table query plans are retained.

A third table is added to each of these two-table plans and the best three-table query plans are retained.

The process continues in this fashion until the best $n-1$-table query plans have been generated. The final table is added to each of these, and the cheapest n-table query plan is retained as the final overall query plan.

Pages 497-503 of Ramakrishnan & Gerhke has a more detailed description of this process.