Query Optimization

- **Process of selecting the most efficient query evaluation plan**

- **Specially meaningful for complex query**

Two solutions
- **Relational Algebra level**
  - **convert expression that takes less execution time**

- **Detailed strategy for processing the query**
  - **choosing a specific indices**
  - **choosing an algorithm for a specific task**

In relational algebra (RA) a query is looked like as follows :

$$\Pi_A (\sigma_{B = C \wedge D = 99} (AB \times CD))$$

Several kinds of optimisation strategies are used to optimise queries :

1. Algebraic manipulations reduces the running time of queries. For example, for the above query, if we migrate the selection D = 99 inside the cartesian product we get the following expression :

   $$\Pi_A (\sigma_{B = C} (AB \text{ X } \sigma_{D = 99} (CD)))$$

Most importantly the cartesian product and selection for B = C convert to an equijoin. It becomes as follows :

$$\Pi_A (AB \underset{B = C}{\text{X}} \sigma_{D = 99} (CD))$$

2. Efficient usage of indices and other facts about the file organisation may be considered as another way to optimise queries. The organisation of the file holding relation may be chosen appropriately to reduce the block access required to answer

the query. If the file is already sorted or indexed, the lesser number of block access will be required.

General Strategies for Optimisation

1. Perform selection as early as possible

2. Combine sequences of unary operations, such as selections and projections

3. Look for common subexpressions in an expression. Subexpression involving a join that cannot be modified by moving a selection inside it generally falls in this category.

The following are recurrent themes concerning the physical implementation of relations and optimisation of queries.

Preprocess files appropriately : Sorting or indexing while it may not be economical to maintain an index permanently or keep the file sorted (because cost of insertions goes up), it may make sense to create an index or to sort temporarily in response to a query.

Evaluate options before compute :

Laws involving joins and cartesian products

1. Commutative laws for joins and products

$$E_1 \underset{F}{\bowtie} E_2 \equiv E_2 \underset{F}{\bowtie} E_1$$

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

$$E_1 \mathrm{X} E_2 \equiv E_2 \mathrm{X} E_1$$

2. Associative laws for joins and products

$$(E_1 \underset{F_1}{\bowtie} E_2) \underset{F_2}{\bowtie} E_3 \equiv E_1 \underset{F_1}{\bowtie} (E_2 \underset{F_2}{\bowtie} E_3)$$

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

$$(E_1 \mathrm{X} E_2) \mathrm{X} E_3 \equiv E_1 \mathrm{X} (E_2 \mathrm{X} E_3)$$

Laws involving selections and projections

3. Cascade of projections

$$\Pi_{A1 \ldots An} (\Pi_{B1 \ldots Bm} (E)) \equiv \Pi_{A1 \ldots An} (E)$$

Note that the attribute names $_{A1 \ldots An}$ must be among the B_i's for the cascade to be legal.

4. Cascade of selection

$$\sigma_{F1} (\sigma_{F2} (E)) \equiv \sigma_{F1 \wedge F2} (E)$$

Since F1 ^ F2 = F2 ^ F1, it follows immediately that selections can be commuted, i.e.

$$\sigma_{F1} (\sigma_{F2} (E)) \equiv \sigma_{F2} (\sigma_{F1} (E))$$

5. Commuting selections and projections. If condition F involves only attributes A1 … An, then

$$\Pi_{A1 \ldots An} (\sigma_F (E)) \equiv \sigma_F (\Pi_{A1 \ldots An} (E))$$

More generally, if condition F also involves attributes B_1, … Bm that are not among A1 … An, then

$$\Pi_{A1 \ldots An} (\sigma_F (E)) \equiv \Pi_{A1 \ldots An} (\sigma_F (\Pi_{A1 \ldots An\, ,\, B1, \ldots Bm} (E))$$

6. $\sigma_F (E_1 X E_2) \equiv \sigma_F (E_1) X E_2$

As a useful corollary, if F is of the form F1 ^ F2, where F1 involves only attributes of E_1, and F2 involves only attributes of E_2 we obtain the following:

$$\sigma_F (E_1 X E_2) \equiv \sigma_{F1} (E_1) X \sigma_{F2}(E_2)$$

7. $\sigma_F (E_1 \cup E_2) \equiv \sigma_F (E_1) \cup \sigma_F(E_2)$

8. $\sigma_F (E_1 - E_2) \equiv \sigma_F (E_1) - \sigma_F(E_2)$

We shall not state the laws for pushing a selection ahead of a join, since a join can always be expressed as a cartesian product followed by a selection, and for natural join a projection.

9. Commuting a projection with a CP. Let E1 and E2 be two relational expressions. Let A1, A2, … , An be a list of attributes, of which B1 … Bm are attributes of E1, and the remaining attributes C1, … Ck are from E2. Then

$$\Pi_{A1 \ldots An} (E_1 X E_2) \equiv \Pi_{B1, \ldots Bm} (E_1) \; X \; \Pi_{c1, \ldots ck} (E_2)$$

10. Commuting a projection with a union

$\Pi_{A1 \dots An} (E_1 \cup E_2) \equiv \Pi_{A1, \dots, An} (E_1) \cup \Pi_{A1, \dots, An} (E_2)$

Example :

A library database consisting of the following relations :

> BOOK(title, author, p_name, lc_no)
> PUBLISHER(p_name, p_addr, p_city)
> BORROWER(name, addr, city, card_no)
> LOAN(card_no, lc_no, date)

To keep track of books, we might suppose that there is a view XLOANS that contains additional information about books borrowed. XLOANS is the natural join of BOOK, BORROWER and LOAN which may be defined as follows :

$\Pi_s (\sigma_F (\text{LOAN X BORROWER X BOOK}))$

where F : BORROWER . card_no = LOAN . card_no AND BOOK . lc_no = LOAN . lc_no

while s = title, author, p_name, lc_no, name, addr, city, card_no, date

Here the objective is to list the books that been borrowed before a date in the distant past say 1/1/2005

$\Pi_{title}\, \sigma_{date < 1/1/2005}$ (XLOAN)

Parse tree will be as follows :

Π_{title}

|

|

σ title $\sigma_{date < 1/1/95}$

|

|

Π title, author, book.lc_no, name, addr, city, borrower.card_no, date

|

```
                 |
  σ BOOK . lc_no = LOAN. Lc_no  ∧ BORROWER. Card_no = LOAN . card_no
                 |

               X
            /     \
           /        \
          X          BOOK
        /   \
       /      \
   LOAN    BORROWER
```

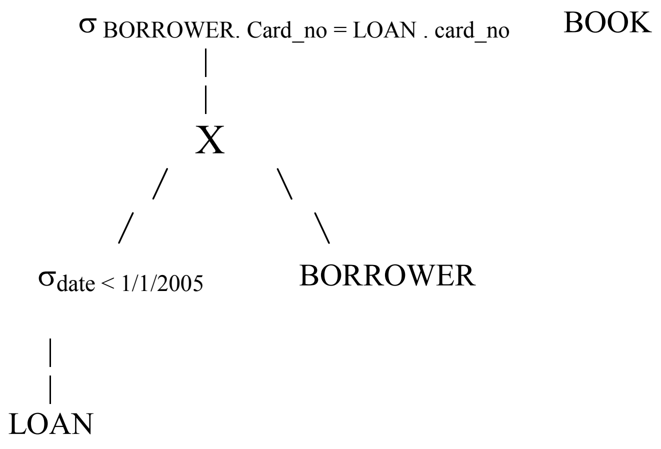
Parse tree of the expression (1)

The first step of the optimisation is to split the seection F into two, with conditions BOOK. lc_no = LOAN. lc_no and BORROWER. card_no = LOAN. card_no respectively. Then we move each of the three selections as far as down the tree as possible. The selection $\sigma_{date < 1/1/95}$ moves below the projection and the two selections. This selection then applies to the product (LOAN X BORROWER) X BOOK. Since date is the only attribute mentioned by the selection, and date is an attribute only of LOAN, we can replace $\sigma_{date < 1/1/95}$ ((LOAN X BORROWER) X BOOK) by
($\sigma_{date < 1/1/95}$ (LOAN)) X BORROWER) X BOOK

We have now moved this selection as far down as possible. The selection with condition BOOK. lc_no = LOAN. lc_no cannot be moved below either cartesian product, since it involves an attribute of BOOK and an attribute not belonging to BOOK. However, the selection on BORROWER. card_no = LOAN. card_no can be moved down to apply to the product

$\sigma_{date < 1/1/95}$ (LOAN) X BORROWER

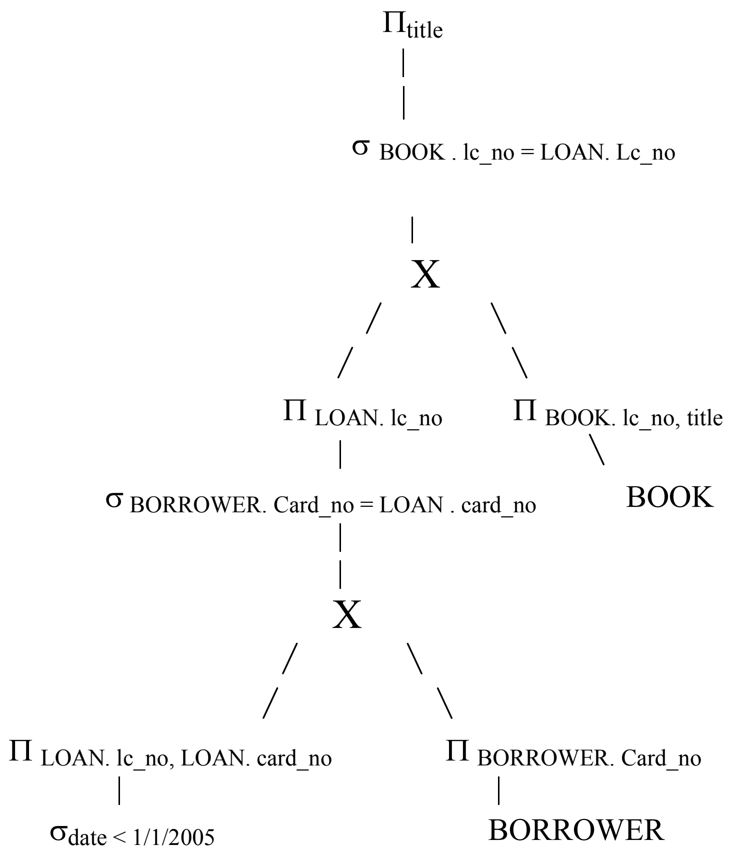
```
              Π title
              |
              |
        σ BOOK . lc_no = LOAN. Lc_no

                |
                X
             /     \
              /        \
```



(2)

Then we can replace Π_{title} and $\sigma_{BOOK.\,lc\_no = LOAN.\,Lc\_no}$ by the cascade

$\Pi_{title,\ BOOK.\,lc\_no,\ LOAN.\,lc\_no}$. Now $\Pi_{title,\ BOOK.\,lc\_no}$ is applied to BOOK and $\Pi_{LOAN.\,lc\_no}$ is applied to the left operand of the higher CP in the above figure.


Πtitle
σ BOOK . lc_no = LOAN. Lc_no
X
Π LOAN. lc_no
Π BOOK. lc_no, title
σ BORROWER. Card_no = LOAN . card_no
BOOK
X
Π LOAN. lc_no, LOAN. card_no
Π BORROWER. Card_no
σdate < 1/1/2005
BORROWER

|
LOAN

Final tree with grouping of operators (3)

Cost based optimization

- **Query optimizer comes up with a query evaluation plan**
  - **computes the same result**
  - **least costly way of generating the result**

- **Computing the precise cost of a evaluation plan is usually not possible without actually evaluating the plan**

Cost components for query execution

- **access cost to secondary storage**
- **storage cost**
- **computation cost**
- **communication cost**

access cost to secondary storage – cost of searching, reading and writing data blocks that reside in secondary storage

storage cost – cost of storing any intermediate files that are generated by execution of query

computation cost – cost of performing in-memory operations during query execution e.g. searching, sorting, merging

communication cost – cost of shipping the data e.g. Distributed database

- **Disk access dominates over memory access**

- **Cost is an estimate – selected plan is not necessarily the least costly plan**

How cost can be estimated before actual evaluation ?

- **Optimizers make use of statistical information about the relations**
  - **relation size**
  - **index depth**

- **DBMS catalogue stores the following statistics**
  - **no. of tuple in a relation**
  - **no. of blocks containing a relation**
  - **size of a tuple in bytes**
  - **statistics about indices**
    - **e.g. height of B^+ tree**
    - **no. of leaf pages**

If we wish to maintain accurate statistics, then, every time a relation is modified we must also update the statistics. This update incurs a substantial amount of overhead. Thus most systems do not update the statistics on every modification. Instead, they update the statistics periodically. As a result, the statistics used for choosing a query processing strategy may not be completely accurate.

Join Algorithms

Assumptions for performing depositor X customer

depositor (customer_name, ac_no)
customer (customer_name, customer_street,cutomer_city)

- no. of records of customer $n_{customer}$ = 10,000
- no. of blocks of customer $b_{customer}$ = 400
- no. of records of depositor $n_{depositor}$ = 5000
- no. of blocks of depositor $b_{depositor}$ = 100

Block Nested loop join algorithm

- **examines every pair of tuples in the two relations**
- **expensive**

Algorithm

```
for each block $B_r$ of r do begin
    for each block $B_s$ of s do begin
        for each tuple $t_r$ in $B_r$ do begin
            for each tuple $t_s$ in $B_s$ do begin
              test pair ($t_r$, $t_s$) to see if they satisfy the join condition
              if they do, add $t_r$ . $t_s$ to the result
            end
        end
    end
end
```

Worst case

- **two memory buffers**
- **no. of block access –** $b_r + b_r * b_s$

Clearly, it is more efficient to use the smaller relation as the outer relation, in case neither of the relations fits in memory.

For the example join total no. of block access – 100 * 400 + 100 = 40,100

Best case

- **enough no. of buffers for both relations to fit simultaneously in memory**
- **no. of block access --** $b_r + b_s$

For the example join total no. of block access – 100 + 400 = 500

Merge Join

Pr	a1	a2
→	a	3
	b	1
	d	8
	d	13
	f	7
	m	5
	q	6

r

ps	a1	a3
→	a	A
	b	G
	c	L
	d	N
	m	B

s

Algorithm

```
  $pr$ := address of 1st tuple of r;
  $ps$ := address of 1st tuple of s;
      while ($ps \neq$ null and $pr \neq$ null) do
begin
          $t_s$ := tuple to which $ps$ points;
          $S_s$ := $\{t_s\}$;
          set $ps$ to point to next tuple of s;
          $done$ := false;
          while (not done and $ps \neq$ null) do
             begin
                $t_s'$ := tuple to which $ps$ points;
                if ($t_s'$[joinattrs] = $t_s$[joinattrs])
                   then begin
                         $S_s := S_s \cup \{ t_s' \}$;
                         set $ps$ to point to next tuple of s;
                        end
                  else $done$ := true;
                end
             $t_r$ := tuple to which pr points;
   while ($pr \neq$ null and $t_r$[joinattrs] < $t_s$[joinattrs]) do
        begin
            set $pr$ to point to next tuple of r;
            $t_r$ := tuple to which pr points;
        end
  while ($pr \neq$ null and $t_r$[joinattrs] = $t_s$[joinattrs]) do
        begin
            for each $t_s$ in $S_s$ do
                begin
                  add $t_s$ NJ $t_r$ to result;
                end
            set $pr$ to point to next tuple of r;
            $t_r$ := tuple to which $pr$ points;
        end
end.
```

Since the relations are in sorted order, tuples with the same value on the join attributes are in consecutive order. Thereby, each tuple in the sorted order needs to be only once, and, as a result, each block is also read only once. Since it makes only a single pass through both the files, the no. of block accesses = $b_r + b_s$.

Case 1
For the said example, if the relations are already sorted on the join attribute customer_name, the merge join takes a total of 400 + 100 = 500 block accesses.

Case 2
If the relations are not sorted and the memory size in the worst case is 3 blocks, the following additional block accesses are required :
For sorting relation customer 400 * (2 ul $\log_2$(400/3)ul + 1) = 6800 blocks
For writing the result (customer) 400 blocks
For sorting relation depositor 100 * (2 ul $\log_2$(100/3)ul + 1) = 1300 blocks
For writing the result (depositor) 100 blocks

Case 3
If the memory size is 25 blocks
For sorting relation customer 400 * (2 ul $\log_2$(400/25)ul + 1) = 1200 blocks

For sorting relation depositor 100 * (2 ul $\log_2$(100/25)ul + 1) = 300 blocks

Hash Join

- **Two passes**
    - **1st pass through the relation with fewer records (say R)**

- **Hashes its records to the hash file buckets**
- **2nd pass through the other relation S**
- **Hashes each records to the appropriate bucket**
- **Records combine with all matching records from R**

Algorithm

```
/* 1st pass */
/* build hash buckets H on R.C(common attribute) */

            do j := 1 to n
                k := hash (R[j].C)
                 add R[j] to bucket address H(k)
             end


/* 2nd pass */
/* look up */
             do i  :=  1 to m
                k := hash (S[i].C)
      /* assume there are h tuples R[1], . . , R[h] stored at H[k] */
                  do j := 1 to h
                          if R[j].C = S[i].C then
                          add joined tuple S[i] X R[j] to result
                  end
             end
```

Block access

- **Best case**
- **All the buckets can be kept in main memory**
- **Cost estimate in terms of block access – $b_r + b_s$**