Query Optimization

Very simple, in theory...

1. Write down all possible ways that a query can be performed.
2. For each query execution plan (QEP), calculate the cost (in Disk I/O's)
3. Choose the least expensive

1a. Write down all possible equivalent algebraic formulations of the query
1b. For each equivalent query plan, write a different plan using every possible combination of access method and join algorithm.

Algebraic rules:

1. Cascading Selects: If \( C_1, C_2 \ldots C_n \) are selection conditions, then
   \[
   \sigma_{C_1 \land C_2 \land \ldots \land C_n}(r) = \sigma_{C_1}(\sigma_{C_2}(\ldots(r)\ldots))
   \]

2. Commutativity of Select: If \( C_1 \) and \( C_2 \) are selection conditions, then
   \[
   \sigma_{C_1}(\sigma_{C_2}(r)) = \sigma_{C_2}(\sigma_{C_1}(r))
   \]

3. Cascading projections: If \( X \supsetneq X_n \supsetneq X_{n-1} \ldots \supsetneq X_1 \) are sets of attributes, then
   \[
   \Pi_{X_1}(\Pi_{X_2}(\ldots(\Pi_{X_n}(r))\ldots)) = \Pi_{X_1}(r)
   \]

4. Commutativity of Selection and projection: if all attributes of \( C \) are contained in \( X \), then
   \[
   \Pi_X(\sigma_C(r)) = \sigma_C(\Pi_X(r))
   \]

5. Commutativity of join: If \( C \) is some join condition or \( \emptyset \), then
   \[
   r \bowtie_C s = s \bowtie_C r
   \]

6. Commutativity of selection and join: if all the attributes mentioned in condition \( C \) are part of the schema of table \( r \), then
   \[
   \sigma_C(r \bowtie s) = (\sigma_C(r) \bowtie s)
   \]

   Extending the idea, if \( C = (C_1 \text{ and } C_2) \) and all the attributes of \( C_1 \) are in schema( \( r \) ) and all the attributes of \( C_2 \) are in schema( \( s \) ) then
   \[
   \sigma_C(r \bowtie_C s) = (\sigma_{C_1}(r) \bowtie \sigma_{C_2}(s))
   \]

6a. Commutativity of select with \( \times \).

7. Commutativity of projection with join: if \( X \) is a set of attributes, \( X = A_1 A_2 \ldots A_m B_1 B_2 \ldots B_n \) where \( A_1 A_2 \ldots A_m \) are all in the sch(\( r \)) and \( B_1 B_2 \ldots B_n \) are all in the sch(\( s \)), and all of the attributes mentioned in condition \( C \) are in \( X \) then
   \[
   \Pi_X(r \bowtie_C s) = \Pi_{A_1 A_2 \ldots A_m}(r) \bowtie_C \Pi_{B_1 B_2 \ldots B_n}(s)
   \]
In the event that some set of attributes $Y$ is part of $C$ but not of $X$, then
\[ \Pi_X(\Pi_A \bowtie_C \Pi_B r \bowtie_C s) = \Pi_X(\Pi_{A1A2...AmY} r \bowtie_C \Pi_{B1B2...BnY} s) \]

7b Commutativity of projection with $\times$

8. Commutativity of $\cap$ and $\cup$;

\[ r \cap s = s \cap r \quad r \cup s = s \cup r \]

note that $-$ is not commutative, $r - s \neq s - r$

9. Associativity of set operations: if $\theta$ stands for any of the following: $\cup$, $\cap$, $\times$ or $\bowtie$ then

\[ r \theta (s \theta t) = (r \theta s) \theta t \]

10. Commutativity of $\sigma$ and set operations: if $\theta$ stands for any of the following: $\cap$, $\cup$, or $-$, then

\[ \sigma(r \theta s) = \sigma(r) \theta \sigma(s) \]

11. Commutativity of $\Pi$ and set operations: if $\theta$ stands for any of the following: $\cap$, $\cup$ then

\[ \Pi(r \theta s) = \Pi(r) \theta \Pi(s) \]

Notice that $\Pi(r - s) \neq \Pi(r) - \Pi(s)$

Exercise construct a simple example of relations $r$ and $s$ such that

\[ \Pi(r - s) \supseteq \Pi(r) - \Pi(s) \]

Query Flattening...Rich Ganski Ingres, work based on Kim, W. "On Optimizing an SQL-like Nested Query". ACM Transactions on Database Systems vol 7, no 3, Sept 1982

3.1 Flattening a Type-N or Type-J Nested Query

SELECT r.a
FROM r
WHERE r.b IN
    (SELECT s.c
     FROM s);

is equivalent to

SELECT r.a
FROM r,s
WHERE r.b = s.c;
For a type-J nested query, the correlated join clause can be transferred directly to the single-level query, in a similar transformation:
Query 1:

```
SELECT r.a
FROM r
WHERE r.b IN
  (SELECT s.c
   FROM s
   WHERE s.d = r.d);
```
is equivalent to

Query 2

```
SELECT r.a
FROM r, s
WHERE r.b = s.c AND s.d = r.d;
```

Example. As a very simple example, assume that R and S contain the following rows:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>2</td>
</tr>
</tbody>
</table>

R:

<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
S:

If we step through a nested-iteration execution of the type-J nested query 2 when applied to this sample data, we see that for the first row of r, no rows in s satisfy the predicate s.d = r.d, since r.d is 1, so we move to the second row. For the second row in r, there are two rows in s that satisfy s.d = r.d, since r.d is 2; thus for the second row of R, the result of the inner query block is the list

```
  c
  a
  b
```
We now see that r.b, namely 'a', is contained in this list, so the outer query returns the r.a value for this row, namely 'b'. Now, if we perform the join specified in query 2, we see that the results are the same: only the second row of R matches with a row of S.

End of Example.

Based on the above equivalences, Kim proposed the following fundamental algorithm for the transformation of type-N and type-J nested queries:
Algorithm NEST-N-J

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND together the clauses of all query blocks, replacing "IN" by "="
3. Retain the SELECT clause of the outermost query block.

The result is a single-level query logically equivalent to the original nested query. The algorithm applies to type-N or type-J nested queries with one or more levels of nesting.

3.2 Processing a Type-JA Nested Query

A type-JA nested query can be transformed to a type-J nested query which references a new temporary relation. That is, a query of the form

Query 3

```
SELECT r.a
FROM r
WHERE r.b =
  (SELECT AGG(s.e)
   FROM s
   WHERE s.d = r.c);
```

is equivalent to

Query 4

```
SELECT r.a
FROM r
WHERE r.b =
  (SELECT t.c2
   FROM t WHERE
   t.cl = r.c);
```

where t is a temporary table obtained by

```
CREATE t (cl , c2) AS
  (SELECT s.d, AGG (s.c)
   FROM s
   GROUP BY s.d);
```

The function AGG() in queries 3 and 4 can be any of the SQL aggregate functions MAX, SUM, AVG, or COUNT.

Kim's proof of this equivalence postulates that the action of the nested iteration processing of a type-JA query can be captured in a temporary table formed with a
GROUP BY clause, as in t. The temporary table contains one row for each distinct value in s.d, with the second column in the row containing the aggregate value of s.e for that s.d value. For each tuple of r, a tuple is retrieved from t whose C, (formerly s.d) value matches the r.d value of the t tuple. The c2 value of the t tuple will contain the aggregate value obtained by the GROUP BY clause, and this can be matched with r.b KM 82:455].

Of course, once we obtain the type-J equivalent, it will in turn be flattened to its single-level equivalent by algorithm NEST-N-J.

**Example.** Assume that R and S contain the following rows:

R:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>a</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
<td>b</td>
</tr>
</tbody>
</table>

S:

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>30</td>
</tr>
<tr>
<td>b</td>
<td>60</td>
</tr>
</tbody>
</table>

Assume also that in query 3 above, the aggregate function AGG() is SUM.

If we step through a nested-iteration execution of the original type-JA nested query 3 with this sample data, we see that for the first row of r, the first two rows in s satisfy the correlated join predicate s.d = r.c. So we apply SUM to s.e in these two rows, which gives us 30; this does not satisfy the nested predicate, since r.b for the first row is 20. When we move to the second row in r, we see that the last three rows in s satisfy the correlated join predicate s.d = r.c, since the joining value is 'b'. Now when we apply SUM to s.e for these three rows, we get 100, which matches r.b, thus satisfying the nested predicate; so our final result is the r.a value for the second row of r, namely "B".

Before we look at the equivalent type-J query, 4, we must form the temporary relation t by applying query (12) to table s. The result is the following:

T:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>30</td>
</tr>
<tr>
<td>b</td>
<td>100</td>
</tr>
</tbody>
</table>

Now let us flatten the type-J query 4 by applying algorithm NEST-N-J. The result is
Query 5

SELECT r.a
FROM r, t
WHERE r.b = t.C2 AND t.Cl = r.c;

We can see now that if we join r and t using query 5, the result will be "B", as for the original type-JA nested query 3.

This equivalence leads to an algorithm, which transforms a type-JA nested query of depth one to an equivalent type-J nested query of depth 1. The algorithm assumes a type-JA nested query of the following form:

```
SELECT r.C_{n+2}
FROM r.C_{n+1} = (SELECT AGG ( s.C_{n+1} )
    S.Cl = r.Cl AND
    s.C2 = r.C2 AND
    ...
    s.C_n = r.C_n);
```

The fundamental algorithm is specified as follows:

**Algorithm NEST-JA**

1. Generate a temporary relation T( C_1...,C_n,C_{n+1}) from S such that t.C_{n+1} is the result of applying the aggregate function AGG() on the C_{n+1} column of S which have matching values in R for C_1, C_2, etc.
2. Transform the inner query block of the initial query by changing all references to s columns in join predicates, which also reference r to the corresponding t columns. The result is a type-J nested query, which can be passed to algorithm NEST-N-J for transformation to its single-level equivalent.