Optimizing an SQL-like Nested Query
Overview:

- Introduction

- Classification of the Nested Queries

- Algorithms to transform nested to nonnested queries

- Potential performance improvements

- Strategy to proceed SQL-like nested queries of arbitrary complexity
Introduction:

The illustrative examples used in presentation are based on the following database of suppliers, parts, projects, and shipments:

- **SUPPLIER** (NO, SNAME, SLOC, SBUDGET)
- **PART** (PNO, PNAME, DIM, PRICE, COLOR)
- **PROJECT** (JNO, JNAME, PNO, JBUDGET, JLOC)
- **SHIPMENT** (SNO, PNO, JNO, QTY)

Throughout this presentation, $R_i$ denotes a relation in the database, $C_k$ the k-th column of a relation, $B$ the size in pages of available main-memory buffer.

The fundamental structure of an SQL-like query is syntactically represented by a query block which consists of SELECT, FROM, and WHERE clauses.

**Ex.:1**

```sql
SELECT MAX(PNO)
FROM PART
WHERE PRICE > '25';
```

Where X – const. or number of const.
op – scalar comparison operator (<, >, <=, etc.) or set membership operator (IS IN, IS NOT IN)
Extension of the simple predicates of the form \([R_i.C_k \text{ op } X]\), for n-relation queries:

**Nested predicate.** \([R_i.C_k \text{ op } Q]\),
where \( Q \) - SQL-like query block
\text{ op } - \text{ may be a scalar or set membership operator.}

if \([Q \text{ op } R_i.C_k]\). set membership operator is then replaced by the set containment operator (CONTAINS, DOES NOT CONTAIN).

**Join predicate.** \([R_i.C_k \text{ op } R_j.C_h]\),
where \( R_i \neq R_j \)
\text{ op } - \text{ a scalar comparison operator, here (=) operator.}

**Division predicate.** \([Q_i \text{ op } Q_i]\).
where \text{ op - scalar comparison operator,}
set comparison operator ,
set membership and containment operator.
\( Q_i \) (or \( Q_j \)) may be a constant or a list of constants,
provided - \( Q_i \) (or \( Q_j \)) has in its WHERE clause a join predicate
which references the relation of the outer query block.
Nesting of the Query Blocks: predicates

Only the nested predicate and the division predicate give rise to the nesting of query blocks.

A nested predicate may cause one of four basic types of nesting, and a division predicate yields a fifth basic nesting.
The inner query block Q does not contain a join predicate that references the relation of the outer query block.

the SELECT clause of Q indicates an aggregate function associated with the column name.

There is only one way to process a type-A nested query on a single processor:
1. the inner block has to be evaluated first.
2. the nested predicate of the outer block then becomes a simple predicate, since Q can be replaced by a constant.
3. the outer block then is no longer nested and can be processed completely.

A type-A nested query (predicate) of depth 1 is graphically represented as follows.
Nesting of the Query Blocks: type – N nesting

• Q does not contain a join predicate that references the relation of the outer query block.

• The column name in the SELECT clause of Q does not have an aggregate function associated with it.

The System R approach to processing the above query is:

1. to process the inner query block Q in order to replace the type-N nested predicate, PNO IS IN Q, with a simple predicate PNO IS IN X, where X is the result of evaluating Q,

2. then to completely process the resulting nonnested query.

The query graph for a type-N nested query of depth 1:

If \( op \) is the set noninclusion operator (IS NOT IN) requires a different treatment.
Nesting of the Query Blocks: type – N nesting

Note!!!

Note that Ex. 3 shows other formulations of the canonical query:

```sql
SELECT SNO
FROM SHIPMENT, PART
WHERE SHIPMENT.PNO = PART.PNO
    AND PRICE > '25';
```

This observation has some interesting implications, as can be seen later.

A type-N nested query of depth \( n(n \geq 1) \) is defined as a nested query whose query graph consists of only straight arcs labeled ‘N’.

\[
\text{Ri} \quad \text{N (C}_n\text{)} \quad \text{Rj} \quad \text{N (C}_m\text{)} \quad \text{Rk}
\]
Nesting of the Query Blocks: type – J nesting

- the WHERE clause of Q contains a join predicate that references the relation of the outer query block
- the column name in the SELECT clause of Q is not associated with an aggregate function,

```
SELECT SNO 
FROM SHIPMENT 
WHERE PNO IS IN (SELECT PNO 
FROM PROJECT 
WHERE SHIPMENT.SNO = PROJECT.JNO 
AND JLOC = 'NEW YORK') 
```

The query graph for a type-J nested query of depth 1:

The query graph for a type-J nested query of depth 1 involving the set noninclusion operator.
Nesting of the Query Blocks: type – J nesting

Ex. 4

```
SELECT SNO
FROM SHIPMENT
WHERE PNO IS IN (SELECT PNO
FROM PROJECT
WHERE SHIPMENT.SNO = PROJECT.JNO
AND JLOC = 'NEW YORK') ;
```

A type-J nested query of depth \( n(n \geq 1) \) is defined as a nested query whose query graph consists of at least one circular arc and no straight arc labeled ‘A’.

\[
\text{R}_i \xrightarrow{N(C_n)} \text{R}_j \xrightarrow{N(C_m)} \text{R}_k
\]

\( \text{R}_i.C_n = \text{R}_k.C_m \)
Nesting of the Query Blocks: type – JA nesting

- the WHERE clause of Q contains a join predicate that references the relation of the outer query block.
- an aggregate function is associated with the column name in the SELECT clause of Q.

Ex.5

```sql
SELECT SNO
FROM SHIPMENT
WHERE PNO = (SELECT MAX(PNO)
FROM PROJECT
WHERE PROJECT.JNO = SHIPMENT.JNO
AND JLOC = 'NEW YORK');
```

The query graph for a type-JA nested query of depth 1:

```
Ri -- A (Ck) -- Rj
Ri.Cn = Rj.Cm
```
A type-JA nested query of depth \( n(n \geq 1) \) is defined as a nested query whose query graph exhibits at least one circular arc and at least one straight arc labeled ‘A’.

\[
\begin{align*}
R_i & \stackrel{N(C_n)}{\longrightarrow} R_j \stackrel{A(C_m)}{\longrightarrow} R_k \\
R_i.C_n &= R_k.C_m
\end{align*}
\]
Nesting of the Query Blocks: type – D nesting

A join predicate and a division predicate together give rise to a type-D nesting.

- the join predicate in either Qi or Qj (or both references the relation of the outer block).

Type-D nested query expresses the relational division operation.

The query graph for a type-D nested query of depth 1:
some query may be expressed in its canonical form or type-N nested form.

Let $Q_1$ be

\[
\text{SELECT } R_i.C_k \\
\text{FROM } R_i, R_j \\
\text{WHERE } R_i.Ch = R_j.C_m
\]

And let $Q_2$ be

\[
\text{SELECT } R_i.C_k \\
\text{FROM } R_i, \\
\text{WHERE } R_i.Ch \text{ ISIN (SELECT } R_j.C_m \\
\text{FROM } R_j)\;
\]

**Lemma 1.** $Q_1$ and $Q_2$ are equivalent; that is, they yield the same result.

establishes the equivalence of the canonical and type-N nested form of a two-relation query in which the op is not the set noninclusion operator, \text{IS NOT IN}. 

Let Q₁ be

```
SELECT  Ri.Ck  
FROM     Ri, Rj  
WHERE    Ri.Ch = Rj.Cm
```

And let Q₂ be

```
SELECT  Ri.Ck  
FROM     Ri  
WHERE    Ri.Ch IS IN (SELECT  Rj.Cm  
                       FROM     Rj)
```

Proof:

By definition, the inner block of Q₂ can be evaluated independently of the outer block and the result of evaluating it is X, a list of values in the Cₘ column of Rⱼ. Q₂ is then reduced to:

```
SELECT  Ri.Ck  
FROM     Ri  
WHERE    Ri.Ch ISIN X;
```

The predicate Ri. Ch IS IN X is satisfied only if X contains a constant x such that Ri.Ch = X. That is, it can be satisfied only for those tuples of Ri and Rⱼ which have common values in the Ch and Cₘ columns, respectively. The join predicate Ri. Ch = Rⱼ. Cₘ specifies exactly this condition.
important to recognize that Lemma 1 establishes only that the type-N nested form of a query in which the op of the nested predicate is the set inclusion operator can be transformed to its canonical form.

Lemma 1 suggests Algorithm NEST-N-J for transforming a type-N nested query of depth $n - 1$ to its canonical form.
Processing a Type-N or Type-J Nested Query: 3

Algorithm NEST-N-J:

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND the WHERE clauses of all query blocks into one WHERE clause.
3. Replace \([R_i.Ch \text{ op} (SELECT R_j.Cm)]\) by a join predicate \([R_i.Ch, \text{new-op} R_j.Cm] \)
   and AND it to the combined WHERE clause obtained on step 2.
   Note that if \(\text{op}\) is IS IN, the corresponding \(\text{new-op}\) is ‘=’; otherwise - the same as \(\text{op}\).
4. Retain the SELECT clause of the outermost query block.

```
SELECT  Ri.Ck  
FROM    Ri, Rj  
WHERE   Ri.Ch = Rj.Cm AND  
        Ri.Cn=Rj.Cp);  
```
Algorithm NEST-N-J:

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND the WHERE clauses of all query blocks into one WHERE clause.
3. Replace \([R_i.Ch \text{ op} \ (SELECT \ R_j.Cm)]\) by a join predicate \([R_i.Ch, \text{ new-op} \ R_j.Cm]\), and AND it to the combined WHERE clause obtained on step 2. Note that if \(\text{op}\) is IS IN, the corresponding \(\text{new-op}\) is ‘=’; otherwise - the same as \(\text{op}\).
4. Retain the SELECT clause of the outermost query block.

```sql
SELECT R_i.Ck
FROM R_i
WHERE R_i.Ch IS IN (SELECT R_j.Cm
FROM R_j
WHERE R_i.Cn=R_j.CP);
```
Algorithm NEST-N-J:

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND the WHERE clauses of all query blocks into one WHERE clause.
3. Replace \([R_i.Ch \ op (SELECT R_j.Cm)]\) by a join predicate \([R_i.Ch, \new-op R_j.Cm]\), and AND it to the combined WHERE clause obtained on step 2.
   Note that if \(\op\) is IS IN, the corresponding \(\new-op\) is ‘=’; otherwise - the same as \(\op\).
4. Retain the SELECT clause of the outermost query block.

Processing a Type-N or Type-J Nested Query: 3

\[
\begin{align*}
\text{SELECT} & \quad R_i.Ck \\
\text{FROM} & \quad R_i \\
\text{WHERE} & \quad R_i.Ch \ IS \ IN \ (SELECT \ R_j.Cm \\
& \quad \text{FROM} \ R_j \\
& \quad \text{WHERE} \ R_i.Cn=R_j.CP); \\
\end{align*}
\]
Algorithm NEST-N-J:

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND the WHERE clauses of all query blocks into one WHERE clause.
3. Replace \([\text{Ri.Ch} \text{ op } \text{SELECT Rj.Cm}]\) by a join predicate \([\text{Ri.Ch}, \text{new-op} \text{ Rj.Cm}]\),
   and AND it to the combined WHERE clause obtained on step 2.
   Note that if \(\text{op}\) is IS IN, the corresponding \text{new-op} is ‘=’; otherwise - the same as \(\text{op}\).
4. Retain the SELECT clause of the outermost query block.

```sql
SELECT Ri.Ck
FROM Ri
WHERE Ri.Ch IS IN (SELECT Rj.Cm
                      FROM Rj
                      WHERE Ri.Cn=Rj.CP);
```
Algorithm NEST-N-J:

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND the WHERE clauses of all query blocks into one WHERE clause.
3. Replace \([R_i.Ch \text{ op (SELECT } R_j.Cm)\] by a join predicate \([R_i.Ch, \textbf{new-op } R_j.Cm]\), and AND it to the combined WHERE clause obtained on step 2.
   - Note that if \(\text{op}\) is IS IN, the corresponding \(\textbf{new-op}\) is ‘=’; otherwise - the same as \(\text{op}\).
4. Retain the SELECT clause of the outermost query block.

```
SELECT   Ri.Ck
FROM      Ri, Rj
WHERE     Ri.Ch = Rj.Cm AND
          Ri.Cn=Rj.Cp);
```
Processing a Type-N or Type-J Nested Query: 4

**SELECT SNAME**
**FROM** SUPPLIER
**WHERE** SNO IS NOT IN (SELECT SNO
**FROM** SHIPMENT
**WHERE** PNO = 'Pl')

alternate interpretation of the query is a type-D nested query.

**SELECT SNAME**
**FROM** SUPPLIER
**WHERE** (SELECT PNO
**FROM** SHIPMENT
**WHERE** SHIPMENT.SNO = SUPPLIER.SNO)
DOES NOT CONTAIN
PI;

Pseudocanonical form:

**SELECT SNAME**
**FROM** SUPPLIER, X
**WHERE** ¬(SUPPLIER.SNO = X.SNO);
The set noninclusion operator in a type-N or type-J nested query requires an extension to Algorithm NEST-N-J. Now \([\text{Ri.Ck IS NOT IN (SELECT Rj.Ch)}]\) is replaced by the antijoin predicate \(\neg(\text{Ri.Ck} = \text{Rj.Ch})\) and ANDed to the merged WHERE clause obtained on steps 2 and 3 of the algorithm.

The result of a query is independent of the order in which its predicates are evaluated. However, an antijoin predicate must be evaluated only after all join predicates have been evaluated.
**LEMMA 2.** $Q_3$ and $Q_4$ are equivalent; that is, they produce the same result.

Let $Q_3$ be

```
SELECT Ri.Ck
FROM   Ri
WHERE  Ri.Ch = (SELECT AGG(Rj.Cm)
                 FROM   Rj
                 WHERE Rj.Cn = Ri.Cp);
```

And let $Q_4$ be

```
SELECT Ri.Ck
FROM   Ri
WHERE  Ri.Ch = (SELECT Rt.C2
                 FROM   Rt
                 WHERE Rt.C1 = Ri.Cp);
```

where $Rt(C_1, C_2)$ is obtained by

```
Rt(C_1, C_2) = (SELECT Rj.Cn, AGG(Rj.Cm)
                FROM   Rj
                GROUP BY Rj.Cn);
```

Lemma 2 directly leads to an algorithm which transforms a type-JA nested query of depth 1 to an equivalent type-J nested query of depth 1.
Lemma 2. \( Q_3 \) and \( Q_4 \) are equivalent; that is, they produce the same result.

Let \( Q_3 \) be

\[
\text{SELECT} \ R_i.C_k \\
\text{FROM} \ R_i \\
\text{WHERE} \ R_i.C_h = (\text{SELECT} \ \text{AGG}(R_j.C_m) \\
\text{FROM} \ R_j \\
\text{WHERE} \ R_j.C_n = R_i.C_p);
\]

And let \( Q_4 \) be

\[
\text{SELECT} \ R_i.C_k \\
\text{FROM} \ R_i \\
\text{WHERE} \ R_i.C_h = (\text{SELECT} \ R_t.C_2 \\
\text{FROM} \ R_t \\
\text{WHERE} \ R_t.C_1 = R_i.C_p);
\]

where \( R_t(C_1, C_2) \) is obtained by

\[
R_t(C_1, C_2) = (\text{SELECT} \ R_j.C_n, \text{AGG}(R_j.C_m) \\
\text{FROM} \ R_j \\
\text{GROUP BY} \ R_j.C_n);
\]

Processing a Type-JA Nested Query: 1

Proof. It is shown in Section 2 that the operation of \( Q_3 \) may be thought of as first fetching a tuple of \( R_i \) and all tuples of \( R_j \) whose \( C_n \) column values are the same as the \( C_p \) column value of the \( R_i \) tuple, then applying the aggregate function AGG on the \( C_m \) column of the \( R_j \) tuples to obtain a constant \( X \), and, finally, outputting the \( C_k \) value of the \( R_i \) tuple if \( x = C_\epsilon \), column value of the \( R_i \) tuple. Now \( R_t \) is a binary relation of each distinct value in the \( C_n \) column of \( R_j \) and the corresponding value obtained by applying the aggregate function AGG on the \( R_j \) tuples. Then it is clear that the query may be processed by fetching each tuple of \( R_i \), then fetching the \( R_t \) tuple whose \( C_1 \) column has the same value as the \( C_p \) column of the \( R_i \) tuple, and outputting the \( C_k \) column value of the \( R_t \) tuple if the \( C_2 \) column value of the \( R_t \) tuple is the same as the \( C_h \) value of the \( R_i \) tuple. But this is exactly the operation of \( Q_4 \).
Algorithm NEST-JA:

1. Generate a temporary relation $R_t( C_1, \ldots, C_n, C_{n+1})$ from $R_2$ such that each $C_{n+1}$ column value of $R_t$ is a constant obtained by applying the aggregate function $AGG$ on the $C_{n+1}$ column of the $R_2$ tuples which share a common value in columns $C_1$ through $C_n$. In other words, the primary key of $R_t$ is its first $n$ columns.

2. In inner block of the initial query change all references to $R_2$ columns in join predicates that reference $R_1$ to corresponding $R_t$ columns.

```sql
SELECT R1. C_{n+2} 
FROM R1 
WHERE R1. C_{n+1} = (SELECT AGG(R2.C_{n+1}) 
FROM R2 
GROUP BY C_1, \ldots, C_n);
```

$$R_t(C_1, \ldots, C_n, C_{n+1}) = (SELECT C_1, \ldots, C_n, AGG(C_{n+1}) 
FROM R_2 
GROUP BY C_1, \ldots, C_n);$$
Algorithm NEST-JA:

1. Generate a temporary relation $R_t( C_1, \ldots, C_n, C_{n+1})$ from $R_2$ such that each $C_{n+1}$ column value of $R_t$ is a constant obtained by applying the aggregate function $AGG$ on the $C_{n+1}$ column of the $R_2$ tuples which share a common value in columns $C_1$ through $C_n$. In other words, the primary key of $R_t$ is its first $n$ columns.

2. In inner block of the initial query change all references to $R_2$ columns in join predicates that reference $R_1$ to corresponding $R_t$ columns.

```sql
SELECT R1. C_{n+2} 
FROM R1 
WHERE R1.C_{n+1} = (SELECT R_t.C_{n+1} 
FROM R_t 
WHERE R_t.C_1 = R1.C_1 AND 
    R_t.C_2 = R1.C_2 AND 
    R_t.C_n = R1.C_n); 
```
transformation of a type-JA nested query of depth 2

type-JA (NA)

type-JA (AA)

N A

A A

NEST- JA

NEST- JA

A N

type-JA (AN)

type-JA

e

A N

NEST- N -J
Algorithm NEST-JA(G)

1 = n (the nesting depth of the query);
DO WHILE (there-is-at-least-one-straight-arc-labeled-A);
   IF the I-th straight arc is labeled N THEN I = I - 1;
   ELSE DO;
      Apply Algorithms NEST-JA and (NEST-N-J) to the n - I + 1 nodes to the right of the I-th
      straight arc; The I-th straight arc of the resulting query of depth I is labeled N; n = I;
      END;
   END;
END;
**Lemma 3.** $Q_5$ and $Q_6$ are equivalent; that is, they produce the same result.

Let $Q_5$ be
\[
\text{SELECT Ri.Ck} \\
\text{FROM Ri} \\
\text{WHERE (SELECT Rj.Ch} \\
\text{FROM Rj} \\
\text{WHERE Rj.Cn=Ri.Cp)} \\
\text{op (SELECT Rk.Cm} \\
\text{FROM Rk)};
\]

And let $Q_6$ be
\[
\text{SELECT Ri.Ck} \\
\text{FROM Ri} \\
\text{WHERE Ri.Cp = (SELECT C1} \\
\text{FROM Rt)};
\]

where $R_t$ is obtained as:
\[
R_t(C1) = (\text{SELECT Rj.Cn} \\
\text{FROM Rj.RX} \\
\text{WHERE (SELECT Rj.Ch} \\
\text{FROM Rj.RY} \\
\text{WHERE RY.Cn = RX.Cn)} \\
\text{op (SELECT Rk.Cm} \\
\text{FROM Rk)};
\]

Lemma 3 provides the basis for Algorithm NEST-D, which transforms a general type-D nested query to an equivalent canonical two-relation query.
PROOF.

As has been shown, the operation of $Q_5$ may be thought of as fetching each tuple of $R_i$ and checking whether the division predicate is satisfied by the $C_p$ column value of the tuple. But what if there is a list of the $C_n$ column values of $R_j$ which satisfy the division predicate? Then all that needs to be done is to fetch each $R_i$ tuple and determine whether the $C_p$ column value of the tuple is in the list. But this is precisely the operation of $Q_5$, since $R_t$ is just such a list.

Let $Q_5$ be

$$\text{SELECT } Ri.Ck \\text{FROM } Ri \\text{WHERE } (\text{SELECT } Rj.Ch \\text{FROM } Rj \\text{WHERE } Rj.Cn=Ri.Cp) \text{ op } (\text{SELECT } Rk.Cm \\text{FROM } Rk);$$

And let $Q_6$ be

$$\text{SELECT } Ri.Ck \\text{FROM } Ri \\text{WHERE } Ri.Cp = (\text{SELECT } C1 \\text{FROM } Rt) ;$$

where $R_t$ is obtained as:

$$Rt(C1) = (\text{SELECT } Rj.Cn \\text{FROM } Rj.RX \\text{WHERE } (\text{SELECT } Rj.Ch \\text{FROM } Rj.RY \\text{WHERE } RY.Cn = RX.Cn) \text{ op } (\text{SELECT } Rk.Cm \\text{FROM } Rk);$$
Algorithm NEST-D

1. Generate a temporary relation $R_{t1}(C_1, ..., C_m)$: $R_{t1}(C_1, ..., C_m) = (\text{SELECT } C_1, ..., C_m \text{ FROM } R_3)$;

Also generate $R_{t2}(C_1, ..., C_n)$: $R_{t2}(C_1, ..., C_n) = (\text{SELECT } C_1, ..., C_n \text{ FROM } R_2)$;

2. Divide $R_{t2}(C_1, ..., C_n)$ by $R_{t1}(C_1, ..., C_m)$. The result is a new temporary relation $R_{t3}(C_{m+1}, ..., C_n)$.

3. Transform the initial query to canonical
   - drop the query block on $R_3$,
   - replace all references to columns of $R_2$ to corresponding columns of $R_{t3}$ in the query block on $R_2$,
   - eliminate the SELECT and FROM clauses of the query block on $R_2$.

The FROM clause of the resulting canonical query must now include $R_{t3}$ as well.
Algorithm NEST-D

1. Generate a temporary relation $R_{t1}(C1, ..., Cm)$: $R_{t1}(C1, ..., Cm) = (SELECT C1, ..., Cm FROM R3)$;

Also generate $R_{t2}(C1, ..., Cn)$: $R_{t2}(C1, ..., Cn) = (SELECT C1, ..., Cn FROM R2)$.

2. Divide $R_{t2}(C1, ..., Cn)$ by $R_{t1}(C1, ..., Cm)$. The result is a new temporary relation $R_{t3}(Cm+1, ..., Cn)$.

3. Transform the initial query to canonical
   - drop the query block on $R3$,
   - replace all references to columns of $R2$ to corresponding columns of $R_{t3}$ in the query block on $R2$,
   - eliminate the SELECT and FROM clauses of the query block on $R2$.

The FROM clause of the resulting canonical query must now include $R_{t3}$ as well.
Algorithm NEST-G

1. Transform each type-A predicate to a simple predicate by evaluating the Q represented by the right-hand node on the query graph for the subquery (predicate). Then eliminate Q and the straight arc labeled ‘A’, leading to Q from the outermost query block.

2. Transform each type-JA nested subquery to an equivalent type-N or type-J subquery by Algorithm NEST-JA(G). The query graph for the resulting type-N or type-J nested subquery replaces the query graph for the initial type-JA subquery.

3. Transform each type-D nested subquery to its canonical form by Algorithm NEST-D. Replace the division predicate by an appropriate set of join predicates, and remove from the query graph of the initial subquery the two right-hand nodes and both the straight and the circular arc leading to them.

4. Transform the resulting query, which consists only of type-N and type-J subqueries, to an equivalent canonical query by Algorithm NEST-N-J.
Processing a General Nested Query: 2

```
SELECT R1.C1
FROM R1
WHERE R1.C2 ISIN (SELECT R2.C2
                   FROM R2
                   WHERE R2.C3 = (SELECT AGG(R3.C3)
                                   FROM R2
                                   WHERE R3.C4 = R1.C4)
                   AND R2.C4 IS IN (SELECT R4.C4,
                                     FROM Rd)
                   AND R1.C3 = (SELECT AGG(R5.C5)
                                 FROM R5
                                 WHERE R5.C6 IS NOT IN (SELECT R6.C6
                                                           FROM R6))
                   AND (SELECT R7.C7
                          FROM R7
                          WHERE R7.C8 = R1.C5)
                   (SELECT R8.C8
                    FROM R8);}
```
Processing a General Nested Query: 3

R1.C4 = R3.C4

R1.C5 = R7.C8

R1.C4 = R3.C4

A(C3)

N(C2)

N(C4)

N_x(C6)
Processing a General Nested Query: 4

**Type-JA depth 2**: type-J equivalent.

`NEST-JA(G)`

**Type-JA depth 1**: conjunction of join predicates

`NEST-JA, 
R2.C3 = Rt1.C2 AND
Rt1.C1 = R1.C4`

Where the temporary relation `Rt1` is obtained by

`Rt1(C1, C2) = (SELECT R3.C4, AGG(R3.C3)
FROM
GROUP BY R3.C4);`

**Type-N depth 1**: join predicate, `R2.C4 = R4.C4`

`NEST-N-J`

Then **Type-JA depth 2**: type-J depth 1

`NEST-N-J`

`R1.C2 IS IN (SELECT R2.C2
FROM R2, R4, Rt1
WHERE R2.C3 = Rt1.C2 AND
Rt1.C1 = R1.C4 AND
R2.C4 = R4.C4);`
Processing a General Nested Query: 5

R2.C3 = Rt1.C2
AND  Rt1.C1 = R1.C4
AND  R2.C4 = R4.C4

R1.C5 = R7.C8
Processing a General Nested Query: 6

type-A depth 2 is evaluated.

type-N → SELECT AGG(R_5.C_5)
recursive call FROM R_5, R_6
NEST-G WHERE ¬(R_5.C_6 = R_6.C_6);

The transformed node is then evaluated to a constant X, and the nested predicate of the initial type-A subquery becomes a simple predicate, R_1.C_3 = x.
Processing a General Nested Query:

\[ R_1 \]
\[ R_2 \]
\[ R_3 \]
\[ R_4 \]
\[ R_5 \]
\[ R_6 \]
\[ R_7 \]
\[ R_8 \]
\[ N(C_2) \]
\[ R_9 \]
\[ R_{10} \]

\[ R_2.C_3 = R_1.C_2 \]
\[ \text{AND} \]
\[ R_1.C_1 = R_4.C_4 \]
\[ \text{AND} \]
\[ R_2.C_4 = R_4.C_4 \]
\[ \text{AND} \]
\[ R_1.C_3 = x \]

\[ R_1.C_5 = R_7.C_8 \]

\[ R_2.C_3 = R_{10}.C_2 \]
\[ \text{AND} \]
\[ R_{10}.C_1 = R_1.C_4 \]
\[ \text{AND} \]
\[ R_2.C_4 = R_4.C_4 \]
\[ \text{AND} \]
\[ R_1.C_3 = x \]
Then, type-D subquery is evaluated so as to replace the division predicate with a join predicate by Algorithm NEST-D. The resulting join predicate is R1. CS = Rn. CI, where the unary relation Ra(C,) is the quotient of dividing RT(CB, CT) by RdCs.

The resulting query is type-J nested and Algorithm NEST-N-J can be used to transform it into its canonical equivalent, shown in the following.

\[
\text{SELECT } R1.C1 \\
\text{FROM } R1, Rt1, Rt2 \\
\]
Performance improvements:

Consider Q3 and Q4:

Let \( P_i = 50, P_j = 30, P_t = 5, B = 6, \) and \( f_i.N_i = 100. \)

The nested-iteration method of processing Q3 is, worst case, 3050 page fetches.

Q4 incurs 615 page fetches.

**type-D nested query:**

Let \( P_i = 50, P_j = P_n = 30, P_t = 5, P_k = P_{ts} = 1, B = 4, \) and \( f_i.N_i = 1000. \)

The nested-iteration method requires, in the worst case, 30051 page fetches.

Processing the type-D nested query by the algorithms shown here is 751 page I/OS.
Thank you for the attention!!!