Execution Strategies for SQL Subqueries

Mostafa Elhemali, César Galindo-Legaria, Torsten Grabs, Milind Joshi
Microsoft Corp
With additional slides from material in paper, added by S. Sudarshan

Motivation

- Optimization of subqueries has been studied for some time
- Challenges
  - Mixing scalar and relational expressions
  - Appropriate abstractions for correct and efficient processing
  - Integration of special techniques in complete system
- This talk presents the approach followed in SQL Server
- Framework where specific optimizations can be plugged
- Framework applies also to “nested loops” languages
Outline

- Query optimizer context
- Subquery processing framework
- Subquery disjunctions

Algebraic query representation

- Relational operator trees
- Not SQL-block-focused

```
SELECT SUM(T.a) FROM T, R
WHERE T.b = R.b AND R.c = 5
GROUP BY T.c
```

algebrize transform
Operator tree transformations

Select (A.x = 5)  
Join  
A  B

GropBy A.x, B.k, sum(A.y)  
Join  
A  B

Simplification / normalization  
Exploration  
Implementation

SQL Server Optimization process

cost-based optimization

pool of alternatives

search(0)  search(1)  search(2)

T0  T1
(input)  (output)

simplify

use simplification / normalization rules

use exploration and implementation rules, cost alternatives

T2

(cost alternatives)
Plan Generation Overview

Outline

- Query optimizer context
- Subquery processing framework
- Subquery disjunctions
SQL Subquery

- A relational expression where you expect a scalar
  - Existential test, e.g. NOT EXISTS(SELECT…)
  - Quantified comparison, e.g. T.a = ANY (SELECT…)
  - Scalar-valued, e.g. T.a = (SELECT…) + (SELECT…)

- Convenient and widely used by query generators

Algebrization

```sql
SELECT *
from customer
where 100,000 <
  (select sum(o_totalprice)
   from orders
   where o_custkey = c_custkey)
```

**Subqueries:** relational operators with scalar parents
Commonly “correlated,” i.e. they have outer-references
Subquery removal

- Executing subquery requires mutual recursion between scalar engine and relational engine
- **Subquery removal**: Transform tree to remove relational operators from under scalar operators
- Preserve special semantics of using a relational expression in a scalar, e.g. at-most-one-row

The Apply operator

\[ R \text{Apply}_{\mathcal{J}N} E(r) = \bigcup_{r \in R} \{r\} \mathcal{J}N E(r) \]

- **R Apply** $E(r)$
  - For each row $r$ of $R$, execute function $E$ on $r$
  - Return union: $\{r_1\} \times E(r_1) \cup \{r_2\} \times E(r_2) \cup \ldots$
  - Abstracts “for each” and relational function invocation
- Also known as d-join and tuple-substitution join
- Variants: left outer join, semi-join, anti-join
- Exposed in SQL Server (FROM clause)
  - LATERAL clause in SQL standard
- Useful to invoke table-valued functions
Subquery removal

SQL Query:
- SELECT *, (SELECT C_NAME FROM CUSTOMER WHERE C_CUSTKEY = O_CUSTKEY) FROM ORDERS

Translated to
- ORDERS ApplyOj(π[C_NAME] σ [C_CUSTKEY = O_CUSTKEY] CUSTOMER)

In general:
- R ApplyOj max1row(E(r))

Subqueries with exists/not exists become
- R ApplySj E(r)
- R ApplyASj E(r)
Conditional Scalar Execution

- Expression
  - CASE WHEN EXISTS(E1(r)) THEN E2(r) ELSE 0 END

- Translated to
  - \( \text{CASE WHEN } p = 1 \text{ THEN } e2 \text{ ELSE 0 END} \)
  - \((R \text{ Apply}[\text{semijoin, probe as } p] E1(r))\)
  - \(\text{Apply}[\text{outerjoin, pass-through } p=1]\)
  - \(\text{max1row}(E2(r))\) as \(e2\)

Disjunction of SubQueries

- \(WHERE p(r) \text{ OR EXISTS( } E1(r) \text{) OR EXISTS( } E2(r) \text{)}\)

- \(R \text{ Apply}_{SJ} (\sigma_{p(r)} \text{ CT(1) UA } E1(r) \text{ UA } E2(r))\)
  - CT(1): Constant Table returning 1
  - UA: Union All

- Can also translate to apply with passthrough
Quantification and NULLs

- Consider predicate \( 5 \text{ NOT IN } S \) which is equivalent to \( <> \text{ ALL} \)
- The result of this predicate is as follows, for various cases of set \( S \):
  1. If \( S = \{\} \) then \( p \) is TRUE.
  2. If \( S = \{1\} \) then \( p \) is TRUE.
  3. If \( S = \{5\} \) then \( p \) is FALSE.
  4. If \( S = \{\text{NULL}, 5\} \) then \( p \) is FALSE.
  5. If \( S = \{\text{NULL}, 1\} \) then \( p \) is UNKNOWN.
- \[(\text{FOR ALL } s \text{ in } S: p)\]
  - \( = (\text{NOT EXISTS } s \text{ in } S: \text{NOT } p) \) But only without nulls
  - In general predicate \( A \text{ cmp } B \) is translated as \( A <\text{cmp}' > B \text{ OR } A \text{ IS NULL OR } B \text{ IS NULL} \)
    - where \( \text{cmp}' \) is the complement of \( \text{cmp} \)

Apply removal

- Executing Apply forces nested loops execution into the subquery
- **Apply removal**: Transform tree to remove Apply operator
  - The crux of efficient processing
  - Not specific to SQL subqueries
  - Can go by “unnesting,” “decorrelation,” “unrolling loops”
  - Get joins, outerjoin, semijoins, … as a result
Apply removal

Apply does not add expressive power to relational algebra
Removal rules exist for different operators

Why remove Apply?

- Goal is **NOT** to avoid nested loops execution, but to normalize the query

- Queries formulated using “for each” surface may be executed more efficiently using set-oriented algorithms

- … and queries formulated using declarative join syntax may be executed more efficiently using nested loop, “for each” algorithms
Removing Apply Cont.

- Apply removal that preserves the size of the expression.
  - With Apply
    - ORDERS $\text{Apply}_{\text{order}} \left( \sigma_{\text{C.CUSTKEY} = \text{O.CUSTKEY}} \right) \text{CUSTOMER}$
  - Removing apply
    - ORDERS $\Join \left( \sigma_{\text{C.CUSTKEY} = \text{O.CUSTKEY}} \right) \text{CUSTOMER}$
- Apply removal that duplicates subexpressions.
  $$\text{R} \: \text{Apply}_{\text{binary}} \left( \sigma_{\text{R.a} = \text{S.a}} \right) \cup \sigma_{\text{R.b} = \text{T.b}} \text{T}$$
  $$= \text{R} \: \Join_{\text{R.a} = \text{S.a}} \sigma_{\text{R.b} = \text{T.b}} \text{T}$$

- Apply removal not always possible
  - max1row/pass-through predicates, opaque functions

Magic Sets

- Originally formulated for recursive query processing
- Special case for non-recursive queries
  $$\text{R} \: \text{Apply} \left( \text{E(r)} \right) = \text{R} \: \Join_{\text{R.r=R'.r}} \text{DS}$$
  $$\text{DS} = \text{R' \: Apply} \left( \text{E(r')} \right)$$
  $$\text{R'} = \text{distinct}[\text{R.r}] \: \text{R}$$
Magic Sets with Group By

A: $R \Join_{p(r,s)}^p (r,x) \ G_{s,x} = \text{agg} S$

M: $R \Join_{r'}^r (r,x) (G_{s,x} = \text{agg} R \ Join_{r',s}^p S)$

- Other options
  - B: Pull groupby above join
  - C: “Segmented execution”, when $R$ and $S$ are the same
    - E.g. Select all students with the highest mark

Reordering Semijoins and Antijoins

- Pushing down semi/anti joins
  \[
  (G_{A,F}R) \ Join_{p(A,S)}^p S = G_{A,F} (R \ Join_{p(A,S)}^p S)
  
  (G_{A,F}R) \ AntiJoin_{p(A,S)}^p S = G_{A,F} (R \ AntiJoin_{p(A,S)}^p S)
  \]

- Converting semi-join to join (to allow reordering)
  \[
  R \ Join_{p(R,S)}^p S = G_{\text{key}(R),\text{Any}(R)} (R \ Join_{p(R,S)}^p S)
  \]

ORDERS $SJ(\text{CUSTOMER} JN \text{ SUPPLIER} JN \text{ LINEITEM})$

- How about anti-joins?
Subquery Disjunctions

SELECT COUNT(*)
FROM LINEITEM
WHERE L_SHIPDATE != ALL(SELECT O_ORDERDATE
from ORDERS)

- generates an antijoin with predicate
  L_SHIPDATE = O_ORDERDATE OR L_SHIPDATE IS NULL

- which can be rewritten using
  \((R AS_Sp1 OR_Sp2 S) = ((R AS_Sp1 S) AS_Sp2 S)\)

- Another useful rule
  \(R APP\_S1 E_1(r_1) \cup A E_2(r_2) \cup A E_3(r_3) \ldots \rightarrow\)
  \(R APP\_S1 E_1(r_1) \cup (R APP\_S2 E_2(r_2)) \cup (R APP\_S3 E_3(r_3)) \ldots\)

Categories of execution strategies

select ...
from customer
where exists(… orders …)
and …

- semijoin
- normalized logical tree

- apply
- hash / merge join
- apply

- forward lookup
- set oriented
- reverse lookup
Forward lookup

CUSTOMER ORDERS Lkup(O_CUSTKEY=C_CUSTKEY)

APPLY[semijoin](bind:C_CUSTKEY)

The “natural” form of subquery execution
Early termination due to semijoin – pull execution model
Best alternative if few CUSTOMERs and index on ORDER exists

Reverse lookup

ORDERS CUSTOMERS Lkup(C_CUSTKEY=O_CUSTKEY)

APPLY(bind:O_CUSTKEY)

DISTINCT on C_CUSTKEY

APPLY(bind:O_CUSTKEY)

DISTINCT on O_CUSTKEY

ORDERS

Mind the duplicates
Consider reordering GroupBy (DISTINCT) around join
Subquery processing overview

SQL without subquery → relational expr without Apply → logical reordering → set-oriented execution

“nested loops” languages → relational expr with Apply → physical optimizations → navigational, “nested loops” execution

SQL with subquery → Removal of Subquery

Removal of Apply

Parsing and normalization → Cost-based optimization

The fine print

- Can you always remove subqueries?
  - Yes, but you need a quirky Conditional Apply
  - Subqueries in CASE WHEN expressions

- Can you always remove Apply?
  - Not Conditional Apply
  - Not with opaque table-valued functions

- Beyond yes/no answer: Apply removal can explode size of original relational expression
Outline

- Query optimizer context
- Subquery processing framework
- Subquery disjunctions

Subquery disjunctions

```sql
select * from customer
where c_catgory = "preferred"
or exists(select * from nation where n_nation = c_nation and …)
or exists(select * from orders where o_custkey = c_custkey and …)
```

![Diagram of subquery disjunctions]

Natural forward lookup plan
Union All with early termination short-circuits “OR” computation
Apply removal on Union

```
UNION (DISTINCT)

SELECT(CATEGORY = "preferred")
CUSTOMER

SEMIJOIN
CUSTOMER ORDERS

SEMIJOIN
CUSTOMER NATION
```

Distributivity replicates outer expression
Allows set-oriented and reverse lookup plan
This form of Apply removal done in cost-based optimization, not simplification

---

Optimizing Apply

- Caching of results from earlier calls
  - Trivial if no correlation variables
  - In-memory if few distinct values/small results
  - May or may no be worthwhile if large results
- Asynchronous IO
- Batch Sort
Asynchronous IO

- Ask OS to prefetch data, continue doing other things while prefetch is happening
  - better use of resources, esp with multiple disks/CPUs
- SELECT <blah blah>
  FROM PART natural join SUPPLIER natural join PARTSUPP
  WHERE <restrictive selections>
  AND PS_SUPPLYCOST = (SELECT MIN(PS_SUPPLYCOST)
  FROM PARTSUPP, SUPPLIER
  WHERE P_PARTKEY = PS_PARTKEY
  AND S_SUPPKEY = PS_SUPPKEY)
- Plan used by SQL Server (how the hell did it come up with this?)

\[
R.\text{ApplySyn}(\text{Seek}_R.P\_PARTKEY=PS\_PARTKEY, PARTSUPP))
\]

Batch Sort

- Sort order of parameters can help inner query
- But sorting outer query can be time consuming
  - esp if we stop after a few answers
- So: batch a group of outer parameters, sort them, then invoke inner in sorted order
- Batch size increased step-wise
  - so first few answers are fast at cost of more IO, later ones optimize IO more but with some delay
Summary

- Presentation focused on overall framework for processing SQL subqueries and “for each” constructs
- Many optimization pockets within such framework – you can read in the paper:
  - Optimizations for semijoin, antijoin, outerjoin
  - “Magic subquery decorrelation” technique
  - Optimizations for general Apply
  - ...
- Goal of “decorrelation” is not set-oriented execution, but to normalize and open up execution alternatives

A question of costing

Fwd-lookup … 10ms to 3 days

Bwd-lookup … 10ms to 3 days, cases opposite to fwd-lkp
Optimizer that picks the right strategy for you … priceless
Set-oriented execution … 2 to 3 hours
Execution Strategies for Semijoin

- Outer query on orders, exists subquery on lineitem (Section 6.1)

![Graph showing response time for selectivity combinations (ORDERS vs. LINEITEM)]

Execution Strategies for Antijoin

- Outer query uses only orders, exists subquery on lineitem

![Graph showing response time for antijoin strategies]
Strategies for Subquery Disjunction

- Section 7.1: One disjunction is a select, other is an exists on a subquery

Execution Optimization for Apply