SubQueries in Hive

Harish Butani

November 14, 2013

The need for SubQueries support is documented in Jira 784; this is part of the larger effort to get Hive's SQL dialect to be as close to the SQL 92 standard. Many popular databases have added SubQuery support not from the get go, but as an add on in one of their later releases. Two seminal papers **Kim82** and **Dayal87** provide a kind of blueprint on how to introduce SubQueries. Many DBs appear to have started by providing support for the Use Cases listed in these papers than then over time supporting more Use Cases. We propose to take a similar approach.

In the following we first describe the scope of the problem. The literature describes different kinds of SubQueries based on several factors: is the Query correlated, is there aggregation involved, what Operator is involved etc. If we have to support all SubQuery use cases then a general *nested iteration method* must be supported that evaluates a SubQuery for each tuple of the outer Query. This obviously is not desirable in the MR context. But there is a class of SubQueries that can be rewritten through Algebraic transformations into Joins. We document the cases that can be rewritten. We also provide Implementation details.

1 Definition of SubQueries and our Restrictions

A SubQuery is a *SQL expression* that returns a Set of tuples. The SQL-92 standard allows SubQuery expressions to appear anywhere an expression can. But there are restrictions on the kinds of SubQueries that can appear as Select Items, Group By clauses etc. In this document we only focus on SubQueries appearing in the SQL *Where Clause*. So the first **Restriction** is

Restriction.1.h SubQueries only supported in the SQL Where Clause.

- Reason: We focus on providing SubQuery support via Algebraic transformations. This is only possible for SubQuery predicates.

We use the following notation to specify **Restrictions**:

- Restriction.number.[soft|medium|hard]
  - soft implies, we may relax this restriction. The reason for the restriction is an implementation detail.
  - medium implies, we may relax the restriction to a less constrained one; for e.g. we don't plan to allow nested SubQueries in the Where clause; but it be possible to relax this to allow nested SubQueries that only correlate with their Parent Query Block.

  - hard implies, we will not relax this restriction. Removing this restriction is a major enhancement in Design/Implementation.
  - When a Restriction matches the SQL specs behavior, we call the restriction a **Check**.

In general, a predicate involving a SubQuery has the following form:

- the Predicate Operator can be one of:
  - comparison operator, with optionally an *all* or *any* qualifier.
  - the *in* or *not in* operators.
  - the *exists* or *not exists* operators. In this case there is no LHS.
- The comparison maybe between a scalar expression on the Parent Query's tuple and a SubQuery, or between 2 SubQueries.

A SubQuery can be characterized as:

- Aggregated or not. An *aggregated* query is one which involves a Group By operation. This maybe implicit or explicit. An implicit Group By is one in which the aggregation occurs over all rows of the input, so there is only 1 row(in fact there is always 1 row) in the Output.
- Correlated or not. We define correlated as having 1 or more Linking predicates. A Linking predicate has the equality operator and one of its sides(lhs or rhs) must reference columns from the Parent Query Block; while the other must reference columns in this Query Block.

Given the above definitions: we have several subclasses of Sub Query use cases: based on whether it is Aggregated, Correlated and the operator involved.

The most generic (naive) implementation for SubQueries is to evaluate the SubQuery for each tuple of the outer Query, and then evaluate the predicate for the outer tuple based on the output Set/Value returned. So in Hive we could:

- introduce a SubQuery Expression. Which would appear in the Evaluation Tree in the Filter Operator.
- For each input Tuple: it would evaluate the SubQuery. Depending on the SubQuery this could be done as a Local Task; but in general this may entail spawning a MR Job for each tuple.
- Then based on the results the input tuple would be filtered or not.

Obviously this is not a feasible option. But there are many techniques to improve on this:

- Evaluate the SubQuery for all possible values in one shot. This is accompanied by a Join with the Parent Query.
- Mnemonize the Sub Query result for each distinct value and reuse.

As a first pass at supporting Sub Queries in Hive, we propose to take the first approach: convert Sub Query predicates into Joins in the Parent Query. This is achieved by a set of Algebraic transformation rules that are applied during Filter Plan generation of the Parent Query. But for this to work, that is to ensure that the Join based plan is semantically equivalent to evaluating Sub Queries on a tuple by tuple basis, we have to impose a set of Restrictions on what is allowed in Sub Queries. Next we document all the Restrictions for easy access in one place. Further details on these can be gathered from reading the Algebraic Transformation section.

Restriction.2.h The subquery can only be the RHS of an expression. We don't support comparison between 2 SubQuery lists. For other cases this is what the language allows. Also the LHS of a SubQuery predicate must be a scalar expression.

Restriction.3.m The predicate operators supported are In, Not In, exists and Not exists. Haven't worked out Semantics in all cases for comparison operator Algebraic transformations.

Check.4.h For Exists and Not Exists, the Sub Query must have 1 or more correlated predicates.

Check.5.h For In and Not In the SubQuery must implicitly or explicitly only contain one select item.

Restriction.6.m The LHS in a SubQuery must have all its Column References be qualified. So for e.g.

```
- Query 1:
select ...
from x
where a > 5 and x.b in (select u from y where y.c = 10 )
- Query 2:
select ...
from x
where a > 5 and b in (select u from y where y.c = 10 )
```

Query 1 will work; Query 2 will not. This is because we will translate Sub-Query expressions into Joins, and join predicates require column references to be qualified.

This restriction also applies to correlated predicates. So below, Query 3 is valid, Query 4 is not.

```
- Query 3:
select ...
from x
where a > 5 and x.b in (select u from y where y.c = 10 and x.c = y.v )
- Query 4:
select ...
from x
where a > 5 and b in (select u from y where y.c = 10 and c = y.v)
```

Restriction.7.h SubQuery predicates can appear only as top level conjuncts. We split the Where Clause of a Query into conjuncts (at top level AND operators). We allow only children of top level AND operators to be a SubQuery expression. Generic support requires us to convert an arbitrary Where clause into Conjunctive Normal Form; don't plan to support this. For e.g. these are not supported:

```
- query 1
select ...
from x
where a > 5 or
    x.b in (select u from y where y.c = 10 and x.c = y.v )

- query 2
select ...
from x
where  not x.b in (select u from y where y.c = 10 and x.c = y.v )
```

Restriction.8.m We allow only 1 SubQuery expression per Query. The semantics of multiple SubQueries is not worked out. We may relax this Restrictions for specific SubQuery types and/or Operators. For e.g. this is not allowed:

```
select ...
from x
where a > 5 and
    x.b in (select u from y where y.c = 10 and x.c = y.v ) and
    x.c not in (select v from y where y.d = 15)
```

Restriction.9.m We will not support nested SubQuery expressions. Support where correlation expressions refer to any ancestor Query Block, is more involved, we don't plan to support this. Support where a correlation expression can refer to just the Parent Block can be looked at. For e.g.s these are not supported. Query 2, we may consider supporting.

```
-query 1
select ...
from x
where
    x.b in (select u
    from y
    where y.c = 10 and
  exists (select m from z where z.A = x.C)
   )
- query 2
select ...
from x
where
    x.b in (select u
    from y
    where y.c = 10 and
```

```
  exists (select m from z where z.A = y.D)
   )
```

Restriction.10.h In a SubQuery references to Parent Query columns is only supported in the where clause. So they are not allowed in Join Conditions, as Select Items, Group By Expressions etc. Only when the correlations are Filter predicates is the Algebraic transformation worked out. So these e.g.s are not supported:

```
-query 1
select ...
from x
where
    x.b in (select u + x.D
    from y
    where y.c = 10
   )
- query 2
select ...
from x
where
    x.b in (select sum(V)
    from y
    where y.c = 10
    group by u + x.D
   )
```

Restriction.11.m A SubQuery predicate that refers to a Parent Query column must be a valid Join predicate. In fact the correlating Operator must be the Equal Operator(we won't allow the Equal Nulls Same operator). This is because in case the SubQuery contains a Group By (or we introduce a Group By) applying the predicate as a PostJoin filter is not always Semantically equal to applying the Filter and then doing a Group By. For e.g. the 2 queries don't give the same result:

```
-query 1
select ...
from x
where
    a in (select sum(u) as s
    from y
    where y.c = x.b and y.d > x.c
   )
- rewritten query
select ...
from x left semi join
  (select c, sum(u) as s
   from y
  ) sq1 sq1.c = x.b and sq1.s = x.a
where sq1.d > x.c
```

Check.12.h SubQuery predicates cannot only refer to Parent Query columns. Though we can translate this, we will flag this as an error because allowing such predicates are not useful; user should specify them on the Parent Query Block. Allowing them may have implications on Algebraic transformation. For e.g. the following query is flagged as an error:

```
-query 1
select ...
from x
where
    a in (select sum(u) as s
    from y
    where y.c = x.b and x.c > 10
   )
```

Restriction.13.m In the case of an implied Group By on a correlated SubQuery, the SubQuery always returns 1 row. For e.g. a count on an empty set is 0, while all other UDAFs return null. Converting such a SubQuery into a Join by computing all Groups in one shot, changes the semantics: the Group By SubQuery output will not contain rows for Groups that don't exist.

For e.g. running the following query(in mysql) on tpcds.items table with 18k rows, returns 18k:

```
SELECT count(*)
FROM tpcds.item i
where exists (select sum(i_wholesale_cost)
      from item j
      where i.i_item_sk = j.i_item_sk and 1 = 2) ;
```

Where as this one returns 0:

```
SELECT count(*)
FROM tpcds.item i join (select j.i_item_sk, sum(i_wholesale_cost)
from item j
where 1 = 2 group by j.i_item_sk  ) as s
 on i.i_item_sk = s.i_item_sk
```

Given that:

- An Exists Query with an Aggregated SubQuery with no GroupBy will always return all the Parent Query's rows.
- For Not Exists, under similar conditions, no Parent Query rows will be returned.

We propose to check for this condition and not allow it.

Restriction.14.h Correlated Sub Queries cannot contain Windowing clauses. For e.g. the following are not equivalent.

```
-- original query
select ..
from x
where x.a in (select  lag(y.s, 5) over (partition by y.v) as l,
      from y
      where x.b = y.t
      )
-- rewritten query
select ..
from x left outer join
     ( select lag(y.s, 5) over (partition by y.v) as l,
   y.t
       from y
       group by y.t ) sq1 on x.b = sq1.t and x.a = y.l
```

The lag in the rewritten query is on all groups. What we need to do is rewrite as:

```
-- rewritten query
select ..
from x left outer join
     ( select lag(y.s, 5) over (partition by y.t, y.v) as l,
   y.t
       from y
       group by y.t ) sq1 on x.b = sq1.t and x.a = y.l
```

That is push the Group By into the partition by definition of each Over Clause. We will not support this.

Restriction.15.h all unqualified column references in a SubQuery will resolve to table sources within the SubQuery. So we will not attempt to resolve an unqualified column references to the Parent Query. Makes the rules consistent, Restriction.6.m that requires Parent Query column references be qualified. So for e.g.

```
select ..
from x
where x.a in (select b
      from y
      where a > 10)
```

The 'a' in 'a $>$ 10' will be resolved against table y. No attempt will be made to check that 'a' is a column of table 'x'.

2 Semantics of Algebraic transformations

In this section we document the Algebraic transformations and why they are semantically equivalent to evaluating the Sub Query on each Parent Query Tuple.

2.1 In Operator

2.1.1 UnCorrelated SubQuery

The Query is converted to a Left SemiJoin on the SubQuery expression. For e.g.

```
-- original query
select C
from R1
where R1.A in (Select B from R2)
-- rewritten query
Select C
from R1 left semijoin R2 on R1.A = R2.B
```

There are no Restrictions on what can appear in the SubQuery. The semantics work because:

- We apply a Left SemiJoin on the Join output of the Parent Query, i.e. on top any Joins that may be in the Parent Query.
- The Left Semi Join outputs Parent Query tuples that match at least 1 row in the SubQuery. This exactly what the naive evaluation will return.

2.1.2 Correlated Sub Query

Convert the Query to a Left Semi Join on the SubQuery expression and the correlated predicate. For e.g.

```
-- original query
select C
from R1
where R1.A in (Select B from R2 where R1.X = R2.Y)
-- rewritten query
select C
from R1 left semi join
    (select B, R2.Y as sq1_col0 from R2) sq1
      on R1.X = sq1.sq1_col0 and R1.A = sq1.B
```

Without a Group By or Distinct this transformation is Semantically valid:

- the *Left Semi Join* operation on the correlated predicates And the Subquery expression outputs only Parent Query tuples that that have 1 or more rows in the SubQuery that match this row. Which is exactly what the SubQuery expression implies.

2.1.3 Correlated Sub Query, in the presence of a Group By

In this case we add the SubQuery expression in a Correlation predicate to the SubQuery's Group By Expressions and as an Select Item. For e.g.

```
-- original query
select C
from R1
```

```
where R1.A in (Select sum(B) as s from R2 where R1.X = R2.Y)
-- rewritten
select C
from R1 left semi join
  (select R2.Y as sq1_col0, sum(B) as s
   from R2
   group by R2.Y
  ) sq1
  on R1.X = sq1.Y and R1.A = sq1.s
```

This works semantically because:

- instead of computing the SubQuery for each Parent tuple, we compute Sub Query Groups for each correlated value.
- the rewritten Sub Query will return at most one row per value of the correlated values.
- We only return Parent Query rows that match on the correlation predicate and also only if the SubQuery expression equals the Aggregated value on the correlated group.
- This is exactly what the SubQuery is specifying in most cases. But see **Restriction.13.m**.

2.1.4 Correlated Sub Query, in the presence of a Distinct

Given these constraints:

- Hive doesn't allow a Query to contain a Distinct and Group By.
- Hive doesn't allow a distinct Query to have a UDAF in a SelectItem.

We treat this as the simple Correlated case: without Group By. For e.g.

```
-- original query
select C
from R1
where R1.A in (Select distinct B as s from R2 where R1.X = R2.Y)
-- rewritten
select C
from R1 left semi join
  (select distinct R2.Y as sq1_col0, B as s
   from R2
  ) sq1
  on R1.X = sq1.Y and R1.A = sq1.s
```

The Select distinct ensures that:

- if for a particular value of R1.X the SubQuery would return 'n' rows => the rewritten SQ1 query would return 'n' rows with sq1_col0 = that value of R2.Y
- if for a particular value of R1.X the SubQuery would return no rows => the rewritten SQ1 query would not return a row with sq1_col0 = that value of R2.Y

2.2 Not In Operator

2.2.1 Not In semantics in the presence of Nulls

[This is thanks to Sivaramakrishnan Narayanan from Qubole. See his comment on Hive Jira 784.]

Consider tables T1 and T2 with the following data:

T1.x
1
2
null

T2.y
1
null

Now consider the not-in subquery

```
select * from T1 where T1.x not in (select y from T2)
```

This should produce an empty result set. Because of the null row in T2 the 'not equal ALL' check fails for every row in T1. A value is not in a set if it is *not equal to ALL* values in the set.

Therefore when we convert a Not In operator Algebraically we must add this check. We do this by joining in a SubQuery that counts the number of rows with nulls for the joining columns. If this count is not 0 we should return no rows.

2.2.2 UnCorrelated SubQuery

Algebraically we convert this into an *Anti-Join* operation. Since Hive doesn't have Ant-Join, we convert this into a Left Outer Join with a is null Post Join Filter. For e.g.

```
-- original query
select C
from R1
where R1.A not in (Select B from R2)
-- rewritten query
Select C
from R1 join
    (
     select count(*) as c1
     from R2
     where B is null
    ) sq1
    on sq1.c1 = 0 left outer join
    ( select B
      from R2) sq2
     on R1.A = sq2.B
where sq2.B is null
```

The semantics work because:

- *Anti Join* is what the Uncorrelated Not In is specifying: return rows from the Parent Query that don't have any matching rows with the Inner Query.
- Since Hive doesn't have a Anti-Join function we express this as a Left Outer Join followed by a null check post join condition.
  - A Parent row with null value in the SubQuery Expression should be in the output.
  - Since we only join on Equality, a null value in the SubQuery's select item implies that the Parent Query row didn't join with any SubQuery row.
- The join with the *null count* SubQuery ensures that when there are nulls in the Join column of the SubQuery, we return no rows.

2.2.3 Correlated Sub Query

This is similar to the Uncorrelated case, except that the Join Condition includes the correlated predicates. For e.g.

```
-- original query
select C
from R1
where R1.A not in (Select B from R2 where R1.X = R2.Y)
-- rewritten query
Select C
from R1 join
    (
     select count(*) as c1
     from R2
     where B is null
    ) sq1
    on sq1.c1 = 0 left outer join
    ( select B,
      R2.Y as sq2_col0
      from R2) sq2
     on R1.A = sq2.B and R1.X = sq2.sq2_col0
where sq2.B is null
```

Without a Group By in the Sub Query, semantically this is just an Anti-Join and works for the same reason as the section above.

2.2.4 Correlated Sub Query, in the presence of a Group By

As in the case of the In Operator, we add the SubQuery expression in a Correlation predicate to the SubQuery's Group By Expressions and as an Select Item. For e.g.

```
-- original query
select C
```

```
from R1
where R1.A not in (Select sum(B) as s from R2 where R1.X = R2.Y)
-- rewritten
select C
from R1 join
  (
    select count(*) as c1
    from (select sum(B) as s from R2 group by R2.Y) sq1.1
    where s is null
  ) sq1 on sq1.c1 = 0 left outer join
  (select R2.Y as sq2_col0, sum(B) as s
   from R2
   group by R2.Y
  ) sq2
  on R1.X = sq2.Y and R1.A = sq2.s
where sq2.sq2_col0 is null
```

This works semantically, because:

- instead of computing the SubQuery for each Parent tuple, we compute Sub Query Groups for each correlated value.
- Consider the following cases:

R1.A	R1.X	R2.sum(B)	R2.y	output
5	3	7	5	yes because rows will not join
5	3		no row with value 5	yes because no joining Group from SubQuery
5	null	any	5	yes row will not join with SubQuery group row where R2.y = 5
null	3		any	yes row will not join with any SubQuery group

2.2.5 Correlated Sub Query, in the presence of a Distinct

Similar to the Distinct case for the In Operator. For e.g.

```
-- original query
select C
from R1
where R1.A not in (Select distinct B as s from R2 where R1.X = R2.Y)
-- rewritten
select C
from R1 join
  (
    select count(*) as c1
    from R2
```

```
    where B is null
  ) sq1 on sq1.c1 = 0 left outer join
  (select distinct R2.Y as sq1_col0, B as s
   from R2
  ) sq1
  on R1.X = sq1.Y and R1.A = sq1.s
where  sq1.sq1_col0 is null
```

2.3 Exists Operator

Exists has the same semantics as a correlated In Operator without the SubQuery Expression. So for e.g. a exists without Group By is rewritten as :

```
-- original query
select C
from R1
where exists (Select B from R2 where R1.X = R2.Y)
-- rewritten query
select C
from R1 left semi join
    (select B, R2.Y as sq1_col0 from R2) sq1
      on R1.X = sq1.sq1_col0
```

Apart from the missing SubQuery Expression an Exists is transformed in the same way as a In Operator.

2.4 Not Exists Operator

Not Exists has similar semantics as a correlated Not In Operator without the SubQuery Expression. The one differences in that we don't need to check if the SubQuery matching expression contains a null. So for e.g. a not exists with Group By is rewritten as :

```
-- original query
select C
from R1
where not exists (Select sum(B) as s from R2 where R1.X = R2.Y)
-- rewritten
select C
from R1 left outer join
  (select R2.Y as sq1_col0, sum(B) as s
   from R2
   group by R2.Y
  ) sq1
  on R1.X = sq1.Y
where sq1.sq1_col0 is null
```

Apart from the missing SubQuery Expression a Not Exists is transformed in the same way as a Not In Operator.

3 Support for SubQueries in the Having clause

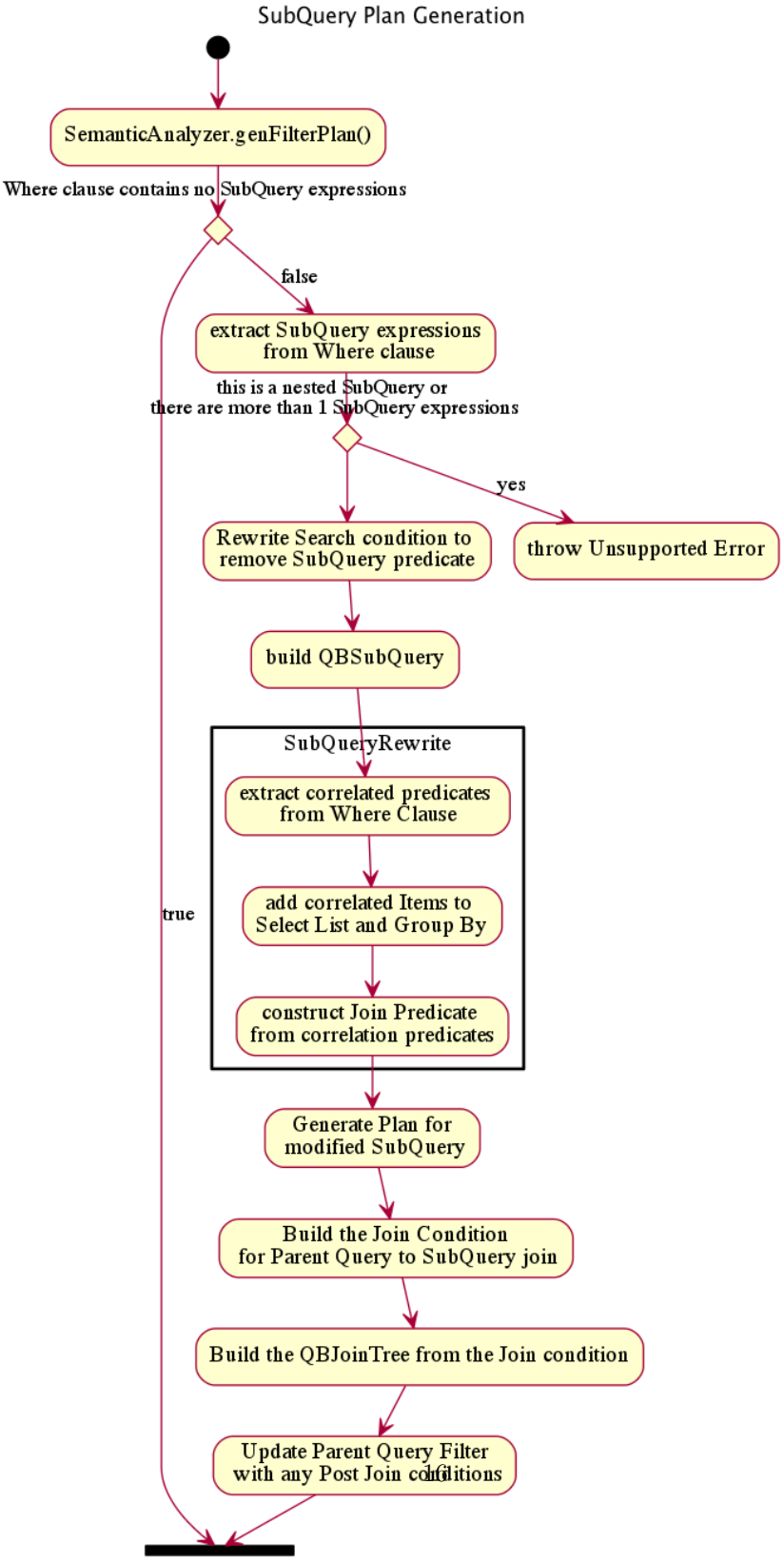
Algebraic transformations for SubQueries in the Having clause work just like the Where Clause. The one detail we need to handle is enabling correlating on aggregation expressions. For e.g.

```
select b.key, min(b.value)
from src b
group by b.key
having exists ( select a.key
from src a
where a.value > 'val_9' and a.value = min(b.value)
)
```

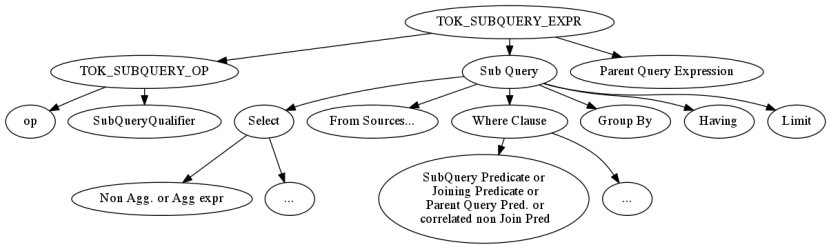
In the case of having we attempt to resolve a joining condition with the Outer Query's RowResolver; we check if the condition is an expression known to the Outer Query's RowResolver.

4 Implementation Details

4.1 Overall flow for SubQuery Plan generation



4.2 SubQuery Expression AST structure



4.3 Sub Query rewrite

4.3.1 Track UDAF and Windowing invocations in SubQuery

- go over every Select Item AST
- if we encounter a UDAF invocation track it as a Aggregation or Windowing invocation depending of presence of OVER clause.

4.3.2 SubQuery rewrite

- If the SubQuery has no where clause, there is nothing to rewrite.
- Decompose SubQuery where clause into list of Top level conjuncts.
- For each conjunct
  - Break down the conjunct into (LeftExpr, LeftExprType, RightExpr, RightExprType)
  - If the top level operator is an Equality Operator we will break it down into left and right; in all other case there is only a lhs.
  - The ExprType is based on whether the Expr refers to the Parent Query Query Sources, refers to the SubQuery sources or both.
  - We resolve an unqualified Column reference against the Parent Query's RowResolver first; so our resolution rules are different from the nested scope rules implied by the SQL spec.
  - If the lhs or rhs expr refers to both Parent and SubQuery sources, we flag this as Unsupported.
  - If the conjunct as whole, only refers to the Parent Query sources, we flag this as n Error.
  - A predicate is Correlated if lhs refers to SubQuery and rhs refers to Parent Query or the reverse.
  - Say the lhs refers to SubQuery and rhs refers to Parent Query; the other case is handled analogously.
    - remove this from the SubQuery where clause.
    - for the SubQuery expression(lhs) construct a new alias
    - in the correlated predicate, replace the SubQuery expression(lhs) with the alias AST.

    - add this altered predicate to the Join predicate tracked by the QBSubQuery object.
    - add the alias AST to a list of subQueryJoinAliasExprs. This list is used in the case of Outer Joins to add null check predicates to the Parent Query's where clause.
    - Add the SubQuery expression with the alias as a SelectItem to the SubQuery's SelectList.
    - In case this query has a UDAF invocation add this SubQuery expression to the GroupBy; add it to the front of the GroupBy.
  - If predicate is not correlated, let it remain in the SubQuery where clause.

4.3.3 SubQuery build Join and postJoin conditions

Details are in the Algebraic transformation section. Here we provide a summary table.

Operator	Non Agg Non Corr	Non Agg Corr	Agg Non Corr	Agg Corr
In	JoinCond: SQ.outerExpr = SQ.selectItem 0 JoinType: Left Semi	JoinCond: SQ.outerExpr = SQ.selectItem 0 and SQ.correlationPredicates JoinType: LOJ SQ Select: add SQ expressions from correlated predicates	JoinCond: same as (N,N, In) JoinType: Left Semi	JoinCond: JoinType: SQ Select: same as (N, C, In) SQ GBy: add SQ exprs from corr. preds.
Not In	JoinCond: JoinType: same as (N,N, In) PostJoinCond: SQ.selectItem 0 is null	JoinCond: JoinType: SQ Select: same as (N, C, In) Post Join Cond: same as (N,N, Not In)	JoinCond: JoinType: PostJoinCond: same as (N,N, Not In)	JoinCond: JoinType: PostJoinCond: same as (N,C, Not In)
Exists	Error	similar to (N,C, In)	Error	similar to (A,C, In)
Not Exists	Error	similar to (N,C,Not In)	Error	similar to (A,C, Not In)

4.3.4 Building the Join Operator between the Parent Query and SubQuery

1. QBJoinTree construction: Setup a QBJoinTree between a SubQuery and its Parent Query. The Parent Query is the lhs of the Join. The Parent Query is represented by the last Operator needed to process its From Clause. In case of a single table Query this will be a TableScan, but it can be a Join Operator if the Parent Query contains Join clauses, or in case

of a single source from clause, the source could be a SubQuery or a PTF invocation.

We setup the QBJoinTree with the above constraints:

- the lhs of the QBJoinTree can be a another QBJoinTree if the Parent Query operator is a JoinOperator. In this case we get its QBJoinTree from the 'joinContext'
- the rhs is always a reference to the SubQuery. Its alias is obtained from the QBSubQuery object.

The QBSubQuery also provides the Joining Condition AST. The Joining condition has any correlated predicates and a predicate for joining the Parent Query expression with the SubQuery. The QBSubQuery also specifies what kind of Join to construct. Given this information, once we initialize the QBJoinTree, we call the 'parseJoinCondition' method to validate and parse Join conditions.

2. Join Operator construction: We altered the *genJoinOperator* to accept the lhs JoiningOperator. If passed in it is used as the lhs Operator instead of calling genJoinOperator recursively on the JoinSrc from the QBJoinTree. We need to do this because the JoinOperator for the left subtree of this QBJoinTree has already has had its JoinOperators constructed. This happened during the processing of the from Clause of the Parent Query.

4.4 Explain Sub Query Transformation

Because there are so many rules for Algebraic transformation, we propose adding new statement to hive: **explain subquery rewrite sql_statement**. This will list the transformations applied and also the transformed query that will be evaluated.

5 References

[Dayal87] Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers, Umeshwar Dayal, VLDB 1987.

[Kim82] On Optimizing an SQL-like Nested Query, Won Kim, TODS 82.