Transformations of Existential Subqueries using Early-out Joins

1.0 Requirements

1.1 Problem Statement

Existential subqueries are constructs that are commonly used to filter rows from a table. They typically take the form

```
SELECT <cols/expressions>
FROM   T
WHERE  <expression> IN (SOME subquery SQ)
```

This query includes an <expression> that involves columns from table T (called the Source) and SQ is any query that returns a result set of rows that can be matched/compared to <expression>. SQ is called the “Filtering Source”. The semantics of this SQL-construct is that the query above produces <cols/expressions> for each row in the Source table where the corresponding <expression> for that row exists in the result set produced by the filtering source. Presto processes these queries by simply converting this operation into a semi-join. A semi-join in Presto is realized during query execution as a special operator and has certain advantages in performance over regular joins. In this document we propose a technique to improve the performance and scalability of existential subqueries by rewriting them to other logically identical formats.

1.2 A Note on Semi-Joins

Semi-joins are a special kind of join algorithm. A semi-join’s purpose is to filter a rowset based on its inclusion in another rowset. A semi-join of the form “A semi-join B” where A is the source and B is the filtering source, must satisfy the following conditions.

a. The join operator must include each row from A that has a match with B on the join condition prescribed in the query
b. Each row from A can appear at most once in the output of the join

At execution time, semi-joins are typically processed ignoring duplicate values in B. I.e most database engines search for each value in A, a corresponding match in B, but halt the search of the Filtering Source’s input

stream when it encounters a match. In Presto execution this is realized by a special operator called the HashSemiJoinOperator which constructs a hash table out of the filtering source input set, ignoring the hash collisions (it drops duplicates from the build side). Presto then probes this hash set with the input from the Source (A). Rows from the Source that match are produced as output. Typically, in a hash join operation we would prefer that the hash table is constructed out of the join input that is smaller. This is beneficial for two reasons - a) it imposes less pressure on memory, since the hash table has to be maintained in memory, and b) it improves performance since the construction of the hash table can be time-consuming when the input is large. In the version of the semi-join algorithm that is implemented in Presto, it is not possible to choose which join input to build the hash set on since the duplicates may only be ignored on the Filtering Source's values, and therefore the Filtering Source, regardless of size, will always be the build input to the semi-join.

A semi-join is one instance of what we will call an "Early-Out Join" where the search for a matching tuple may be halted as soon as one match is found. A semi-join is therefore a left early-out join where the probe from the left input to the join may exit early if successful.

1.3 Join Reordering in Presto

The most commonly used join, Inner-Join "A join B on (condition)", is an operation that is required to produce all rows from A and B that match each other on (condition). While this is also processed as a hash join in Presto, the choice of which join input to construct a hash table from, and which input to probe with, is deliberately made in an informed manner by the Cost-Based Optimizer (CBO). Reordering of inputs is possible since this operation is symmetric. The CBO therefore makes a statistics-based decision and judiciously chooses the smaller input of the join as the build side (i.e. to construct the hash table from).

1.4 Proposed Solution

Join reordering is available only for inner joins in Presto. Therefore we want to devise a framework that allows conversion of existential queries to Inner-joins in order to avail of the flexibility to reorder join inputs where beneficial. In some cases, it is entirely possible that the original plan, realized as the semi-join, is the most optimal (i.e. the result set from the filtering source is small),

and we want to be able to retain that option. In this document we will lay out a set of query rewrites that we believe will allow us the best of both worlds.

2.0 Externals

This feature utilizes two other features that are already controlled by flags in Presto - Cost-Based join reordering, and the constraint-based optimization framework. Therefore in order for this feature to be effective, the end-user would have to enable three distinct flags

1. join_reordering_strategy='AUTOMATIC'
   [optimizer.join-reordering-strategy]
2. exploit_constraints=true [optimizer.exploit-constraints]
3. early_out_join_transformations_enabled=true (NEW)
   [optimizer.early-out-join-transformations-enabled]

We will also introduce another parameter to govern whether aggregations should be pushed below the join. More details on this in section 4

early_out_join_byte_reduction_threshold = 1 (default) (NEW)
[optimizer.early-out-join-transformation-byte-reduction-threshold]

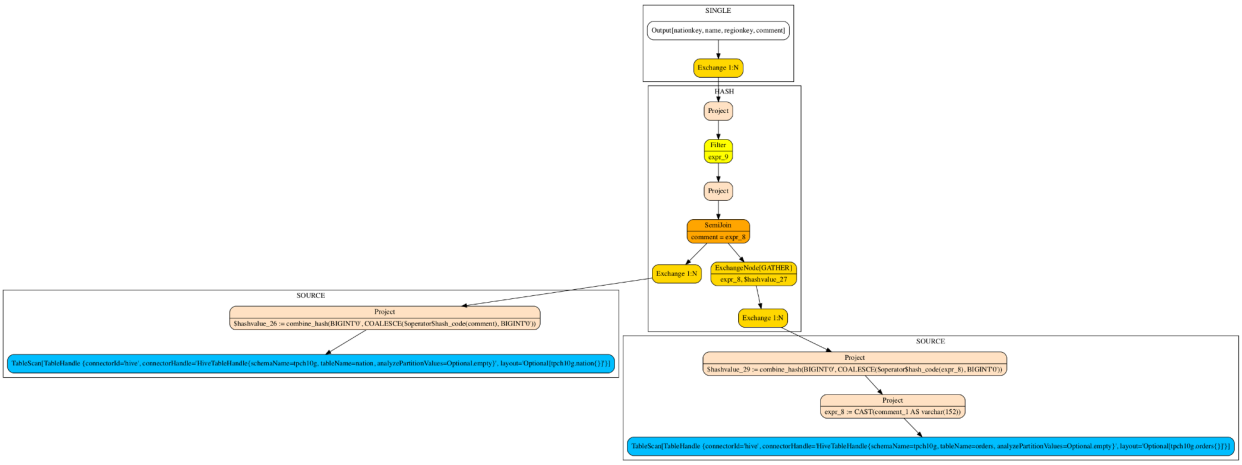
3.0 High Level Design

As previously mentioned Presto directly converts an existential query into a semi-join, and no other optimizer rule apply for this case to further transform/optimize this query pattern[1]. The obvious drawback to implicitly treating an existential query as a semi-join is the poor performance that stems from the filtering source being large. Consider the following simple query on tpch data and its corresponding query plan.

```
SELECT *
FROM   customer
WHERE  custkey IN (SELECT custkey
                   FROM   orders)
```

[1] correlated IN predicates are converted to Left Outer Joins - but that is not relevant to this discussion

```
AND NAME = 'Customer#000156251'
```



The semijoin attempts to always use “orders” on the build side and this query can run poorly or even fail due to resource constraints. For e.g. this query would fail on Presto if you limit the memory to the process to 2G on a 10G tpch-schema

-- STRAIGHT UP SEMI JOIN FAILS
presto:tpch10g> select * from customer where custkey in (select custkey from orders) and name = 'Customer#000156251';

Query 20220624_184452_00002_h3qwz, FAILED, 4 nodes
Splits: 64 total, 31 done (48.44%)
0:04 [9.62M rows, 62.7MB] [2.54M rows/s, 16.6MB/s]

Query 20220624_184452_00002_h3qwz failed: Java heap space

The corresponding query plan is

presto:tpch10g> explain select * from customer where custkey in (select custkey from orders) and name = 'Customer#000156251';

Query Plan

- Output[custkey, name, address, nationkey, phone, acctbal, mktsegment, comment] => [custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117)]

```
        Estimates: {rows: 1 (195B), cpu: 1519452955.84, memory: 270000000.00, network: 270000400.89}
    - RemoteStreamingExchange[GATHER] => [custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint,
phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117)]
            Estimates: {rows: 1 (195B), cpu: 1519452955.84, memory: 270000000.00, network: 270000400.89}
        - FilterProject[filterPredicate = expr_10, projectLocality = LOCAL] => [custkey:bigint, name:varchar(25),
address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10),
comment:varchar(117)]
                Estimates: {rows: 1 (195B), cpu: 1519452760.00, memory: 270000000.00, network: 270000205.05}/{rows: 1
(195B), cpu: 1519452955.84, memory: 270000000.00, network: 270000205.05}
            - Project[projectLocality = LOCAL] => [custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint,
phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), expr_10:boolean]
                    Estimates: {rows: 1 (197B), cpu: 1519452562.11, memory: 270000000.00, network: 270000205.05}
                - SemiJoin[custkey = custkey_1][$hashvalue, $hashvalue_44] => [custkey:bigint, name:varchar(25),
address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10),
comment:varchar(117), $hashvalue:bigint, expr_10:boolean]
                        Estimates: {rows: 1 (207B), cpu: 1519452364.23, memory: 270000000.00, network: 270000205.05}
                        Distribution: PARTITIONED
                    - RemoteStreamingExchange[REPARTITION][$hashvalue] => [custkey:bigint, name:varchar(25),
address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10),
comment:varchar(117), $hashvalue:bigint]
                            Estimates: {rows: 1 (205B), cpu: 574451952.09, memory: 0.00, network: 205.05}
                        - ScanFilterProject[table = TableHandle {connectorId='hive',
connectorHandle='HiveTableHandle{schemaName=tpch10g, tableName=customer,
analyzePartitionValues=Optional.empty}', layout='Optional[tpch10g.customer{domains={name=[
[["Customer#000156251"]] ]}}]'}, filterPredicate = (name) = (VARCHAR'Customer#000156251'), projectLocality = LOCAL]
=> [custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double,
mktsegment:varchar(10), comment:varchar(117), $hashvalue_43:bigint]
                                Estimates: {rows: 1500000 (286.79MB), cpu: 287225771.00, memory: 0.00, network: 0.00}/{rows: 1
(205B), cpu: 574451542.00, memory: 0.00, network: 0.00}/{rows: 1 (205B), cpu: 574451747.05, memory: 0.00, network:
0.00}
                                $hashvalue_43 := combine_hash(BIGINT'0', COALESCE($operator$hash_code(custkey),
BIGINT'0')) (1:23)
                                LAYOUT: tpch10g.customer{domains={name=[ [["Customer#000156251"]] ]}}
                                comment := comment:varchar(117):7:REGULAR (1:23)
                                acctbal := acctbal:double:5:REGULAR (1:23)
                                nationkey := nationkey:bigint:3:REGULAR (1:23)
                                name := name:varchar(25):1:REGULAR (1:23)
                                custkey := custkey:bigint:0:REGULAR (1:23)
                                phone := phone:varchar(15):4:REGULAR (1:23)
                                mktsegment := mktsegment:varchar(10):6:REGULAR (1:23)
                                address := address:varchar(40):2:REGULAR (1:23)
                    - LocalExchange[SINGLE] () => [custkey_1:bigint, $hashvalue_44:bigint]
                            Estimates: {rows: 15000000 (257.49MB), cpu: 675000000.00, memory: 0.00, network: 270000000.00}
                        - RemoteStreamingExchange[REPARTITION - REPLICATE NULLS AND ANY][$hashvalue_45] =>
[custkey_1:bigint, $hashvalue_45:bigint]
                                Estimates: {rows: 15000000 (257.49MB), cpu: 675000000.00, memory: 0.00, network:
270000000.00}
                            - ScanProject[table = TableHandle {connectorId='hive',
connectorHandle='HiveTableHandle{schemaName=tpch10g, tableName=orders,
analyzePartitionValues=Optional.empty}', layout='Optional[tpch10g.orders{}]'}, projectLocality = LOCAL] =>
[custkey_1:bigint, $hashvalue_46:bigint]
                                        Estimates: {rows: 15000000 (257.49MB), cpu: 135000000.00, memory: 0.00, network: 0.00}/{rows:
15000000 (257.49MB), cpu: 405000000.00, memory: 0.00, network: 0.00}
                                        $hashvalue_46 := combine_hash(BIGINT'0', COALESCE($operator$hash_code(custkey_1),
BIGINT'0')) (1:70)
                                        LAYOUT: tpch10g.orders{}
                                        custkey_1 := custkey:bigint:1:REGULAR (1:70)

(1 row)
```

In comparison we can see that a logically equivalent query on the same setup succeeds[2].

```
SELECT DISTINCT c.*
FROM   (SELECT uuid(),
               *
        FROM   customer
        WHERE  NAME = 'Customer#000156251') c,
       orders o
WHERE  c.custkey = o.custkey;
```

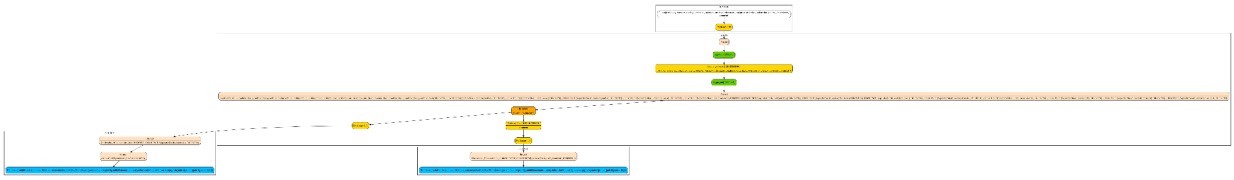
```
presto:tpch10g> explain SELECT DISTINCT c.*
        -> FROM   (SELECT Random(),
        ->               *
        ->        FROM   customer
        ->        WHERE  NAME = 'Customer#000156251') c,
        ->       orders o
        -> WHERE  c.custkey = o.custkey;

Query Plan
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------
 - Output[_col0, custkey, name, address, nationkey, phone, acctbal, mktsegment, comment] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117)]
        _col0 := random (1:9)
    - RemoteStreamingExchange[GATHER] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117)]
        - Project[projectLocality = LOCAL] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117)]
            - Aggregate(FINAL)[random, custkey, name, address, nationkey, phone, acctbal, mktsegment, comment][$hashvalue] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue:bigint]
                - LocalExchange[HASH][$hashvalue] (random, custkey, name, address, nationkey, phone, acctbal, mktsegment, comment) => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue:bigint]
                    - Aggregate(PARTIAL)[random, custkey, name, address, nationkey, phone, acctbal, mktsegment, comment][$hashvalue_80] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue_80:bigint]
                        - Project[projectLocality = LOCAL] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue_80:bigint]
                                Estimates: {rows: 15 (3.09kB), cpu: 1519458600.45, memory: 214.25, network: 270000214.25}
                                $hashvalue_80 := combine_hash(combine_hash(combine_hash(combine_hash(combine_hash(combine_hash(combine_hash(combine_hash(combine_hash(BIGINT'0', COALESCE($operator$hash_code(random), BIGINT'0')), COALESCE($operator$hash_code(custkey), BIGINT'0')), COALESCE($operator$hash_code(name), BIGINT'0')), COALESCE($operator$hash_code(address), BIGINT'0')), COALESCE($operator$hash_code(nationkey), BIGINT'0')), COALESCE($operator$hash_code(phone), BIGINT'0')), COALESCE($operator$hash_code(acctbal), BIGINT'0')), COALESCE($operator$hash_code(mktsegment), BIGINT'0')), COALESCE($operator$hash_code(comment), BIGINT'0')) (2:16)
                            - InnerJoin[("custkey_34" = "custkey")][$hashvalue_75, $hashvalue_77] => [random:double, custkey:bigint, name:varchar(25), address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117)]
                                    Estimates: {rows: 15 (2.96kB), cpu: 1519455431.65, memory: 214.25, network: 270000214.25}
                                    Distribution: PARTITIONED
```

[2] We use random() instead of uuid() in the examples since distinct is not supported on uuid yet. But this illustrates the efficacy of the approach

```
                    - RemoteStreamingExchange[REPARTITION][$hashvalue_75] => [custkey_34:bigint, $hashvalue_75:bigint]
                            Estimates: {rows: 15000000 (257.49MB), cpu: 675000000.00, memory: 0.00, network: 270000000.00}
                        - ScanProject[table = TableHandle {connectorId='hive', connectorHandle='HiveTableHandle{schemaName=tpch10g,
tableName=orders, analyzePartitionValues=Optional.empty}', layout='Optional[tpch10g.orders{}]'}, projectLocality = LOCAL] => [custkey_34:bigint,
$hashvalue_76:bigint]
                                Estimates: {rows: 15000000 (257.49MB), cpu: 135000000.00, memory: 0.00, network: 0.00}/{rows: 15000000 (257.49MB),
cpu: 405000000.00, memory: 0.00, network: 0.00}
                                $hashvalue_76 := combine_hash(BIGINT'0', COALESCE($operator$hash_code(custkey_34), BIGINT'0')) (6:8)
                                LAYOUT: tpch10g.orders{}
                                custkey_34 := custkey:bigint:1:REGULAR (6:8)
                    - LocalExchange[HASH][$hashvalue_77] (custkey) => [random:double, custkey:bigint, name:varchar(25), address:varchar(40),
nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue_77:bigint]
                            Estimates: {rows: 1 (214B), cpu: 574452184.75, memory: 0.00, network: 214.25}
                        - RemoteStreamingExchange[REPARTITION][$hashvalue_78] => [random:double, custkey:bigint, name:varchar(25),
address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue_78:bigint]
                                Estimates: {rows: 1 (214B), cpu: 574451970.50, memory: 0.00, network: 214.25}
                            - ScanFilterProject[table = TableHandle {connectorId='hive', connectorHandle='HiveTableHandle{schemaName=tpch10g,
tableName=customer, analyzePartitionValues=Optional.empty}', layout='Optional[tpch10g.customer{domains={name=[ [["Customer#000156251"]] ]}}]'},
filterPredicate = (name) = (VARCHAR'Customer#000156251'), projectLocality = LOCAL] => [random:double, custkey:bigint, name:varchar(25),
address:varchar(40), nationkey:bigint, phone:varchar(15), acctbal:double, mktsegment:varchar(10), comment:varchar(117), $hashvalue_79:bigint]
                                    Estimates: {rows: 1500000 (299.67MB), cpu: 287225771.00, memory: 0.00, network: 0.00}/{rows: 1 (214B), cpu:
574451542.00, memory: 0.00, network: 0.00}/{rows: 1 (214B), cpu: 574451756.25, memory: 0.00, network: 0.00}
                                    random := random()
                                    $hashvalue_79 := combine_hash(BIGINT'0', COALESCE($operator$hash_code(custkey), BIGINT'0')) (4:17)
                                    LAYOUT: tpch10g.customer{domains={name=[ [["Customer#000156251"]] ]}}
                                    comment := comment:varchar(117):7:REGULAR (4:16)
                                    acctbal := acctbal:double:5:REGULAR (4:16)
                                    nationkey := nationkey:bigint:3:REGULAR (4:16)
                                    name := name:varchar(25):1:REGULAR (4:16)
                                    custkey := custkey:bigint:0:REGULAR (4:16)
                                    phone := phone:varchar(15):4:REGULAR (4:16)
                                    mktsegment := mktsegment:varchar(10):6:REGULAR (4:16)
                                    address := address:varchar(40):2:REGULAR (4:16)

(1 row)
```



```
presto:tpch10g> select distinct c.* from (select random(), * from customer where
name = 'Customer#000156251') c, orders o where c.custkey = o.custkey;
       _col0       | custkey |        name        |        address         | nationkey |
phone      | acctbal | mktsegment |                    comment
-------------------+---------+--------------------+------------------------+-----------+--------------
--+---------+------------+-----------------------------------------------------------------
 0.285896288805213 |  156251 | Customer#000156251 |
urz1DOJ,ZKWJni8FIxmgRBX |         7 | 17-321-701-8875 | -185.91 |
HOUSEHOLD  | , ironic packages are never about the ironic pinto beans. pint
(1 row)
```

```
Query 20220624_184514_00006_h3qwz, FINISHED, 4 nodes
Splits: 92 total, 92 done (100.00%)
0:01 [16.5M rows, 62.7MB] [12.5M rows/s, 47.5MB/s]
```

This is due to the fact that the CBO reorders the inputs to the inner join, and chooses the smaller table to be the build input to the join. Furthermore this is a cardinality-reducing join that produces a small result set (a very common case), which makes the aggregation lightweight. This query rewrite enables the CBO to participate in planning the query and determining the appropriate join order. In the rest of this section we will focus on proving the logical equivalence of this transformation and some further tweaks to ensure that we always pick the best plan based on the available information.

3.1 Logical Equivalence (A)

Let us consider the following query to be the canonical version of the existential query

```
SELECT <cols/expressions>
FROM   a
WHERE  <expression1> IN
       (
              SELECT <expression2>
              FROM   b)
```

It is obvious that this is equivalent to performing a semi-join with A as the data source to the join and B as the filtering source where the matching condition is <expression1> = <expression2>. This is what Presto does today.

We posit that this is equivalent to the following rewrite to an inner join

```
SELECT DISTINCT id,
                sq1.<cols/expressions>
```

```
FROM            (
                        SELECT uuid() AS id,
                               <cols/expressions>,
                              <expression1>
                        FROM   a) sq1,
                (
                        SELECT <expression2>
                        FROM   b) sq2
WHERE           sq1.<expression1> = sq2.<expression2;
```

We previously discussed that the semi-join ignores duplicates from the filtering source (B) and just performs a check for existence for each element in A in the result set of B. In the above rewrite the join is transformed to an inner join where all matching rows in A and B are produced from the join (1:N join). However, notice the following conditions

1. A unique id is appended to each row of A[3]
2. The output contains only elements from A (uncorrelated subquery)
3. We perform a final distinct aggregation on the result of the join

From these conditions, the following conclusions may be inferred

a. Rows in A that do not match any row in B on the expressions will not appear in the output - from the definition of inner join.
b. For every row of A that has more than one match in B, the output will have the same value for the “id” column.
c. Since the output columns are a strict subset of the columns in A, the distinct aggregation is guaranteed to remove all rows in A that have the same value for id, but also retains rows from A that are duplicates otherwise.
d. An additional nuance here is that nulls are never considered equivalent (i.e. NULL != NULL) and nulls never match any other value. Therefore rows for which <expression1> in A or <expression2> in B evaluate to NULL will never appear in the join output for either join.

Conclusions (a-d) show that the rewritten query satisfies the semantics of the existential query and is therefore logically equivalent.

3.2 Logical Equivalence (B)

[3] This intermediate “id” column will be removed/suppressed after the aggregation

There is another rewrite that is also equivalent to the canonical version of the existential query

```
SELECT sql1.<cols/expressions>
FROM            (
                        SELECT <cols/expressions>
                        <expression1>
                        FROM   a) sql1,
                (
                        SELECT DISTINCT <expression2>
                        FROM   b) sq2
WHERE           sql1.<expression1> = sq2.<expression2;
```

This rewrite filters out duplicate values of b.<expression2> before the join. Therefore the inner join can only match each row in A with one value from B. This is trivially equivalent to the definition of the existential subquery.

3.3 Logical Equivalence (C)

For completeness we will also include the third equivalent rewrite of the existential subquery - as a semi-join. This is what Presto does today. (Not quite sql syntax)

```
SELECT sql1.<cols/expressions>
FROM            a SEMIJOIN b
WHERE           a.<expression1> = b.<expression2;
```

3.4 The Whole Picture

The previous tpch-example illustrated one instance in which a rewrite of the form 3.1 may be beneficial to query performance. In this section we will

describe various cases where each of the logically equivalent rewrites may be beneficial. We will also contrast our proposal with an alternate approach from Trino that attempts to mitigate the same problem and show how our proposal is better.

Queries involving semi-joins may exhibit variable performance metrics depending on the size of the join inputs and/or the data distribution of the join inputs. Let us enumerate the possible cases that could impact performance here. These mostly have to do with the size of the filtering source join input B, and whether the join significantly reduces cardinality of the output result set.

Case 1: B is smaller than A (Left Early Out Join)

If the input from the filtering source is smaller, we would like to pick that as the build side of the join. In this case it is always better to use a semi-join (rewrite 3.3). Choosing a semi-join here avoids the overhead of additional aggregations and there is no need to reorder the join inputs.

Case 2: B is larger than A (Right Early Out Join)

In this case it is desirable to use A as the build input to the join. Therefore we would like to rewrite this query as an inner join (either 3.1 or 3.2). The difference between these rewrites is that in 3.1 we eliminate duplicate matches on the filtering source (B) after the inner join by performing a distinct aggregation, while in 3.2 we prevent duplicate matches by eliminating duplicates in the filtering source (B) before the join.

Case 2.1: The join is cardinality reducing

If the join reduces cardinality, then the size of the intermediate result set from the inner join is small and the overhead of performing the final aggregation in 3.1 is low. It may be expensive to perform a final distinct on B before the join, especially since the join will also have to build a hash set similar to the aggregation below it. In this case rewrite 3.1 is preferred.

Case 2.2: The join does not reduce cardinality

If the join does not reduce cardinality, then the size of the intermediate result set from the inner join could be large. This could lead to a bigger memory footprint and may incur significant overhead from the final aggregation in 3.1. Therefore a better option may be to use rewrite 3.2 to eliminate duplicates from the filtering source. This may cause a reduction in the intermediate result set (since duplicates in B are removed) but leads to a trade off between performing a distinct aggregation on B vs a distinct aggregation on the inner join output. In this case rewrite 3.2 may be preferred.

4.0 Design Details

This feature will be implemented as a set of optimizer rules that parse the existing query tree/plan and mutate the plan according to one of the three rewrite strategies described above. As previously mentioned three configuration flags or their corresponding session properties will have to be enabled in order for these rules to kick in.

4.1 TransformUncorrelatedInPredicateSubqueryToDistinctInnerJoin

The first rewrite that transforms the IN predicate into the inner join will be an iterative optimizer rule called TransformUncorrelatedInPredicateSubqueryToDistinctInnerJoin. When enabled, this rule will supersede the existing rule TransformUncorrelatedInPredicateSubqueryToSemiJoin and always rewrite an uncorrelated IN predicate to an inner join followed by a distinct aggregation. This rule now opens up the search space and enables the cost-based optimizer to reorder the join inputs as needed.

Rewrite 3.1 adds an “id” to each row in the data source (A) in order to uniquely identify the row and to prevent the distinct aggregation from eliminating duplicate rows in A. This unique id will be added to the plan only if the existing set of inputs from A to the join do not already form a unique key. The constraint optimization framework allows us to easily infer whether this is the case from the LogicalProperties that are computed for each node in the plan.

Thus far in this document we have been referring to the unique id in each row as a uuid(). While this is logically correct, we will implement this using the AssignUniqueId plan node that encapsulates the input and annotates each row with a unique id.

4.2 TransformDistinctInnerJoinToLeftEarlyOutJoin

In this rewrite the optimizer identifies an inner-join immediately below a distinct aggregation. If the two conditions are met, a new distinct aggregation is added to the left input of the join. It is logically correct to add this additional distinct aggregation to the left input of the join if and only if all the columns/expressions from the left input are a part of the grouping keys in the distinct aggregation. We will build upon the equivalence class properties introduced in the constraint-optimization work to perform this check. Additionally we will use the plan node statistics to evaluate whether the join is cardinality reducing. If the size of the output result set is greater than some threshold of the sum of the sizes of the inputs, then we will interpret this join to not be cardinality reducing and push down the distinct aggregation. The threshold here is governed by a configurable parameter called optimizer.early-out-join-transformation-byte-reduction-threshold, whose default is set to 100%. The distinct aggregation above the join may get removed as a result of this pushdown since it may no longer be doing any useful work.

While this is a good rule in and of itself, it can be used to realize the rewrite described in 3.2. In our early-out join world, we expect this rule to kick in when TransformUncorrelatedInPredicateSubqueryToDistinctInnerJoin has converted the IN predicate to an inner-join and the CBO has flipped the join inputs such that the left input to the join is now the filtering source B. Therefore pushing the distinct aggregation below the join (and removing the one above the join) effectively gives us the rewrite in 3.2. Note that if the join is deemed to reduce cardinality effectively we will not push down the aggregation.

4.3 TransformDistinctInnerJoinToRightEarlyOutJoin

This is the final rule that determines whether the distinct aggregation gets pushed down the right input of the inner join. Similar to 4.2, this rule looks for an inner-join immediately below a distinct aggregation and attempts to add an additional distinct aggregation to the right input of the join if and only if all the columns/expressions from the right input are a part of the grouping keys in the distinct aggregation and the outputs of the inner join are the columns/expressions from the left input. In this case, adding a distinct aggregation to the right input, and effectively filtering out duplicates is also what a semi-join does. Therefore in this rule, instead of adding the additional distinct, we can simply convert the inner-join into a semi-join, thereby realizing rewrite 3.3.

In the early-out join world, this rule will kick in when TransformUncorrelatedInPredicateSubqueryToDistinctInnerJoin converts the IN predicate to an inner-join but the CBO determines that the filtering source (B) is the smaller input and leaves the join order unchanged. Now since B is the smaller input it is desirable to use a semi-join operator to dedupe B and perform the join.