Guided Research Report

Thomas Glas
Technische Universität München

1 INTRODUCTION

In our Guided Research, we built a tool that optimizes SQL queries on the SQL-level, with a primary focus on unnesting correlated subqueries. Our tool accomplishes this by parsing SQL and constructing a relational algebra tree, decorrelating any correlated subqueries, and then de-parsing the optimized tree back to SQL. The input to our tool is an SQL query string, and it produces a re-written decorrelated form of the SQL query. As a result, our tool can be used as a optimization layer in front of relational databases, which can lead to considerably better execution times for systems that lack these optimization capabilities.

We built our tool to be a general purpose standalone program, to maximize compatibility with different database systems. As an SQL optimization tool, our tool notably lacks essential metadata such as cardinality estimates or types of available physical join operators, which are often utilized for query optimization. Additionally, as relational algebra is a procedural language, and SQL is a declarative language, our tool's optimization scope is limited to SQL-representable constructs. Despite these limitations, our tool provides a valuable solution for optimizing SQL queries with nested correlated subqueries and achieving significant performance improvements. Our results of applying our tool to the TPC-H benchmark show that the execution times of unoptimized queries, can be reduced from hours to seconds after decorrelating nested subqueries using our tool.

2 BACKGROUND AND RELATED WORK

2.1 SQL & Relational Algebra

SQL and relational algebra are both fundamental concepts for database systems. While SQL is a declarative language that allows users to express their desired output, relational algebra is a procedural language that combines different operators on relational data to produce the desired output defined by SQL. The typical set of operators used in relational algebra include *selections*, *projections* and *joins*. The relationship between SQL and Relational Algebra is important in the design of database systems, as the SQL input to databases is translated into relational algebra, which is followed by query optimizations performed on the relational algebra tree.

2.2 Correlated Subqueries

SQL queries containing correlated subqueries are a common construct used for retrieving data from databases. Correlated subqueries are nested queries that reference attributes from the outer query, and are typically executed once for every tuple returned by the outer query. While correlated subqueries are a useful tool for describing subsets of data in an easy manner, they can have significant performance implications. Because correlated subqueries are executed for every tuple in the outer query, the quadratic increase in runtime complexity leads to slow query performance. A common method to avoid this problem is to rewrite the SQL in a way which avoids correlated subqueries. Ideally, database systems should be able to convert the correlated subqueries into a non-correlated form themselves, but not many database systems have implemented such algorithms.

2.3 Unnesting Arbitrary Queries

In their work, Neumann and Kemper [5] presented an algorithm designed to unnest arbitrary SQL queries. The algorithm introduces the concept of dependent joins as a relational algebra operator, in which the right subtree is dependent on tuples produces by the left subtree. The central idea is to first compute a domain D and transform dependent joins into a dependent join between D and the correlated subquery. Given a dependent join between subtrees T1 and T2, the transformation is defined as the following.

$$D := \Pi_{\mathcal{F}(T_2) \wedge \mathcal{A}(T_1)}(T_1)$$
$$T_1 ⧑_p T_2 \equiv T_1 \bowtie_{p \wedge T_1 = \mathcal{A}(D)} (D ⧑ T_2)$$

The dependent join is then pushed down the relational algebra tree until the right subtree of the dependent join is no longer dependent on the left subtree. The authors define rules for pushing down dependent joins through various relational algebra operators. Once this state is reached, the dependent join can be replaced with a regular join, and the correlated subquery is thus removed from the relational algebra tree. This algorithm is applicable to arbitrary relational algebra trees containing dependent joins, thereby allowing for arbitrary SQL queries to be unnested and decorrelated. As an optional step, after the dependent join has been removed, the join with D can be eliminated if the values produced by D can be replaced with values that already exist in the subtree, further optimizing query performance in most cases.

2.4 Popular Open Source Relational Databases

As per our own testing, we know that many popular relational database systems are not able to effectively decorrelate subqueries. *Umbra* and *DuckDB* are two database systems which have implemented capable decorrelation algorithms. Official documentation for *PostgreSQL*, *SQLite*, *MariaDB* and *MySQL* all don't mention general decorrelation capabilities either. MySQL can decorrelate subqueries, but only with many conditions, such as no *OR* clauses being allowed in the predicate of the *where* clause [3]. MariaDB can decorrelate simple subqueries in *exists* clauses to uncorrelated subqueries in *in* clauses [2]. SQLite attempts subquery flattening, but with many restrictions and no mention of correlated subqueries [4].

3 IMPLEMENTATION

Since our tool builds on top of the PostgreSQL parser, which is written in C, we decided to implement our tool in C++. In this section, we will describe the implementation details of the three phases of our query optimizer: parsing SQL to relational algebra, decorrelating the relational algebra tree, and deparsing relational algebra back to SQL. Some steps need information about which

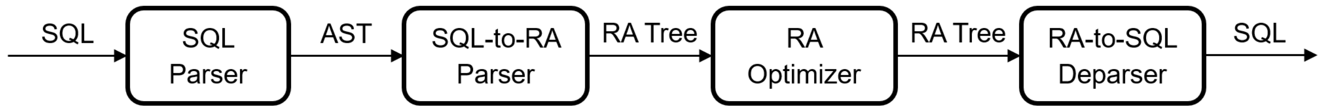


Figure 1: Architecture of our tool. SQL is parsed into an abstract syntax tree (AST) and relational algebra (RA) tree. After the relational algebra tree is optimized, it is deparsesd back to SQL.

attributes belong to which relations. Since our tool does not have any information about the schema of the tables, it relies on the use of aliases to match attributes to relations. Since the TPC-H queries we use for benchmarking do not use relation aliases before attributes, we hard-coded a map from TPC-H attribute prefix to TPC-H table. The high level architecture of our tool is shown in Figure 1.

3.1 Parsing and deparsing SQL

We set the scope for parsing and deparsing to the SQL constructs occuring in TPC-H queries. We chose to work with the PostgreSQL SQL dialect, as it is the most used across databases and we aim to maximizes compatibility of our tool.

3.1.1 Parsing SQL queries to relational algebra. At time of writing, we didn't find any existing open-source PostgreSQL SQL-to-relational algebra parsing tool, so we built one ourselves. We decided to use the SQL parser from PostgreSQL, and found the *libpg_query* project, which has extracted the SQL parser from PostgreSQL for use outside of the server environment [1]. *libpg_query* takes SQL statements as string inputs and produces abstract syntax trees, either as a JSON or Protobuf objects. We decided to work with the Protobuf representation, since it was well documented and easy to work with. The next step was to translate the Protobuf representation of SQL statements into a relational algebra. To achieve this, we parse the Protobuf AST in same order as the nodes of the relational algebra tree are expected from top to bottom. First, we parse all common table expressions (CTEs), the relational algebra trees for the CTEs are stored in a separate vector, and are not integrated in the the relational algebra tree of the main query. Second, we parse the *select* clause of the main query, which returns a *projection* node. The *projection* node acts as the root node of the subtree for the rest of the query. Third, we parse the *order by* clause if present, which returns a *sort* operator. Fourth, we parse the *having* clause if present, which returns a *having* node. Fifth, we parse the *group by* clause if present, which returns a *group by* node. Sixth, we parse the *where* clause, which returns a *selection* node. This node can also contain subtrees representing any subqueries present in the predicate. Last but not least, we parse the from clause if present, which returns a subtree containing *cross product* nodes between *relation* nodes referenced in from clause. It can also include subtrees if there were any subqueries present in the from clause. All operators returned after the root are appended to the bottom of the existing relational algebra tree, thus building it from top down.

3.1.2 Design of Relational Algebra. The relational algebra designed in this tool is a collection of operator node classes. Each operator node is derived from a base class containing the node case, pointers to child nodes, and the required number of children for the node. It also contains methods which converts the node into a string representation for printing and debugging, and a method which returns whether the node is full, based on the expected and actual number of child nodes. The node cases include relational algebra operators, such as projections, selections and joins, but also non-relational algebra nodes such as those representing expressions, predicates, case-when constructs, null tests, etc. While all node types representing relational algebra operators have a required number of child nodes of at least one, non-relational algebra nodes do not have any child nodes. Relational algebra operators which are not explicitly implemented, but are integrated into the projection operator are the rename and map operators. To cover the map operator, the projection operator stores a vector of expressions. To cover the rename operator, each expression in the projection has an option to be renamed. Since the rename and map operations can only be expressed in a select clause in SQL, there is no need for standalone versions of these operators to use outside of projections.

3.1.3 Parsing Subqueries in Selections. Subqueries which occur within predicates of selection nodes are represented as unique subquery markers in the predicate. Predicates can only contain other sub-predicates, but not entire subtrees with relational algebra nodes. The actual relational algebra tree for the subquery is attached as a subtree to the main relational algebra tree through a join operator. The subquery markers in the predicates also contain information on which type of join operator is used to join the subquery to the main tree. To associate the subquery marker in the predicate with the subquery's actual subtree, the join operator containing the subtree also stores the same subquery marker. When a subquery marker is encountered in a predicate, the corresponding subtree can be found by searching for a join node which contains the same subquery marker. This also works vice versa and is used in the optimization processes.

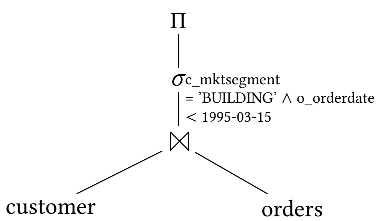
3.1.4 Parsing Correlated Subqueries. While parsing subqueries, we already check if it is a correlated query by searching for attributes which are not defined within the subtree. In contrast to uncorrelated subqueries, correlated subqueries are joined to the main tree using dependent joins, instead of regular join operators.

3.1.5 Deparsing relational algebra to SQL. The parser provided by *libpg_query* has the ability to deparse its Protobuf AST representation back to SQL, but transforming our relational algebra tree to Protobuf would have been more complex than directly transforming it to SQL ourselves. This also gives us more flexibility and would allow our tool to easily deparse into different SQL dialects in the future. We recursively iterate through the relational algebra top-down, translating each node to a string representing the node in SQL. Each projection node initializes empty strings for following SQL clauses: select, from, where, group by, having, order by. All subsequent nodes in the same query layer as the projection append to these strings, gradually building up the SQL query.

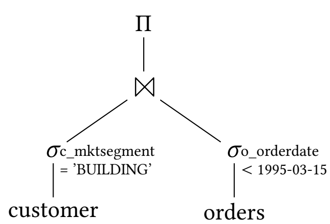
3.2 Optimizing Relational Algebra

We apply three different optimizations to the relational algebra tree: predicate pushdown, general query unnesting and unnesting *exists* and *in* subqueries. We will go into detail for each of these steps in the following sections. The relational algebra trees for CTEs are optimized separately and independently of the main relational algebra tree.

3.2.1 Predicate Pushdown. The effects of predicate pushdowns can generally not be represented in SQL. As an example, figure 2 shows two relational algebra trees which are represented by the same SQL query, even though in subfigure 2(b) all predicates have been pushed down as far as possible. This begs the question of why we include predicate pushdowns as an optimization step. Our general unnesting algorithm will introduce new projections into the relational algebra tree. Without predicate pushdowns, these new subqueries will not be able to take advantage of selections on the base tables. Figure 3 shows how the new projection Π_D introduces sideways information passing, which takes advantage of predicates pushed down the left subtree.



(a) Before predicate pushdown.



(b) After predicate pushdown.

Figure 2: Effect of predicate pushdown on relational algebra tree, both are deparsed to the same query in SQL.

Our predicate pushdown algorithm first finds all selection nodes in the tree, and splits the predicates along "and" operators. For each split predicate, we determine which relations are referenced by the attributes in the predicate. In the next step we search if there is a node deeper down in the tree, where the required relations for the predicate are already defined. If such a location is found, then the predicate is put into this location. If this node is a selection, then we add the predicate to it, otherwise we create a new selection node with the predicate and insert it into the tree at this location. If no deeper location is found where the predicate can be pushed down to, then the predicate is inserted back into its original selection node.

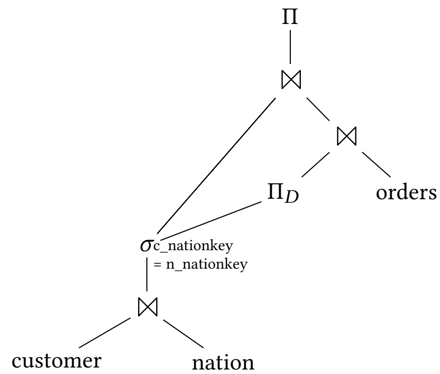


Figure 3: Sideways information passing takes advantage of predicates pushed down the left subtree.

We added the ability to push down correlated predicates (predicates containing attributes produced by an outer query). In this case, the relations which are referenced by the correlating attributes are ignored during the pushdown, and a deeper node is searched for based on only the non-correlating attributes. This can affect which rule is used when pushing the dependent join down other joins during the decorrelation phase, as both sides of a dependent join may be correlated. As a result, pushing down correlated predicates can change the structure of the join tree and resulting SQL query. In our tests, query performance did not improve with correlated predicate pushdowns, as it would often introduce additional joins which made the queries more complex.

Our predicate pushdown algorithm also contains a toggle to combine neighboring selections and cross products into join nodes. When a predicate is pushed down into a new selection node, if the selection node has a cross product as its child node, then the cross product is converted into a join node, and the predicate of the selection is inserted into the join node. The selection is then removed from the tree. This changes the deparsing of the relational algebra tree, since join nodes are transformed into explicit SQL joins with predicates, instead of cross products in the from clause and the join predicates in the where clause.

3.2.2 General Query Unnesting. Our general query unnesting algorithm is based on Neumann and Kemper's algorithm to unnest arbitrary queries [5]. While their algorithm can unnest arbitrary queries in relational algebra, we have some limitations since we are translating the unnested relational algebra tree back to SQL. Specifically, the method with which semi-joins are decorrelated would result in the same SQL query when the relational algebra tree is deparsed back to SQL. In summary, our general unnesting approach decorrelates all subqueries, which are on the right hand side of a comparison operator in a predicate. This excludes subqueries which are part of an "exists" or "in" predicate, these are handled in a separate step.

The first step is to use DFS to find all subquery markers which are linked to dependent join nodes. We replace these subquery markers with a new attribute, which will be produced by the subquery. At this point the subquery is effectively removed from the predicate. After we locate the corresponding dependent join nodes of the markers, we decorrelate the joins in reverse DFS order. From here on, we follow the algorithm as described by Neumann and Kemper.

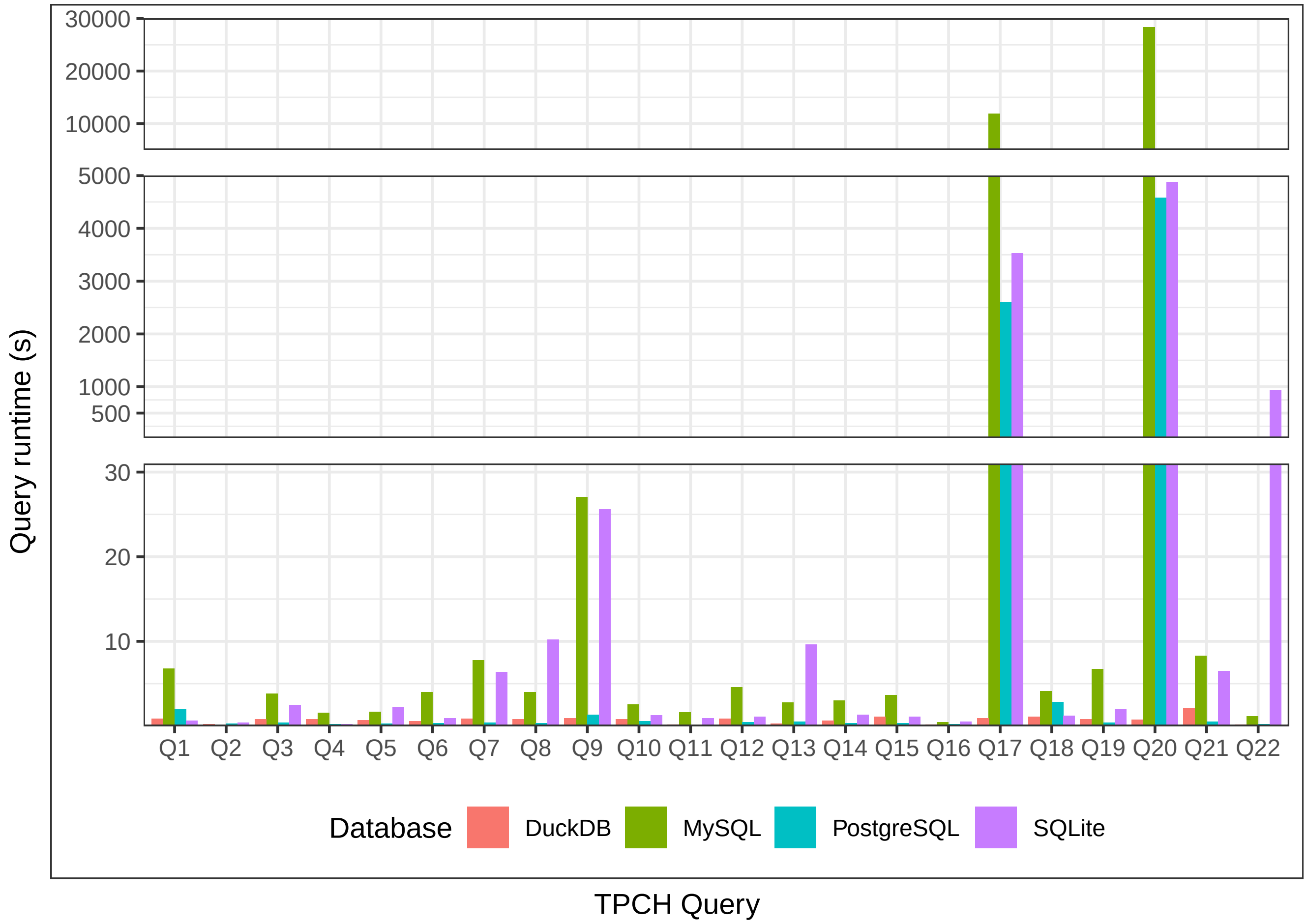


Figure 4: Runtimes of original TPC-H queries (SF 1) without decorrelating subqueries.

First, we transform the dependent join into a regular join, and introduce a new dependent join as its right child. Second, we comput the domain "D" by checking for correlated attributes in the subtree, and set a projection on D as the left child of the dependent join Third, we push down dependent join down until its right subtree is no longer dependent on the left. When it is, we convert it into a cross product. At this point we determine if decoupling of our side-ways information passing is possible, by checking whether the (formerly) correlated predicates all use equi-predicates. Now we can optionally decouple by removing the cross product and renaming all "D" attributes in the subtree to their equivalent attribute in the subtree. At this point, "a=a" predicates will exist in the subtree, which are removed from the tree in a separate step. If decoupling is not possible or not wished, we use CTEs to model the sideways information passing. Do do this by moving the shared left subtree into a CTE, and point both the "D" projection and the original dependent join from step 1 to the CTE instead of the subtree. After moving the subtree into the CTE, we traverse the subtree of the (formerly) correlated subquery and rename all relevant attributes to select the "CTE" attributes. In this way, we mimic in information sideways passing as presented by Neumann and Kemper.

3.2.3 Unnesting "exists" and "in" subqueries. We decorrelate "exists" and "in" subqueries by first searching for any subquery markers using DFS, which have a join type for dependent exists or dependent in subqueries. Next, we use the subquery markers to locate the subtree for the dependent subqueries. The correlated subqueries are decorrelated in reverse DFS order, so that most nested subqueries are decorrelated first. During decorrelation, we first identify all correlated predicates in the subquery and move the subquery into a CTE. We then remove the correlating predicates from the CTE. We originally added a group by operator, grouping on the attributes used in the correlated predicates, as in the general unnesting algorithm. However, our tests showed that the group by operator would add significant run time to query executions. Since the group by operator is not necessary to produce the correct result for exists and in subqueries, we decided to remove the group by operator. In the original place of the subquery, we create a new subquery which selects from the CTE, with a selection node containing the correlating predicates.

Our tool also has the ability to transform trivially correlated exists subqueries into uncorrelated in subqueries, as implemented by mariadb [2]. It can also decorrelate complex correlated exists subqueries by left outer joining the subquery, and adding predicates to check whether the left join resulted in null values. Since our tests didn't show improved performance for these variants of queries, we disabled them and use our simple joint approach for decorrelating "exists" and "in" subqueries.

4 EVALUATION

We evaluated performance impact of our tool using the TPC-H benchmark. We used the SF1 dataset and ran the TPC-H queries in their original form and also on their decorrelated versions produced by our tool. Six of TPC-H's 22 queries contain correlated subqueries, which we will focus on. We have chosen five relational open-source databases to benchmark on: PostgreSQL, SQLite, MySQL, Umbra and DuckDB. To use our tool with SQLite, we needed to manually alter the syntax of some generated SQL queries to fit the SQLite syntax, e.g. for date intervals. DuckDB and Umbra have already integrated Neumann and Kemper's query unnesting algorithm, so we do not expect our optimized queries to have significant impact, but we will be use DuckDB as a point of reference for other database systems. Our benchmarks were executed on a server with a 12th Gen Intel(R) Core(TM) i7-12700H CPU and 12GB memory. All databses were used with their default configurations. Queries ran three times to warm up the cache, then run 10 times for the benchmark. We took the median runtime of the 10 executions to evaluate the performance of a query. For PostgreSQL, MySQL and SQLite, the runtimes of the unoptimized Q17 and Q20 queries ranged between 45 minutes and 8 hours, in these cases we only ran the query once due to time constraints and not needing such precise accuracy when measuring runtime improvements from hours to seconds. The following sections discuss the runtimes of the original TPC-H queries, then the runtimes of decorrelated TPC-H queries which have correlated subqueries in *exists* and *in* clauses, and finally the runtimes of decorrelated TPC-H queries which have correlated subqueries in regular predicates. In each section, discuss the results for each database.

4.1 Original TPC-H Queries

We ran all TPC-H queries on all databases and summarized the resulting runtimes in Figure 4. Most notably, PostgreSQL, SQLite and MySQL, Q17 and Q20 have extremely long execution times due to the correlated subqueries. As expected, Umbra and DuckDB execute these queries fast, due to their inbuilt decorrelation algorithm. Q17 and Q20 are the most interesting to us, as our tests suggest that these queries represent types of correlated subqueries which cannot be decorrelated by widely used database systems. Even though the queries contain correlated subqueries, Q2, Q4, Q21 and Q22 had short execution times in almost all databases. We will evaluate whether the decorrelated versions of these queries perform any differently. Additionally, for Q2, Q17 and Q20, our tool can generate two different decorrelated versions of these queries, a *decoupled* query without sideways information passing, and a *non-decoupled* version with sideways information passing. We benchmarked both versions to compare any performance differences. The following subsections will into the runtime differences between the original and decorrelated queries for each of our database systems. Since our tool does not alter queries which do not contain correlated subqueries in any meaningful way, these will not be discussed.

4.2 Decorrelated TPC-H Queries: correlated *exists* and *in* subqueries

The queries in the TPC-H benchmark contain three queries which have correlated subqueries in *exists* and *in* predicates: Q4, Q21 and Q22. Figure 5 shows the runtime improvement factor of the decorrelated queries over the original queries. In DuckDB and SQLite and PostgreSQL, the decorrelated form of these queries had the same runtimes as the original queries, within margin of error. In MySQL, Q22 was 24% faster in its decorrelated form. SQLite had long runtimes of 15min for Q22, unfortunately our decorrelated query could not improve this. In summary, there were no significant differences in execution times between the original form and decorrelated form of these queries.

4.3 Decorrelated TPC-H Queries: correlated subqueries in regular predicates

The queries in the TPC-H benchmark contain three queries which have correlated subqueries in regular predicates: Q2, Q17 and Q20. Figure 6 shows the runtime improvement factor of the decorrelated queries with decoupled sideways information passing over the original query. Figure 7 shows the runtime improvement factor of the decorrelated queries without decoupled sideways information passing over the original query.

4.3.1 DuckDB. As expected, our decorrelated SQL queries barely affected the execution times of the queries in DuckDB, since it has already implemented a decorrelation algorithm based on Neumann and Kemper's in their optimizer.

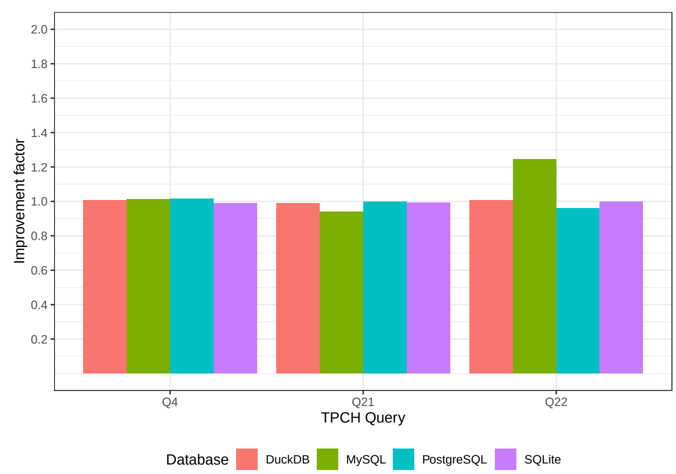


Figure 5: Runtime improvement factor of decorrelated *exists* and *in* queries compared to original queries.

4.3.2 PostgreSQL. For Q2, while the decorrelated query with decoupling improves the execution time only slightly, the version without decoupling is 10x faster than the original query. The original Q17 and Q20 had extremely long execution times of 44min and 76min respectively. For both of these queries, the optimized queries reduced the execution times to under 3s, with significant improvement factors of 1500 and 3000 respectively. For Q20 the query with decoupling is faster, while in Q17 the query without decoupling is faster.

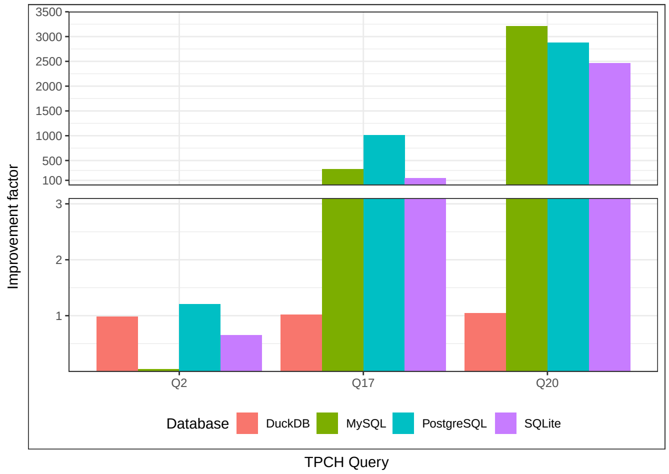


Figure 6: Runtime improvement factor of decorrelated queries with decoupling of sideways information passing, compared to original queries.

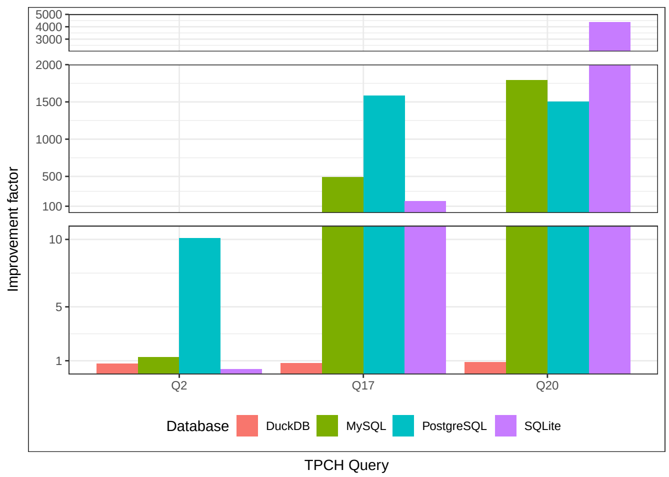


Figure 7: Runtime improvement factor of decorrelated queries without decoupling of sideways information passing, compared to original queries.

4.3.3 SQLite. For Q2, the decorrelated queries were up to 2.8 times slower than the original queries, but still maintained quick runtimes of around 1s. For both Q17 and Q20, the decorrelated query performed better without decoupling. The runtime of Q17 dropped from 58min to 21s, and Q20 dropped from 81min to 1s.

4.3.4 MySQL. The runtimes of the original Q17 and Q20 queries were significantly higher at 3h 18min and 7h 52min respectively. For Q17, the runtimes were dropped to 36s with decoupling and 24s without decoupling. For Q20 the runtimes were dropped to 9s with decoupling and 16s without decoupling. As was the case with PostgreSQL, decoupling performed better for Q20, and no decoupling performed better for Q17. One interesting difference for Q2 is that the decorrelated query with decoupling performs around 20 times worse than the original query, while the version without decoupling query has a similar runtime to the original.

5 CONCLUSION

Our results show the runtime improvements of decorrelated queries vary across database systems. For Q4, Q21 and Q22, query decorrelation on SQL-level did not change query runtime much. This suggests that most database systems' internal optimizers are already well equipped to optimize correlated subqueries in these cases. For TPC-H Q17 and Q20, our tool reduced query runtimes up by 1500 and 4000 times respectively, reducing hour long queries runtimes to only seconds. PostgreSQL, SQLite and MySQL all saw such significant runtime improvements, while as expected, DuckDB did not. The correlated queries which could already be executed quickly by databases still did so with the decorrealted versions, without siginificant speedups or slowdowns. Queries which did not have any correlations were not altered in any significant way by our tool, so runtimes also did not change significantly.

Our results show that many popular database systems do not have the ability to decorrelate complex correlated subqueries, and in these cases our standalone tool provides an easy-to-use and valuable additional optimization layer to drastically improve query runtimes.

The modularity of our tool's design makes it easily adaptable for different SQL dialects. With its current support of PostgreSQL's popular dialect, it already has the potential to impact a wide range of users.

While our decorrelation algorithms are based on Neumann and Kemper's general unnesting algorithm, it was not possible to implement it entirely on SQL-level. Since SQL represents semi-joins using less expressive exists-operators, the simple method of decorrelating semi-joins is not applicable in SQL. Our results suggest that most database systems have little trouble decorrelating subqueries inside semi-joins, which is expected due to how simple it is decorrelate these in relational algebra. For the sake of completeness, we implemented an algorithm which decorrelates semi-joins, even though the resulting queries do not show useful performance benefits.

We plan to open source our tool and look forward to extending its SQL parsing and deparsing capabilities beyond the TPC-H benchmark. Adding the ability to pass table schemas to our tool will also increase the variety of queries it can optimize. Other query optimizations on SQL-level are also possible to integrate into our tool.

REFERENCES

[1] 2023. Libpg_Query. https://github.com/pganalyze/libpg_query. (2023). Accessed: 2023-03-09.
[2] 2023. MariaDB EXISTS-to-IN Optimization. https://mariadb.com/kb/en/exists-to-in-optimization/. (2023). Accessed: 2023-03-09.
[3] 2023. MySQL Correlated Subqueries. https://dev.mysql.com/doc/refman/8.0/en/correlated-subqueries.html. (2023). Accessed: 2023-03-09.
[4] 2023. SQLite Subquery Flattening. https://www.sqlite.org/optoverview.html#flattening. (2023). Accessed: 2023-03-09.
[5] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

A APPENDIX

Figure 8 shows the improvement factors of decorrelated TPC-H queries over the original queries, with different values for PostgreSQL's
max_parallel_workers_per_gather setting. This sets the maximum number of workers that can be started by a single Gather or Gather

Merge node, the default value is 2, while a value of 0 represents single threaded execution. The decorrelated query runtimes used for this Figure, were chosen as the faster between the decoupled and not decoupled version for each query. The Figure shows that we can achieve high improvement factors already with single threaded execution, while higher parallelization increases the improvement factor significantly for Q20 to almost 5000.

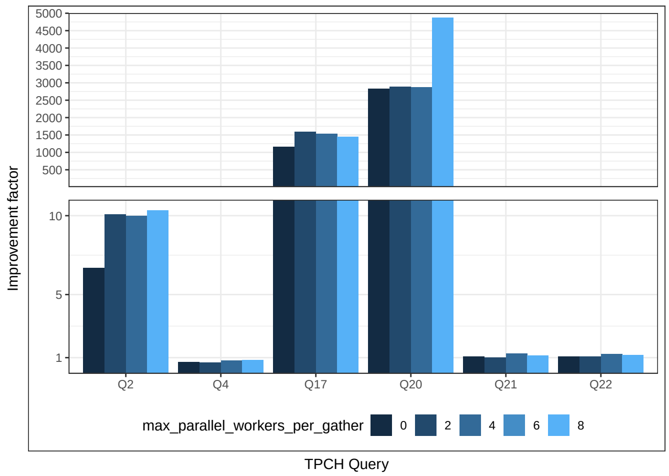


Figure 8: Runtime improvement factor of decorrelated compared to original queries in PostgreSQL for different configurations for parallel query execution.

A.1 Original TPC-H queries

```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as
     sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 +
     l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '90'
     day
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

Listing 1: Original TPC-H 1

```
select
        s_acctbal,
        s_name,
        n_name,
        p_partkey,
        p_mfgr,
        s_address,
        s_phone,
        s_comment
from
        part,
        supplier,
        partsupp,
        nation,
        region
where
        p_partkey = ps_partkey
        and s_suppkey = ps_suppkey
        and p_size = 15
        and p_type like '%BRASS'
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'EUROPE'
        and ps_supplycost = (
                select
                        min(ps_supplycost)
                from
                        partsupp,
                        supplier,
                        nation,
                        region
                where
                        p_partkey = ps_partkey
                        and s_suppkey = ps_suppkey
                        and s_nationkey = n_nationkey
                        and n_regionkey = r_regionkey
                        and r_name = 'EUROPE'
        )
order by
        s_acctbal desc,
        n_name,
        s_name,
        p_partkey;
```

Listing 2: Original TPC-H 2

```
select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as
     revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-15'
        and l_shipdate > date '1995-03-15'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate;
```

Listing 3: Original TPC-H 3

```
select
        o_orderpriority,
        count(*) as order_count
from
```

```
        orders
where
        o_orderdate >= date '1993-07-01'
        and o_orderdate < date '1993-07-01' + interval '3' month
        and exists (
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority;
```

Listing 4: Original TPC-H 4

```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'
        and o_orderdate >= date '1994-01-01'
        and o_orderdate < date '1994-01-01' + interval '1' year
group by
        n_name
order by
        revenue desc;
```

Listing 5: Original TPC-H 5

```
select
        sum(l_extendedprice * l_discount) as revenue
from
        lineitem
where
        l_shipdate >= date '1994-01-01'
        and l_shipdate < date '1994-01-01' + interval '1' year
        and l_discount between 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24;
```

Listing 6: Original TPC-H 6

```
select
        supp_nation,
        cust_nation,
        l_year,
        sum(volume) as revenue
from
        (
                select
                        n1.n_name as supp_nation,
                        n2.n_name as cust_nation,
                        extract(year from l_shipdate) as l_year,
                        l_extendedprice * (1 - l_discount) as volume
                from
                        supplier,
                        lineitem,
                        orders,
                        customer,
                        nation n1,
                        nation n2
                where
                        s_suppkey = l_suppkey
                        and o_orderkey = l_orderkey
                        and c_custkey = o_custkey
                        and s_nationkey = n1.n_nationkey
                        and c_nationkey = n2.n_nationkey
                        and (
                                (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
                                or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
                        )
                        and l_shipdate between date '1995-01-01' and date '1996-12-31'
        ) as shipping
group by
        supp_nation,
        cust_nation,
        l_year
order by
        supp_nation,
        cust_nation,
        l_year;
```

Listing 7: Original TPC-H 7

```
select
        o_year,
        sum(case
                when nation = 'BRAZIL' then volume
                else 0
        end) / sum(volume) as mkt_share
from
        (
                select
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) as volume,
                        n2.n_name as nation
                from
                        part,
                        supplier,
                        lineitem,
                        orders,
                        customer,
                        nation n1,
                        nation n2,
                        region
                where
                        p_partkey = l_partkey
                        and s_suppkey = l_suppkey
                        and l_orderkey = o_orderkey
                        and o_custkey = c_custkey
                        and c_nationkey = n1.n_nationkey
```

```
                                and n1.n_regionkey = r_regionkey
                                and r_name = 'AMERICA'
                                and s_nationkey = n2.n_nationkey
                                and o_orderdate between date '1995-01-01' and date '1996-12-31'
                                and p_type = 'ECONOMY ANODIZED STEEL'
        ) as all_nations
group by
        o_year
order by
        o_year;
```

Listing 8: Original TPC-H 8

```
select
        nation,
        o_year,
        sum(amount) as sum_profit
from
        (
                select
                        n_name as nation,
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
                from
                        part,
                        supplier,
                        lineitem,
                        partsupp,
                        orders,
                        nation
                where
                        s_suppkey = l_suppkey
                        and ps_suppkey = l_suppkey
                        and ps_partkey = l_partkey
                        and p_partkey = l_partkey
                        and o_orderkey = l_orderkey
                        and s_nationkey = n_nationkey
                        and p_name like '%green%'
        ) as profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc;
```

Listing 9: Original TPC-H 9

```
select
        c_custkey,
        c_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        c_acctbal,
        n_name,
        c_address,
        c_phone,
        c_comment
from
        customer,
        orders,
        lineitem,
        nation
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate >= date '1993-10-01'
        and o_orderdate < date '1993-10-01' + interval '3' month
        and l_returnflag = 'R'
        and c_nationkey = n_nationkey
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
        c_comment
order by
        revenue desc;
```

Listing 10: Original TPC-H 10

```
select
        ps_partkey,
        sum(ps_supplycost * ps_availqty) as value
from
        partsupp,
        supplier,
        nation
where
        ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = 'GERMANY'
group by
        ps_partkey having
                sum(ps_supplycost * ps_availqty) > (
                        select
                                sum(ps_supplycost * ps_availqty) * 0.0001
                        from
                                partsupp,
                                supplier,
                                nation
                        where
                                ps_suppkey = s_suppkey
                                and s_nationkey = n_nationkey
                                and n_name = 'GERMANY'
                )
order by
        value desc;
```

Listing 11: Original TPC-H 11

```
select
        l_shipmode,
        sum(case
                when o_orderpriority = '1-URGENT'
                        or o_orderpriority = '2-HIGH'
                        then 1
                else 0
        end) as high_line_count,
        sum(case
                when o_orderpriority <> '1-URGENT'
                        and o_orderpriority <> '2-HIGH'
                        then 1
                else 0
        end) as low_line_count
from
        orders,
        lineitem
where
        o_orderkey = l_orderkey
        and l_shipmode in ('MAIL', 'SHIP')
```

```sql
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate >= date '1994-01-01'
        and l_receiptdate < date '1994-01-01' + interval '1' year
group by
        l_shipmode
order by
        l_shipmode;
```

Listing 12: Original TPC-H 12

```sql
select
        c_count,
        count(*) as custdist
from
        (
                select
                        c_custkey,
                        count(o_orderkey)
                from
                        customer left outer join orders on
                                c_custkey = o_custkey
                                and o_comment not like '%special%requests%'
                group by
                        c_custkey
        ) as c_orders (c_custkey, c_count)
group by
        c_count
order by
        custdist desc,
        c_count desc;
```

Listing 13: Original TPC-H 13

```sql
select
        100.00 * sum(case
                when p_type like 'PROMO%'
                        then l_extendedprice * (1 - l_discount)
                else 0
        end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
        lineitem,
        part
where
        l_partkey = p_partkey
        and l_shipdate >= date '1995-09-01'
        and l_shipdate < date '1995-09-01' + interval '1' month;
```

Listing 14: Original TPC-H 14

```sql
with revenue (supplier_no, total_revenue) as (
        select
                l_suppkey,
                sum(l_extendedprice * (1 - l_discount))
        from
                lineitem
        where
                l_shipdate >= date '1996-01-01'
                and l_shipdate < date '1996-01-01' + interval '3' month
        group by
                l_suppkey)
select
        s_suppkey,
        s_name,
        s_address,
        s_phone,
        total_revenue
from
        supplier,
        revenue
where
        s_suppkey = supplier_no
        and total_revenue = (
                select
                        max(total_revenue)
                from
                        revenue
        )
order by
        s_suppkey;
```

Listing 15: Original TPC-H 15

```sql
select
        p_brand,
        p_type,
        p_size,
        count(distinct ps_suppkey) as supplier_cnt
from
        partsupp,
        part
where
        p_partkey = ps_partkey
        and p_brand <> 'Brand#45'
        and p_type not like 'MEDIUM POLISHED%'
        and p_size in (49, 14, 23, 45, 19, 3, 36, 9)
        and ps_suppkey not in (
                select
                        s_suppkey
                from
                        supplier
                where
                        s_comment like '%Customer%Complaints%'
        )
group by
        p_brand,
        p_type,
        p_size
order by
        supplier_cnt desc,
        p_brand,
        p_type,
        p_size;
```

Listing 16: Original TPC-H 16

```sql
select
        sum(l_extendedprice) / 7.0 as avg_yearly
from
        lineitem,
        part
where
        p_partkey = l_partkey
        and p_brand = 'Brand#23'
        and p_container = 'MED BOX'
        and l_quantity < (
                select
                        0.2 * avg(l_quantity)
                from
                        lineitem
                where
                        l_partkey = p_partkey
```

```
        );
```

Listing 17: Original TPC-H 17

```
select
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice,
        sum(l_quantity)
from
        customer,
        orders,
        lineitem
where
        o_orderkey in (
                select
                        l_orderkey
                from
                        lineitem
                group by
                        l_orderkey having
                                sum(l_quantity) > 300
        )
        and c_custkey = o_custkey
        and o_orderkey = l_orderkey
group by
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice
order by
        o_totalprice desc,
        o_orderdate;
```

Listing 18: Original TPC-H 18

```
select
        sum(l_extendedprice* (1 - l_discount)) as revenue
from
        lineitem,
        part
where
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#12'
                and p_container in ('SM CASE', 'SM BOX',
     'SM PACK', 'SM PKG')
                and l_quantity >= 1 and l_quantity <= 1 +
       10
                and p_size between 1 and 5
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#23'
                and p_container in ('MED BAG', 'MED BOX',
      'MED PKG', 'MED PACK')
                and l_quantity >= 10 and l_quantity <= 10
       + 10
                and p_size between 1 and 10
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = 'Brand#34'
                and p_container in ('LG CASE', 'LG BOX',
     'LG PACK', 'LG PKG')
                and l_quantity >= 20 and l_quantity <= 20
      + 10
                and p_size between 1 and 15
                and l_shipmode in ('AIR', 'AIR REG')
                and l_shipinstruct = 'DELIVER IN PERSON'
        );
```

Listing 19: Original TPC-H 19

```
select
        s_name,
        s_address
from
        supplier,
        nation
where
        s_suppkey in (
                select
                        ps_suppkey
                from
                        partsupp
                where
                        ps_partkey in (
                                select
                                        p_partkey
                                from
                                        part
                                where
                                        p_name like '
     forest%'
                        )
                        and ps_availqty > (
                                select
                                        0.5 * sum(
     l_quantity)
                                from
                                        lineitem
                                where
                                        l_partkey =
     ps_partkey
                                        and l_suppkey =
     ps_suppkey
                                        and l_shipdate >=
      date '1994-01-01'
                                        and l_shipdate <
     date '1994-01-01' + interval '1' year
                        )
        )
        and s_nationkey = n_nationkey
        and n_name = 'CANADA'
order by
        s_name;
```

Listing 20: Original TPC-H 20

```
select
        s_name,
        count(*) as numwait
from
        supplier,
        lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
        and o_orderkey = l1.l_orderkey
```

```
        and o_orderstatus = 'F'
        and l1.l_receiptdate > l1.l_commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.l_orderkey = l1.l_orderkey
                        and l2.l_suppkey <> l1.l_suppkey
        )
        and not exists (
                select
                        *
                from
                        lineitem l3
                where
                        l3.l_orderkey = l1.l_orderkey
                        and l3.l_suppkey <> l1.l_suppkey
                        and l3.l_receiptdate > l3.l_commitdate
        )
        and s_nationkey = n_nationkey
        and n_name = 'SAUDI ARABIA'
group by
        s_name
order by
        numwait desc,
        s_name;
```

Listing 21: Original TPC-H 21

```
select
        cntrycode,
        count(*) as numcust,
        sum(c_acctbal) as totacctbal
from
        (
                select
                        substring(c_phone from 1 for 2) as cntrycode,
                        c_acctbal
                from
                        customer
                where
                        substring(c_phone from 1 for 2) in
                                ('13', '31', '23', '29', '30', '18', '17')
                        and c_acctbal > (
                                select
                                        avg(c_acctbal)
                                from
                                        customer
                                where
                                        c_acctbal > 0.00
                                        and substring(c_phone from 1 for 2) in
                                                ('13', '31', '23', '29', '30', '18', '17')
                        )
                        and not exists (
                                select
                                        *
                                from
                                        orders
                                where
                                        o_custkey = c_custkey
                        )
        ) as custsale
group by
        cntrycode
order by
        cntrycode;
```

Listing 22: Original TPC-H 22

A.2 Decorrelated TPC-H queries with decoupled sideways information passing

```
select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  region,
  nation,
  partsupp,
  supplier,
  part,
  (select
    min(ps_supplycost),
    ps_partkey
  from
    region,
    nation,
    supplier,
    partsupp
  where
    s_suppkey=ps_suppkey
    and s_nationkey=n_nationkey
    and n_regionkey=r_regionkey
    and r_name='EUROPE'
  group by ps_partkey
  ) as t1(m,t1_p_partkey)
where
  ps_supplycost=t1.m
  and p_partkey=t1.t1_p_partkey
  and p_partkey=ps_partkey
  and (s_suppkey=ps_suppkey
  and s_nationkey=n_nationkey)
  and n_regionkey=r_regionkey
  and r_name='EUROPE'
  and (p_size=15 and p_type like '%BRASS')
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey;
```

Listing 23: Decorrelated TPC-H 2 with decoupled sideways information passing

```
with cte_1(l_orderkey) as (
select l_orderkey
from lineitem
where l_commitdate<l_receiptdate
)
select o_orderpriority, count(*) as order_count
from orders
where exists (select *
from cte_1
where cte_1.l_orderkey=o_orderkey
```

```
) and (o_orderdate>=date '1993-07-01' and o_orderdate<(date '1993-07-01'+interval '3' month))
group by o_orderpriority
order by o_orderpriority;
```

Listing 24: Decorrelated TPC-H 4 with decoupled sideways information passing

```
select
  (sum(l_extendedprice)/7.0) as avg_yearly
from
  part,
  lineitem,
  (select (0.2*avg(l_quantity)),
    l_partkey
  from
    lineitem
  group by
    l_partkey
  ) as t1(m,t1_p_partkey)
where
  l_quantity<t1.m
  and p_partkey=t1.t1_p_partkey
  and p_partkey=l_partkey
  and (p_brand='Brand#23' and p_container='MED BOX');
```

Listing 25: Decorrelated TPC-H 17 with decoupled sideways information passing

```
select
  s_name,
  s_address
from
  nation,
  supplier
where
  s_suppkey in
  (select ps_suppkey
  from partsupp,
    (select
      (0.5*sum(l_quantity)),
      l_partkey,
      l_suppkey
    from
      lineitem
    where
      l_shipdate>=date '1994-01-01'
      and l_shipdate<(date '1994-01-01'+interval '1' year)
    group by
      l_partkey,
      l_suppkey
    ) as t3(m,t3_ps_partkey,t3_ps_suppkey)
  where ps_partkey in
    (select p_partkey
    from part
    where p_name like 'forest%'
    )
    and (ps_availqty>t3.m
    and ps_partkey=t3.t3_ps_partkey
    and ps_suppkey=t3.t3_ps_suppkey)
    ) and s_nationkey=n_nationkey
    and n_name='CANADA'
order by s_name;
```

Listing 26: Decorrelated TPC-H 20 with decoupled sideways information passing

```
with cte_2(l_orderkey,l_suppkey) as (
select l3.l_orderkey, l3.l_suppkey
from lineitem l3
where l3.l_receiptdate>l3.l_commitdate
),cte_3(l_orderkey,l_suppkey) as (
select l2.l_orderkey, l2.l_suppkey
from lineitem l2
)
select s_name, count(*) as numwait
from nation, orders, lineitem l1, supplier
where exists (select *
from cte_3
where cte_3.l_orderkey=l1.l_orderkey and cte_3.l_suppkey<>l1.l_suppkey
) and (not exists (select *
from cte_2
where cte_2.l_orderkey=l1.l_orderkey and cte_2.l_suppkey<>l1.l_suppkey
)) and (s_suppkey=l1.l_suppkey and s_nationkey=n_nationkey) and o_orderkey=l1.l_orderkey and n_name='SAUDI ARABIA' and o_orderstatus='F' and l1.l_receiptdate>l1.l_commitdate
group by s_name
order by numwait desc, s_name;
```

Listing 27: Decorrelated TPC-H 21 with decoupled sideways information passing

```
with cte_2(o_custkey) as (
select o_custkey
from orders
)
select cntrycode, count(*) as numcust, sum(c_acctbal) as totacctbal
from (select substring(c_phone from 1 for 2) as cntrycode, c_acctbal
from customer
where c_acctbal>(select avg(c_acctbal)
from customer
where c_acctbal>0.00 and substring(c_phone from 1 for 2) in ('13','31','23','29','30','18','17')
) and (not exists (select *
from cte_2
where cte_2.o_custkey=c_custkey
)) and substring(c_phone from 1 for 2) in ('13','31','23','29','30','18','17')
) as custsale
group by cntrycode
order by cntrycode;
```

Listing 28: Decorrelated TPC-H 22 with decoupled sideways information passing

A.3 Decorrelated TPC-H Queries with sideways information passing not decoupled

```
with cte_2 as (
  select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment,
    ps_supplycost,
    ps_partkey,
    s_suppkey,
    ps_suppkey,
    s_nationkey,
    n_nationkey,
```

```
    n_regionkey,
    r_regionkey,
    r_name
  from
    region,
    nation,
    partsupp,
    supplier,
    part
  where
    p_partkey=ps_partkey
    and (s_suppkey=ps_suppkey
    and s_nationkey=n_nationkey)
    and n_regionkey=r_regionkey
    and r_name='EUROPE'
    and (p_size=15 and p_type like '%BRASS')
)
select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  cte_2,  (select min(ps_supplycost),
      d.p_partkey
    from  (select p_partkey
      from cte_2
      ) as d(p_partkey),
      region,
      nation,
      supplier,
      partsupp
    where
      d.p_partkey=ps_partkey
      and s_suppkey=ps_suppkey
      and s_nationkey=n_nationkey
      and n_regionkey=r_regionkey
      and r_name='EUROPE'
    group by
      d.p_partkey
    ) as t1(m,t1_p_partkey)
where
  ps_supplycost=t1.m
  and p_partkey=t1.t1_p_partkey
order by
  s_acctbal desc,
  n_name,
  s_name,
  p_partkey;
```

Listing 29: Decorrelated TPC-H 2 with sideways information passing not decoupled

```
with cte_2 as (
  select
    l_extendedprice,
    l_quantity,
    p_partkey,
    l_partkey
  from
    part,
    lineitem
  where
    p_partkey=l_partkey
    and (p_brand='Brand#23' and p_container='MED BOX')
)
select
  (sum(l_extendedprice)/7.0) as avg_yearly
from
  cte_2,
  (select
    (0.2*avg(l_quantity)),
    d.p_partkey
  from  (select
      p_partkey
    from cte_2
    ) as d(p_partkey), lineitem
  where l_partkey=d.p_partkey
  group by d.p_partkey
  ) as t1(m,t1_p_partkey)
where
  l_quantity<t1.m
  and p_partkey=t1.t1_p_partkey;
```

Listing 30: Decorrelated TPC-H 17 with sideways information passing not decoupled

```
with cte_4 as (
  select ps_suppkey, ps_partkey, ps_availqty
  from partsupp
)
select s_name, s_address
from nation, supplier
where
  s_suppkey in
  (select
    ps_suppkey
  from
    cte_4,
    (select
      (0.5*sum(l_quantity)),
      d.ps_partkey,
      d.ps_suppkey
    from
      (select
        ps_partkey,
        ps_suppkey
      from cte_4
      ) as d(ps_partkey,ps_suppkey),
      lineitem
    where
      l_partkey=d.ps_partkey
      and l_suppkey=d.ps_suppkey
      and (l_shipdate>=date '1994-01-01'
      and l_shipdate<(date '1994-01-01'+interval '1' year
   ))
    group by
      d.ps_partkey,
      d.ps_suppkey
    ) as t3(m,t3_ps_partkey,t3_ps_suppkey)
  where ps_partkey in
    (select p_partkey
    from part
    where p_name like 'forest%'
    )
    and (ps_availqty>t3.m
    and ps_partkey=t3.t3_ps_partkey
    and ps_suppkey=t3.t3_ps_suppkey)
    ) and s_nationkey=n_nationkey
    and n_name='CANADA'
order by s_name;
```

Listing 31: Decorrelated TPC-H 20 with sideways information passing not decoupled