Improving Unnesting of Complex Queries

Thomas Neumann[1]

Abstract: SQL allows for very flexible nesting of queries, including subqueries that access attributes by the outer part of the query. These correlated subqueries simplify query formulation, but their execution is very inefficient, leading to $O(n^2)$ runtime complexity. Which can become prohibitively expensive for large databases.

Query optimizers therefore try to unnest, i.e., decorrelate, dependent queries. Existing decorrelation techniques, however, are either limited in scope or lead to suboptimal execution plans when correlated queries are stacked repeatedly inside each other. In this work, we present a generalized unnesting approach that can handle deep nestings of correlated subqueries and that generalizes to complex query constructs, including recursive SQL. This generalized unnesting improves the asymptotic complexity, and thus can lead to dramatic performance improvements in the affected queries.

Keywords: query optimization, query unnesting

1 Introduction

One of the key properties of SQL is that it is a *declarative* query language, where the user specifies only the query intent and the database system then finds the best execution strategy for this query. At least, that is the promise of it being declarative. In reality, this declarative nature sometimes breaks down when formulating complex queries, in particular when the queries contain complex subqueries.

```
SELECT *
FROM customer
WHERE c_mktsegment='AUTOMOBILE' AND
  (SELECT COUNT(*)
   FROM orders
   WHERE o_custkey=c_custkey AND
     (SELECT SUM(l_extendedprice)
      FROM lineitem
      WHERE l_orderkey=o_orderkey)>300000)>5
```

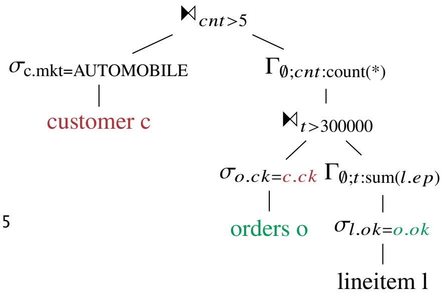

$\ltimes_{cnt>5}$
$\sigma_{\text{c.mkt=AUTOMOBILE}}$
customer c
$\Gamma_{\emptyset;cnt:\text{count(*)}}$
$\ltimes_{t>300000}$
$\sigma_{o.ck=c.ck}$
orders o
$\Gamma_{\emptyset;t:\text{sum}(l.ep)}$
$\sigma_{l.ok=o.ok}$
lineitem l

Fig. 1: Example Query with Nested Correlated Subqueries, the Correlations are Color Coded

Consider the example query in Figure 1. While conceptually simple – it finds customers from the automobile market that have more than 5 orders with a revenue of more than 300K – it is very unpleasant for most database systems. The canonical translation of this query

[1] Technische Universität München, Institut für Informatik, Boltzmannstr. 3, 85748 Garching, Germany, neumann@in.tum.de, https://orcid.org/0000-0001-5787-142X

into relational algebra shown on the right uses a *dependent join* ($\ltimes\!\!\!\rtimes$), where the right-hand side of the join is evaluated for every tuple of the left-hand side. This nested loop evaluation is necessary because the right-hand side accesses values provided from the left-hand side, which would not be possible if we were to use, e.g., a regular hash join. But algorithmically a dependent join is a very bad idea, the runtime complexity of a join $L \ltimes\!\!\!\rtimes R$ is in $\Theta(|L||R|)$, which grows quadratically with the input sizes. And for database engines execution times in $O(n^2)$ are often prohibitive, as our n tends to be in the order of millions. On TPC-H scale factor 1, this query needs over 25 minutes on PostgreSQL 16, even though it processes just 1 GB of data. Note that it is possible to reformulate the query in a way such that PostgreSQL can answer it instantly, by decoupling the subqueries, but that makes the query longer and more difficult to write. And given that SQL aims to be declarative, that transformation should be done by the DBMS automatically.

Historically, database systems tried to cope with correlated subqueries by recognizing common patterns in SQL queries and rewriting them to eliminate correlations [Ki82, JK84, Ki85]. While this allows for an efficient execution of some correlated subquery patterns, this is not a very satisfying approach, as even small changes to the SQL text can break these patterns, again resulting in disastrous execution times. A fundamentally different approach was proposed in our previous work ten years ago [NK15]: It operates on the algebra level, making it robust against SQL variations, and it is based upon two observations: 1) we can always rewrite a dependent join into a dependent join where the right-hand side is joined with the unique correlation values from the left-hand side, and 2) a dependent join where the left-hand side is duplicate free can be pushed down an algebra tree until the right-hand side no longer depends on the left-hand side. We will discuss the approach in more detail in Section 2.

Conceptually, this algorithm allows for eliminating correlations in arbitrary queries, dramatically improving the performance of these queries and making it possible to execute them even for large inputs. It was such a significant improvement that several database systems now use it for unnesting queries. Both Hyper and DuckDB use it in commercial offerings, our research system Umbra uses it, and we are aware of several other systems that have prototype implementations. This more widespread usage exposed some limitations of the original publication [NK15]: First, there are some advanced SQL constructs that are not covered by the algorithm, particularly recursive SQL and complex order by and group by constructs. And second, the algorithm sometimes exhibits pathological behavior when dependent queries are nested inside each other, similar to what happens in our query in Figure 1: The unnesting algorithm has to be executed twice – once for each dependent join. The original algorithm does that bottom-up. But because they are stacked inside one another, the second unnesting pass increases the costs of the first join, leading to a performance degradation that becomes worse the deeper the nesting is. See Section 2.3 for a detailed example.

In this paper, we address both shortcomings. First, we change the algorithm to use a top-down strategy that takes subsequent dependent joins into account, leading to dramatically better performance in the case of deep nestings. And second, we cover the remaining SQL constructs, in particular recursive SQL and ORDER BY LIMIT constructs.

The rest of this paper is structured as follows: We give an overview of the problem statement and discuss the original algorithm from [NK15] in Section 2. We then develop a new algorithm that improves the previous approach in Section 3. Complex SQL constructs are handled in Section 4. An evaluation is included in Section 5. Related work is discussed in Section 6 and the results are summarized in Section 7.

2 Background

In the following section we will first discuss the problem statement and an algebraic representation of correlated subqueries. Then we will briefly repeat the unnesting algorithm from [NK15], and we will show the limitations of that approach.

2.1 Problem Statement

Internally, most database systems represent queries in relational algebra. Which means that we need an algebraic representation for correlated subqueries. For that purpose, we use the dependent join, a variant of a regular join. Both are defined below:

$$\begin{aligned} L \bowtie_p R &:= \{l \circ r | l \in L \land r \in R \land p(l \circ r)\} \\ L \text{⧑}_p R &:= \{l \circ r | l \in L \land r \in R(l) \land p(l \circ r)\} \end{aligned}$$

That is, the dependent join makes the left-hand side of the join visible on the right-hand side. Note that the usage of a dependent join is not optional in the presence of correlation. If we wrote, e.g., $L \bowtie (\sigma_{L.x<R.y}(R))$ we would get a type error, as $L.x$ is undefined in the expression $\sigma_{L.x<R.y}(R)$. If we use a dependent join like this $L \text{⧑} (\sigma_{L.x<R.y}(R))$, $L.x$ becomes bound and is therefore available. Note that dependent joins exist for all variants of joins, i.e. left semi join ($\ltimes$), left outer join (⟕) etc. have the corresponding dependent left semi join ($\blacktriangleright\!\!<$), dependent left outer join ($]\!\text{⧑}$) etc. In all of these variants the right-hand side is evaluated for every tuple from the left-hand side.

To simplify this reasoning about available and required attributes we introduce a bit of notation. If T is an expression in relational algebra, $\mathcal{A}(T)$ is the set of attributes produced by that expression. In our example, $\mathcal{A}(R)$ would include $R.y$. If an attribute is used that is not provided by its input (or a higher-up dependent join) it becomes a free variable, we denote that as $\mathcal{F}(T)$. In our example, $\mathcal{F}(R) = \emptyset$ and $\mathcal{F}(\sigma_{L.x<R.y}(R)) = \{L.x\}$. Note that it is not possible to evaluate an algebra expression if $\mathcal{F}(T)$ is not empty, all values have to

be bound first. This can be done either by placing a dependent join on top that provides these attributes or by changing the expression to remove the free variables.

In the canonical translation of SQL into relational algebra, each subquery is added to the outer query via a dependent join (in the case of scalar subqueries via a dependent single join [NLK17]). This makes the attributes of the outer query available to the inner query. However, this comes at a high cost, the quadratic runtime of dependent joins. Thus, the problem of *query unnesting* can be formulated as follows:

Problem Statement: Given a query in relational algebra, including dependent joins. Rewrite the query into an equivalent algebra expression that does not use dependent joins.

Note that we can trivially replace a dependent join with a regular join if the right-hand side does not depend on the left-hand side:

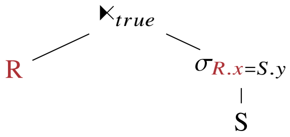
$$L \mathbin{\blacktriangleright\!\!\triangleleft} R \quad \equiv \quad L \bowtie R \quad \text{if } \mathcal{A}(L) \cap \mathcal{F}(R) = \emptyset$$

We usually assume that all dependent joins are non-trivial. Next, we will discuss a bottom-up strategy that eliminates all remaining dependent joins.

2.2 Bottom-Up Unnesting

The paper [NK15] introduces an algorithm that does a bottom-up traversal of the algebra tree and eliminates all dependent joins that it encounters. It uses two strategies for that: First, some dependent joins just exist because of syntax constraints of SQL. For example, this query here

```
SELECT *
FROM R
WHERE EXISTS(SELECT *
             FROM S
             WHERE R.x=S.y)
```



gets translated into a dependent semi join, but it is easy to see that we can pull up the filter condition into the semi join like this: $R \ltimes_{R.x=S.y} S$, eliminating the dependency. The paper calls that *simple unnesting* and always tries it first.

If that is not sufficient to eliminate the dependent join, the approach uses the second strategy, which is more complex but can handle arbitrary queries. It is based on two observations: First, for correlated subqueries it is not really necessary to evaluate the inner query for every tuple of the outer query; it is sufficient to evaluate it once for every possible binding of the free variables. We can compute the *domain* of the free variables D and use that to drive the dependent join:

$$
\begin{aligned}
L \mathbin{\blacktriangleright\!\!\triangleleft} R &\equiv L \Join_{\text{natural } D} (D \mathbin{\blacktriangleright\!\!\triangleleft} R) \quad \text{where} \\
D &:= \Pi_{\mathcal{A}(L) \cap \mathcal{F}(R)}(L)
\end{aligned}
$$

The notation *natural D* here means that we perform a natural join on the attributes of D, which occur on both sides of the join. That join is performed with `IS` semantics, i.e., NULL is considered a dedicated value, and two NULLs compare as equal.

Rewriting the original dependent join into a regular join and a dependent join with the binding domain D has two advantages: First, we know that $|D| \leq |L|$, which means that our query has potentially become cheaper because we execute the inner query less often. Even more importantly, D is duplicate free, which allows for push down rewrites that do not hold for arbitrary tables. The paper then lists a number of rules to push down the dependent join. Here are the most important ones:

$$
\begin{aligned}
D \mathbin{\blacktriangleright\!\!\triangleleft} (\sigma_p(T)) &\rightarrow \sigma_p(D \mathbin{\blacktriangleright\!\!\triangleleft} T) \\
D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 \Join_p T_2) &\rightarrow (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \Join_p T_2 \quad \text{if } \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\
D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 \Join_p T_2) &\rightarrow T_1 \Join_p (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) \quad \text{otherwise if } \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\
D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 \Join_p T_2) &\rightarrow (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \Join_{p \land \text{natural} D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) \quad \text{otherwise} \\
D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 ⟗_p T_2) &\rightarrow (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) ⟗_{p \land \text{natural} D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) \\
D \mathbin{\blacktriangleright\!\!\triangleleft} (\Gamma_{A;agg}(T)) &\rightarrow \Gamma_{A \cup \mathcal{A}(D);agg}(D \mathbin{\blacktriangleright\!\!\triangleleft} T)
\end{aligned}
$$

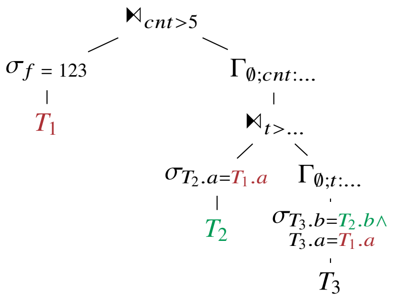
This process is repeated until the dependent join becomes trivial and can be replaced with a regular join. This must always happen because the dependent join is pushed down further with each transformation and the leaves of the algebra tree have no free variables.

As a final step, the query optimizer can decide if it wants to remove the dependent join with D: If all attributes of D are bound in equality conditions, e.g., if D consists of the column x and the query contains the filter condition $x = y$, the join with D can be removed and replaced with $\chi_{x:y}$ instead, such that the column x is computed by the value y. This works because D was duplicate free, thus we know that the dependent join will find at most one join partner, and because our original dependent join was transformed into $\Join_{\text{natural D}}$, and thus the join condition will be enforced at that point of the plan. Conceptually this results in the most elegant plans, as now D is completely gone and the resulting algebra expression looks like any regular query, but performance-wise it might be a bad idea to do so. If the join with D was selective it might be better to keep the join with D, as it removes tuples earlier.

But regardless of whether or not the query optimizer decides to perform the final transformation step, the result will be a completely unnested plan without any dependent joins. While that sounds perfect, we will discuss limitations of that approach next.

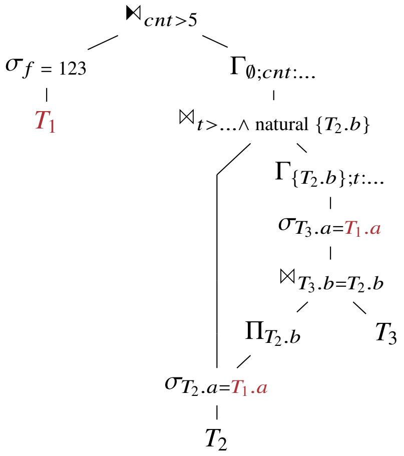
2.3 Limitations of the Bottom-Up Approach

While the bottom-up approach handles most queries just fine, it unfortunately degenerates in some corner cases. We were originally notified about this by Sam Arch, who translated complex UDFs into pure SQL [Fr24]. There, similar to our original example in Figure 1, it could happen that dependent subqueries are nested inside each other. We show a variation[2] below.



$\bowtie_{cnt>5}$
$\sigma_{f\,=\,123}$
T_1
$\Gamma_{\emptyset;cnt:...}$
$\bowtie_{t>...}$
$\sigma_{T_2.a=T_1.a}$
$\Gamma_{\emptyset;t:...}$
T_2
$\sigma_{T_3.b=T_2.b\wedge T_3.a=T_1.a}$
T_3

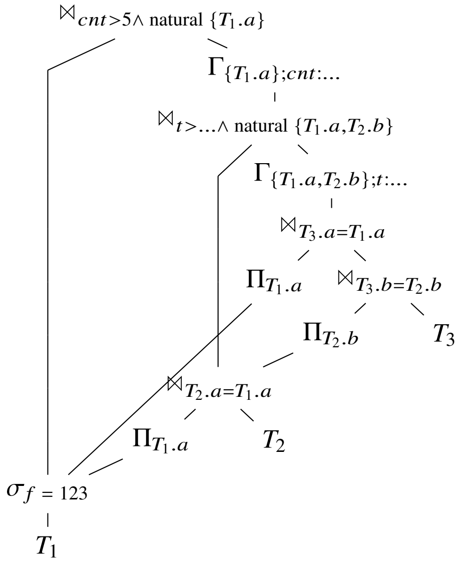
The unnesting algorithm will first unnest the lower dependent join, resulting in the following plan:



The optimizer would probably decide to keep the computation of $\Pi_{T_2.b}(\sigma_{T_2.a=T_1.a}(T_2))$ – instead of using substitution – because the filter condition $T_2.a = T_1.a$ would be selective.

[2] Note that we added a new correlation condition, $T_3.a = T_1.a$, to trigger the problem in a small example.

But if we now run the algorithm again, to eliminate the topmost dependent join, we conceptually evaluate the right-hand side of the lower join for the full cross product of the domains of $T_1.a$ and $T_2.b$, and not for the domain of the join of these two tables:



$\bowtie_{cnt>5\wedge\text{ natural }\{T_1.a\}}$
$\Gamma_{\{T_1.a\};cnt:...}$
$\bowtie_{t>...\wedge\text{ natural }\{T_1.a,T_2.b\}}$
$\Gamma_{\{T_1.a,T_2.b\};t:...}$
$\bowtie_{T_3.a=T_1.a}$
$\Pi_{T_1.a}$
$\bowtie_{T_3.b=T_2.b}$
$\Pi_{T_2.b}$
T_3
$\bowtie_{T_2.a=T_1.a}$
$\Pi_{T_1.a}$
T_2
$\sigma_{f\,=\,123}$
T_1

Note that the result is still correct, because the join condition of $\bowtie_{t>...\wedge\text{ natural }\{T_1.a,T_2.b\}}$ eliminates all $T_1.a$ values from the cross product that do not satisfy the join condition of $\bowtie_{T_2.a=T_1.a}$, but the group by operator processes far too many values.

This problem becomes worse and worse the deeper the nesting is. Sam Arch had sent us a query – tellingly called `crash.sql` – that contained six nested joins and where the unnesting strategy created large intermediate cross products which would all then vanish after filtering of subsequent operators, but which consumed memory and CPU time regardless. This motivated us to develop a holistic unnesting strategy that considered all dependent joins simultaneously, thus avoiding the unnecessary cross products. We will discuss that new algorithm next.

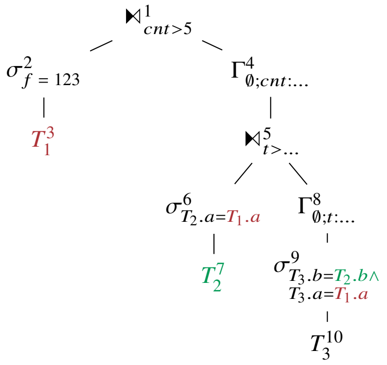
3 Holistic Query Unnesting

In this section, we introduce a holistic unnesting algorithm that processes nested dependent joins together in one pass. It can be seen as a top-down variant of [NK15], as it uses the same idea of pushing down the domain D of the free variable bindings. But we formulate it differently: instead of explicitly pushing down $D\bowtie$ through the tree, the algorithm remembers which D would have to be added, and rewrites all operators in a top-down pass until D becomes unnecessary or can be added safely.

We split the algorithm into three parts: First, a preparatory phase that identifies all non-trivial dependent joins and annotates them with information that the main algorithm needs. Second, the logic to eliminate dependent joins, which will be called for all non-trivial dependent joins in top-to-bottom order and which is the main algorithm, and third, the unnesting rules for individual operators. Note that we do not include a formalization of this approach due to space constraints, formal definitions and a proof of correctness can be found in a technical report [Ne24].

3.1 Identifying Non-Trivial Dependent Joins

First, we have to identify all non-trivial dependent joins, i.e., dependent joins where the right-hand side accesses attributes provided by the left hand side. The easiest way to achieve this is to use an indexed algebra [FMN23], where we can query the structure of the algebra expressions. If this functionality is available, we consider every column access and compute the lowest common ancestor (LCA) of the operator $\circ^1$ that accesses the column and the operator $\circ^2$ that provides the column. If the LCA is not $\circ^1$, it must be a dependent join $⧑^3$ and we annotate $⧑^3$ with the fact that $\circ^1$ is *accessing* the left-hand side of $⧑^3$. The annotations for the example query from the previous section are shown in Figure 2. Dependent joins that have no accessing operators are trivial and can simply be converted into regular joins.


$⧑^1_{cnt>5}$
$\sigma^2_{f\,=\,123}$
$\Gamma^4_{\emptyset;cnt:...}$
T_1^3
$⧑^5_{t>...}$
$\sigma^6_{T_2.a=T_1.a}$
$\Gamma^8_{\emptyset;t:...}$
T_2^7
$\sigma^9_{T_3.b=T_2.b\wedge T_3.a=T_1.a}$
T_3^{10}

$\text{accessing}(⧑^1) := \{\sigma^6, \sigma^9\}$
$\text{accessing}(⧑^5) := \{\sigma^9\}$

Fig. 2: Annotations For Our Example Query

Using an indexed algebra is ideal for this phase because it can do every LCA computation in $O(\log n)$ without any additional data structures. If the DBMS does not support that functionality, the same information can be computed with worse asymptotic complexity by keeping track of the column sets that are available in the different parts of the tree.

3.2 Eliminating Dependent Joins

After identifying all non-trivial dependent joins, we process them in a top-to-bottom order, eliminating each of them. If a dependent join is nested within another dependent join, it will usually be eliminated while the ancestor join is eliminated. That is not always the case because, as we will see below, eliminating the ancestor join might stop before reaching the descendant join. But we never push different D sets across dependent joins, thus eliminating the problem discussed in Section 2.3.

```
fun simpleDJoinElimination(join):
   for op in accessing(join):
      if path from op to join is linear:
         if op is a selection and can merge op into join:
            merge op into join
            remove op from accessing(join)
         else if op is a map and can move op above join:
            move op above join
            remove op from accessing(join)
   if accessing(join) is empty:
      convert join into regular join
   return whether or not we converted the join
```

Fig. 3: Simple Dependent Joins Elimination Join

In our running example, we would start with $⧑^{1}$ then first try a *simple dependent join elimination*. This is a generalization of the simple unnesting from [NK15] and shown in Figure 3: We inspect all operators *op* that are *accessing* the left-hand side of the the join, and we check if the path from that operator to the join consists of only *linear* operators (i.e., operators that allow for partitioning their input, see [G.20]). That can be checked in $O(\log n)$ with an indexed algebra, or with a linear traversal from *op* to *join* otherwise. If yes, we can move *op* up towards the join. If *op* is a selection, we do that if permitted, merging the selection with the join and thus eliminating the accessing operator. For maps (sometimes also called projections), we cannot simply drop the map, but we can check if we could move the map above the join. If that is possible, we do that, again eliminating the operator from the list of *accessing* operators. If afterwards no accessing operator is left, we can simply convert the dependent join into a regular join. That step would be sufficient for the `EXISTS` query from Section 2.2, but in our running example none of the dependent joins can be eliminated like that.

```
struct UnnestingInfo:
   join : join operator of the form $L ⧑ R$
   outerRefs : $\mathcal{A}(L) \cap \mathcal{F}(R)$
   D : domain computation, $\Pi_{\texttt{outerRefs}}(L)$
   parent : parent Unnesting struct (if any)
struct Unnesting:
   info : the UnnestingInfo shared between Unnesting states
   cclasses : union find data structure of equivalent columns
   repr : mapping from outerRefs columns to new columns (if any)
```

Fig. 4: State Maintained During Unnesting

If simple unnesting is not sufficient, the full unnesting algorithm is started. It has to rewrite the right-hand side of the dependent join such that no references from the outer side occur anymore. To do that it maintains some state, as shown in Figure 4. The state is split into the part that remains the same the whole time (`UnnestingInfo`) and the part that changes in the different parts of the query (`Unnesting`). In the global part, it stores the join itself and the set of outer references. That is, the columns from the left that are accessed from the right, and the corresponding domain computation `D`. The `parent` entry allows for recognizing nested dependent joins within the algorithm. In the query fragment local part, it maintains a union-find data structure `cclasses` to keep track of equivalent columns. We use that for transitive reasoning. If the query has predicates of the form $a = b$ and $b = c$, and we have an outer reference to c, we could substitute that with a if a is available. If we have substituted an outer reference with something else we store that in `repr`, which allows us to look up the new name of an outer reference.

The algorithm will substitute outer references with other columns, thus it is convenient to have a helper function that rewrites all column references that occur in an operator, as shown below, that will be called after the suitable replacements have been found:

```
fun rewriteColumns(op, unnesting):
   for each column reference c in op:
      if unnesting.repr contains c:
         replace c with unnesting.repr[c]
```

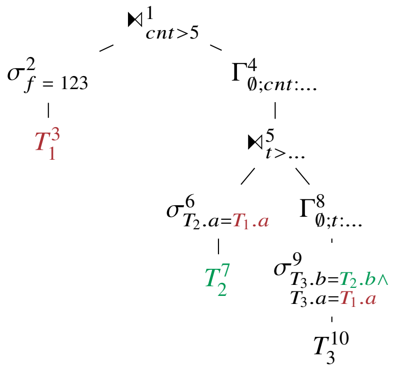
Fig. 5: Helper Function To Rewrite References to Outer Columns

With that infrastructure we can now describe the real elimination algorithm. It will be called for each dependent join, ordered from root to bottom, and can optionally be passed a parent Unnesting structure if we encounter another dependent join during our unnesting (see Section 3.3). It will first check if simple unnesting is sufficient. If yes it will stop, otherwise, it will unnest the left-hand side in the case of nested dependent joins. Then, it creates a new Unnesting structure, merges it with the parent unnesting if needed, and unnests the right-hand side. The pseudo code for that step is shown in Figure 6.

We will illustrate that algorithm with one full unnesting run next. For our running example, the algorithm starts with $\ltimes\!\!\!\rtimes^1$, and then initializes the state as follows:

```
fun dJoinElimination(join, parentUnnesting, parentAccessing):
   // Check if simple unnesting is sufficient
   if simpleDJoinElimination(join):
      // Handle the outer unnesting if needed
      if parentUnnesting is set:
         for each map m moved by simpleDJoinElimination:
            rewriteColumns(m, parentUnnesting)
         unnest(join, parentUnnesting, parentAccessing)
      return

   // In the nested case we have to unnest the left-hand side first
   if parentUnnesting is set:
      accLeft = ∅
      for a in parentAccessing:
          if a is contained in join.left:
             insert a into accLeft
      unnest(join.left, parentUnnesting, accLeft)
      // Update our condition as needed
      rewriteColumns(join.condition.parentUnnestring)
      for each map m moved by simpleDJoinElimination:
         rewriteColumns(m, parentUnnesting)

   // Create a new unnesting struct
   info = new UnnestingInfo for join, set parent=parentUnnesting
   unnest = new Unnesting for info

   // Merge with parent unnesting if needed
   acc = accessing(join)
   if parentUnnesting is set:
      for each a in parentAccessing:
         if a is contained in join.right:
            insert a into acc

   // Unnest right-hand side
   add equivalences from join.condition to unnest.cclasses
   unnest(join.right, info, acc)

   for each c in info.outerRefs:
      add "{c}␣is␣not␣distinct␣from␣{info.repr[c]}" to join.condition
```

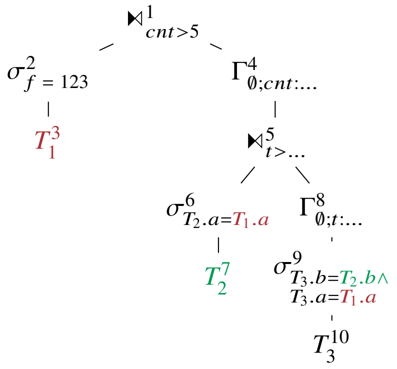
Fig. 6: The General Case of Dependent Join Elimination



outerRef:=$\{T_1.a\}$
cclasses:=$\emptyset$
repr:=$\emptyset$

next: unnest(Γ^4, $\{\sigma^6, \sigma^9\}$)

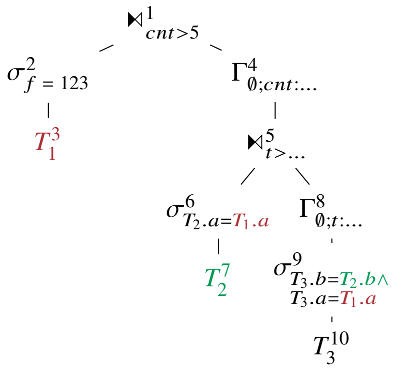
We will cover the exact rules for the `unnest` subroutines in Section 3.3, but intuitively we unnest the group by Γ^4 by recursively unnesting its input and then later adding the outer column references to the group by. Performing those steps leads to the following state:



outerRef:=$\{T_1.a\}$
cclasses:=$\emptyset$
repr:=$\emptyset$

stack: [Γ^4]
next: unnest($\bowtie^5$, $\{\sigma^6, \sigma^9\}$)

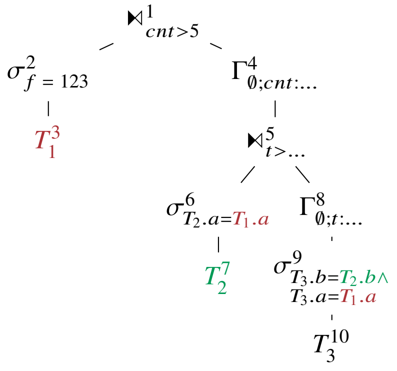
We encounter another dependent join, one where simple elimination is not sufficient. We will merge the unnesting state with the outer dependent join later; for now, we unnest the left input:


⧑$^1_{cnt>5}$
$\sigma^2_{f=123}$
$\Gamma^4_{\emptyset;cnt:...}$
T_1^3
⧑$^5_{t>...}$
$\sigma^6_{T_2.a=T_1.a}$
$\Gamma^8_{\emptyset;t:...}$
T_2^7
$\sigma^9_{T_3.b=T_2.b\wedge T_3.a=T_1.a}$
T_3^{10}

outerRef:={$T_1.a$}
cclasses:={{$T_1.a, T_2.a$}}
repr:=∅

stack: [Γ^4, ⧑^{5}]
next: unnest(σ^6, {σ^6})

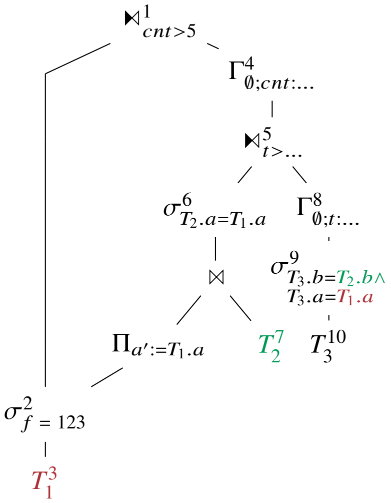
When unnesting the select, we add the equivalence condition implied by the select to the `cclasses` data structure, remove the select from the `accessing` list (the second function argument of `unnest`), and recurse to the input:


⧑$^1_{cnt>5}$
$\sigma^2_{f=123}$
$\Gamma^4_{\emptyset;cnt:...}$
T_1^3
⧑$^5_{t>...}$
$\sigma^6_{T_2.a=T_1.a}$
$\Gamma^8_{\emptyset;t:...}$
T_2^7
$\sigma^9_{T_3.b=T_2.b\wedge T_3.a=T_1.a}$
T_3^{10}

outerRef:={$T_1.a$}
cclasses:={{$T_1.a, T_2.a$}}
repr:=∅

stack: [Γ^4, ⧑^{5}, σ^6]
next: unnest(T_2^7, {})

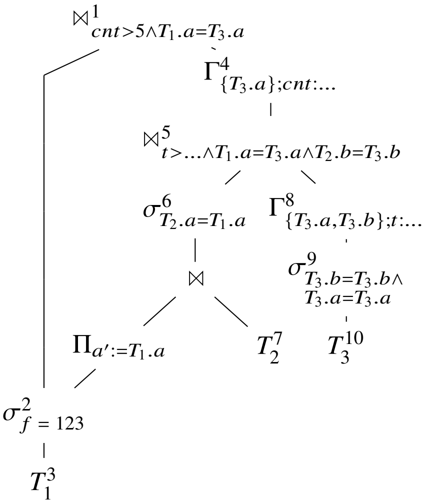
At this point the `accessing` list is empty and we have two choices: either we introduce a join with $D := \Pi_{a:=T_1.a}(\sigma_{f=123}(T_1))$ and use that to substitute $T_1.a$; or we exploit the fact that all outer column references are bound in `cclasses` and we simply replace $T_1.a$ with the equivalent $T_2.a$. The optimal choice depends on the selectivity of σ^2. For now we will simply assume that σ^2 is selective, and thus we introduce a join. Then we pop up the stack until we reach ⧑^{5} again, rewriting columns along the way. Note that in this case, the rewrite does nothing because we decided to not substitute $T_1.a$ with $T_2.a$. Our next `unnest` call is for Γ^8:



outerRef:=$\{T_1.a, T_2.b\}$
cclasses:=$\emptyset$
repr:=$\emptyset$

stack: [Γ^4, $\rtimes^5$]
next: unnest(Γ^8, $\{\sigma^9\}$)

The other side is processed analogously, we recurse below the group by; then below the select; remove that σ^9 from the `accessing` list; and add its constraints to the equivalence classes. Then again we have to decide if we want to add a join or use substitution. To show both cases, we now assume that the optimizer decides to use substitution (even though that is unlikely for this query). Then, we add $T_1.a \to T_3.a, T_2.b \to T_3.b$ to `repr` and rewrite all columns on the way up. This gives us the following unnested plan:


$\bowtie^1_{cnt>5\wedge T_1.a=T_3.a}$
$\Gamma^4_{\{T_3.a\};cnt:...}$
$\bowtie^5_{t>...\wedge T_1.a=T_3.a\wedge T_2.b=T_3.b}$
$\sigma^6_{T_2.a=T_1.a}$
$\Gamma^8_{\{T_3.a,T_3.b\};t:...}$
$\bowtie$
$\sigma^9_{T_3.b=T_3.b\wedge T_3.a=T_3.a}$
$\Pi_{a':=T_1.a}$
T_2^7
T_3^{10}
$\sigma^2_{f\ =\ 123}$
T_1^3

3.3 Unnesting Rules

We now briefly discuss the unnesting rules for individual operators. These are analogous to the push down rules from [NK15], thus we only give an overview over the most important rules due to space constraints. All operators share the common logic that 1) if we call `unnest(○, info, accessing)` and `accessing` is empty, unnesting stops, and we either replace ○ with `info.D`⋈○ or we substitute all outer references with equivalent columns from `info.cclasses`, and 2) when recursing to an input of an operator ○, ○ is removed from the `accessing` list if needed. We do not list that code explicitly.

For selections, the unnesting logic registers the filter conditions and rewrites all columns on the way back:

```
fun unnest(select, info, accessing):
   add equivalences from select.predicate to info.cclasses
   unnest(select.input, info, accessing)
   rewriteColumns(select.condition, info)
```

Similar for maps, where we only have to rename columns after unnesting the input:

```
fun unnest(map, info, accessing):
   unnest(map.input, info, accessing)
   rewriteColumns(map.computations, info)
```

The group by operator unnests its input and adds the outer references to the group by attributes. Note that there is a special case where the group by is *static* (i.e., a select clause with aggregation functions but without `group by`). In that case, we must produce a non-empty output for an empty input, and we provide that with an outer join (or a group join [MN11] if supported by the system):

```
fun unnest(groupby, info, accessing):
   static = groupby.groups is empty or groups.groupingsets contains ∅
   unnest(groupby.input, info, accessing)
   rewriteColumns(groupby, info)
   for c in info.outerRefs:
      add info.repr[c] to groupby.groups
   if static:
      replace groupby with info.D⋈groupy, joining on mapped info.outerRefs
```

Window operators are conceptually handled similar to group by. We have to add the outer references to the partition by clause in order to compute the window functions separately for each binding:

```
fun unnest(window, info, accessing):
   unnest(window.input, info, accessing)
   rewriteColumns(window, info)
   for c in info.outerRefs:
      add info.repr[c] to window.partitionby
```

For joins, we check if we have encountered another dependent join, merging both removals if needed. Otherwise, we check if all dependent accesses occur only on one side. If yes, and if the join type does not have to keep track of the number of join partners, we can simply recurse, like a selection. If not, we unnest both sides and update the join condition as needed:

```
fun unnest(join, info, accessing):
   split accessing into accessingLeft and accessingRight for input of join

   // Check if we have encounter another dependent join
   if accessing(join) is not empty:
      dJoinElimination(join, info, accessing)
      return

   // Check if only one side accesses outer columns
   if accessingRight is empty and info.join cannot output unmatched from the right:
      unnest(join.left, info, accessingLeft)
      rewriteColumns(join.condition, info)
      return
   if accessingLeft is empty and info.join cannot output unmatched from the left:
      unnest(join.right, info, accessingRight)
      rewriteColumns(join.condition, info)
      return

   // Unnest both sides
   unnestingLeft = new Unnesting(info.info)
   unnestingRight = new Unnesting(info.info)
   unnest(join.left, unnestLeft, accessingLeft)
   unnest(join.right, unnestRight, accessingRight)
   rewriteColumnsForJoin(join.condition, unnestingLeft, unnestingRight)
   for c in info.outerRefs:
      add "{unnestLeft.repr[c]}␣is␣not␣distinct␣from␣{unnestRight.repr[c]}" to join.condition
   merge cclasses and repr from unnestLeft and unnestRight into info
```

Note that a different logic for rewriting columns is needed for joins, as we have to take the join type into account to decide which side of the join to use for the replacement. A left outer join, which outputs unmatched rows from the left, would use the left side for the replacement, while a right outer join would use the right side. A full outer join must determine the used side per row, by replacing the columns with expressions such as `COALESCE(unnestLeft.repr[c], unnestRight.repr[c])`.

4 Handling Complex SQL Constructs

While the push down rules from [NK15] and Section 3.3 are sufficient for most queries, there are some complex SQL constructs that are not obvious to unnest, in particular, correlated subqueries in full join conditions, `WITH RECURSIVE`, common-table expressions (CTEs), and `ORDER BY ... LIMIT`. Most systems cannot unnest these constructs, and there has only been little research on how to unnest recursion [FGL24]. We will look at these constructs next.

4.1 Unnesting Correlated Subqueries in Full Join Conditions

The SQL standard allows join conditions to access columns from both sides of the join. The SQL standard also allows subqueries in arbitrary expressions. Thus, it is possible to write a query where the join condition contains a subquery that references attributes from the left and the right side of the join. If the join is a inner or left outer join, that subquery can be cross applied to the right input of the join, and the query can be unnested normally. Right outer joins can be handled by flipping the sides. However, full outer joins are more complicated. The semantics of correlated full outer joins are not well-defined, as the definition for “unmatched tuples” is not clear in the presence of dynamic inputs. Thus, we can not just generate a cross apply for the subquery and unnest the query normally. We instead use the following equivalence

$$R \mathbin{\text{⟗}}_{p(T)} S \equiv R \mathbin{\text{⟗}}_{m \wedge (R.x \text{ is } D_R.x)} \left(S \mathbin{\text{⟕}}_{m \wedge (S.y \text{ is } D_S.y)} \left(\chi_{m:p'}((D_R \times D_S) \mathbin{\text{⧑}} T)\right)\right)$$

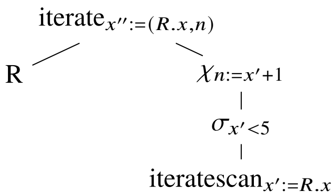
where T is a singleton subquery from within the join condition, p is the join condition that references the results of the subquery, D_R and D_S are the domains of R and S projected to the free variables of p and T (denoted as x and y, respectively), and p' is p remapped to use columns from D_R and D_S instead of R and S. We will attempt to break down the equivalent tree bottom-up. Firstly, the subquery $\left(\chi_{m:p}((D_R \times D_S) \mathbin{\text{⧑}} T)\right)$ computes the predicate for every possible binding of the outer references. Secondly, the left outer join with S on the result of the predicate (via m) ensures that the unmatched tuples from S are only produced once in total, while the matching tuples are produced for every binding of the outer references from R. Finally, the full outer join with R on the result of the predicate (via m) provides the final result. Note that the inner subquery $(D_R \times D_S) \mathbin{\text{⧑}} T$ needs to be further unnested.

Instead of generating the subquery $(D_R \times D_S) \mathbin{\text{⧑}} T$ and unnesting it from scratch, we directly call `unnest` on the subquery T to accomplish the same effect. In this case, we use a slightly modified `UnnestingInfo` structure which ensures the domain `D` represents a cross product of D_R and D_S and that the conditions of *both* joins are amended by the corresponding `is not distinct from` predicates.

4.2 Unnesting WITH RECURSIVE

We assume that a `WITH RECURSIVE` construct is translated into an *iterate* operator [Pa17] and a corresponding *iteratescan* (DuckDB calls these `recursive_cte` amd `recursive_cte_scan`), where the left hand side of the iterate computes the base table and the right hand side is evaluated until empty, reading the last iteration in *iteratescan*:

```
WITH RECURSIVE REC(x) AS (
   SELECT x FROM R
   UNION ALL
   SELECT x+1 from REC WHERE x<5)
SELECT x FROM REC
```


$\text{iterate}_{x'':=(R.x,n)}$
R
$\chi_{n:=x'+1}$
$\sigma_{x'<5}$
$\text{iteratescan}_{x':=R.x}$

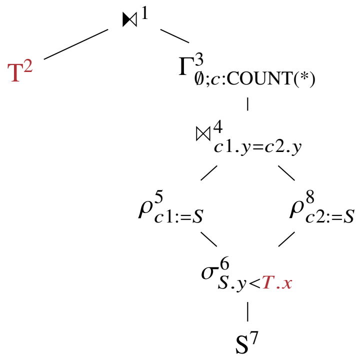
When unnesting this construct, we want to conceptually evaluate the iteration for every possible binding of the outer references. We achieve this by adding the outer references to the columns of the recursive CTE. But now we must make sure that the repeated part is conceptually partitioned by the outer references, too. Thus, we add all `iterationscan` operators of the current `iteration` to the `accessing` list. This makes sure that the unnesting phase reaches these operators, adds the outer references to the scans, too, and rewrites all intermediate operators along the way to take the outer references into account:

```
fun unnest(iteration, info, accessing):
   // Mark iteration scans as accessing
   split accessing into accessingLeft and accessingRight for input of iteration
   for s in iterationscans of iteration:
      add s to accessingRight

   // Unnest the seed side first to make the new columns available
   unnestingLeft = new Unnesting(info.info)
   unnest(iteration.left, unnestLeft, accessingLeft)
   rewriteColumns(iteration.leftColumns, unnestingLeft)
   for c in info.outerRefs:
      add unnestLeft.repr[c] to iteration.leftColumns
      info.repr[c]=unnestLeft.repr[c]

   // Unnest the iteration side and add the corresponding columns
   unnestingRight = new Unnesting(info.info)
   unnest(iteration.right, unnestRight, accessingRight)
   for c in info.outerRefs:
      add unnestRight.repr[c] to iteration.rightColumns

fun unnest(iterationscan, info, accessing):
   // Add the new columns to the scan
   i = iteration operator for iterationscan
   for c in info.outerRefs:
      cb = corresponding column for c in i.leftInput
      add cn:=cb as produced column to iterationscan
      info.repr[c]=cn
```

4.3 Unnesting CTEs

Another complication arises if the algebraic expression is not actually a tree, but a DAG, which can occur in CTEs. Consider the query example below:

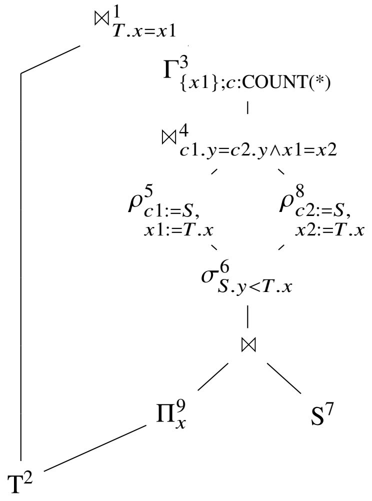
```
SELECT *
FROM S
WHERE (WITH CTE AS
        (SELECT * FROM T
         WHERE T.y<S.x)
    SELECT COUNT(*)
    FROM CTE c1 JOIN CTE c2
         ON c1.y=c2.y) = 1
```



In this example we can ignore the fact that $⧑^1$ has to check for duplicates because Γ^3 is duplicate free. When executing the algorithm from Section 3.1, σ^6 is placed in the `accessing` list of $⧑^1$. But this causes problems when unnesting $\bowtie^4$, since we can no longer split that `accessing` list into left and right, because σ^6 is reachable over both paths. Furthermore, we would try to unnest the shared part twice, which is also bad because it would add domain scans multiple times.

A better strategy is to cut the DAG into a set of trees such that shared operators form their own subtrees. Here, σ^6 would be the root of a new tree, and ρ^5 and ρ^8 would be leaf operations. When collecting accessing operators, we check if they occur in the same tree. If yes, we add them as described before; if not, we instead consider all shared reads as accessing and repeat the procedure. Additionally, one of the shared reads, e.g., the one with the smallest id, is designated to trigger unnesting the shared operator when unnesting reaches it. In this example, this means that `accessing(`$⧑^1$`)` $= \{\rho^5, \rho^8\}$, with ρ^5 forwarding the unnesting pass to σ^6. The end result is the algebra expression shown below:

```
SELECT *
FROM S
WHERE (WITH CTE AS
        (SELECT * FROM T
         WHERE T.y<S.x)
    SELECT COUNT(*)
    FROM CTE c1 JOIN CTE c2
         ON c1.y=c2.y) = 1
```


$⧑^1_{T.x=x1}$
$\Gamma^3_{\{x1\};c:\text{COUNT(*)}}$
$\bowtie^4_{c1.y=c2.y \wedge x1=x2}$
$\rho^5_{c1:=S,\ x1:=T.x}$
$\rho^8_{c2:=S,\ x2:=T.x}$
$\sigma^6_{S.y<T.x}$
$\bowtie$
Π^9_x
S^7
T^2

4.4 Unnesting ORDER BY LIMIT

Subqueries with `ORDER BY` and `LIMIT` or `OFFSET` have to be changed during unnesting because now the limit has to be enforced per value of the outer bindings instead of globally. In the Umbra system, the sort operator offers that functionality natively, which makes unnesting simple, but other systems will want to rewrite a (sub-)query like this

```
SELECT *
FROM S
ORDER BY x LIMIT l OFFSET o
```

into the equivalent window query:

```
SELECT *
FROM (SELECT *, ROW_NUMBER() OVER (ORDER BY x) AS RN FROM S) S
WHERE RN BETWEEN o+1 AND l+o
```

Now unnesting becomes straightforward; we simply have to add the outer references as `PARTITION BY` entries to the `OVER` clause.

5 Evaluation

Query unnesting improves the asymptotic complexity of query execution by eliminating the quadratic runtime overhead of dependent joins. Therefore, we can easily demonstrate arbitrary performance gains by simply increasing the dataset size. This makes performance experiments a bit pointless, as the impact is so huge. To illustrate this point, consider the example query from Figure 1. This query needs over 26 minutes on TPC-H SF1 when executed in Postgres 16 on a Ryzen 9 7950X3D with 128GB RAM. After applying our algorithm, Postgres can execute the query in 1.3s, a speed-up of more than 1000x.

We observe a similar effect when comparing the unnesting algorithm from [NK15] with our new top-down formulation. As a demonstration query, we use the query from Figure 1 and add `c_comment=l_comment` to the last `WHERE` clause, which brings it into a similar shape as the problematic case from Section 2.3. When executing the query in the Umbra system when using the old unnesting logic, the query is terminated after exhausting 120 GB of main memory. Using the new top-down formulation it finishes in 33ms on the same machine. Again this is a speed-up of several orders of magnitude at least, when comparing the time until the original unnested plan was cancelled.

Similarly, for the original query from Sam Arch that motivated this work, which is procbench UDF Query 18 [GR21] after passing through Apfel [FHG22]: The unnesting strategy from [NK15] leads to memory exhaustion, while with our new top-down unnesting Umbra answers the query in 251ms on TPC-DS SF1.

6 Related Work

The first unnesting rules were given as transformations of the SQL representation [Ki82, JK84], which were easy to understand and were integrated into several commercial database systems. Famously, the original rules proposed by Kim contained the *count* bug, which led to wrong results for empty inputs. This was corrected by Kiessling [Ki85]. However, fundamentally these rules all suffer from the fact that they can address specific patterns of queries but fail to cover all cases. There has been a lot of follow-up work on adding new rules to cover even more cases, but none of them were complete. The *magic set* based technique from IBM [Se96] can be used to tackle the runtime aspect of dependent joins by introducing semi joins with the domain of the outer bindings, conceptually similar to the join with D in our approach. While this helps to speed up query processing, that paper does not provide a mechanism to fully eliminate dependent joins. A later generalization allowed for the elimination of dependent joins if the subquery contains an aggregation function [Zu03]. The first approach to provide a complete solution was [NK15], which we already discussed in Section 2 and upon which this work builds. However, while that approach is conceptually complete, it does not describe all SQL constructs, and it runs into problems if dependent joins are nested. We address both limitations in this work.

The need for automatic unnesting has increased in recent years because an increasing number of queries are machine-generated and some of these generators produce correlated subqueries. A recent example is the Apfel transpiler [FHG22] that can translate stored procedures from PSQL into pure SQL. This transformation is very attractive because it eliminates the need to provide a dedicated PSQL implementation, and the resulting queries are actually faster than the original PSQL code. This requires that the database system can unnest correlated subqueries efficiently.

7 Conclusion

Unnesting correlated subqueries is essential for handling subqueries efficiently. Eliminating the correlation improves the asymptotic complexity, and can easily lead to speedups by several orders of magnitude. In this work, we have extended our previous work on bottom-up decorrelation by correctly handling nested dependent joins and by supporting a wider range of SQL constructs. The new approach has already been integrated into our Umbra system and we hope that other systems like DuckDB and Postgres will follow.

Acknowledgments: We thank Altan Birler for providing the unnesting strategy for full outer joins with correlated predicates.

Bibliography

[FGL24] Fejza, Amela; Genevès, Pierre; Layaïda, Nabil: Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers. Proc. VLDB Endow., 17(11):3095–3108, 2024.

[FHG22] Fischer, Tim; Hirn, Denis; Grust, Torsten: Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In (Ives, Zachary G.; Bonifati, Angela; Abbadi, Amr El, eds): SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022. ACM, pp. 2389–2392, 2022.

[FMN23] Fent, Philipp; Moerkotte, Guido; Neumann, Thomas: Asymptotically Better Query Optimization Using Indexed Algebra. Proc. VLDB Endow., 16(11):3018–3030, 2023.

[Fr24] Franz, Kai; Arch, Samuel; Hirn, Denis; Grust, Torsten; Mowry, Todd C.; Pavlo, Andrew: Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In: 14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024. www.cidrdb.org, 2024.

[G.20] G. Moerkotte: Building Query Compilers. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, 2020.

[GR21] Gupta, Surabhi; Ramachandra, Karthik: Procedural Extensions of SQL: Understanding their usage in the wild. Proc. VLDB Endow., 14(8):1378–1391, 2021.

[JK84] Jarke, Matthias; Koch, Jürgen: Query Optimization in Database Systems. ACM Comput. Surv., 16(2):111–152, 1984.

[Ki82] Kim, Won: On Optimizing an SQL-like Nested Query. ACM Trans. Database Syst., 7(3):443–469, 1982.

[Ki85] Kießling, Werner: On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. In: VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden. pp. 241–250, 1985.

[MN11] Moerkotte, Guido; Neumann, Thomas: Accelerating Queries with Group-By and Join by Groupjoin. Proc. VLDB Endow., 4(11):843–851, 2011.

[Ne24] Neumann, Thomas: A Formalization of Top-Down Unnesting, 2024. https://arxiv.org/abs/2412.04294.

[NK15] Neumann, Thomas; Kemper, Alfons: Unnesting Arbitrary Queries. In (Seidl, Thomas; Ritter, Norbert; Schöning, Harald; Sattler, Kai-Uwe; Härder, Theo; Friedrich, Steffen; Wingerath, Wolfram, eds): Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"(DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings. volume P-241 of LNI. GI, pp. 383–402, 2015.

[NLK17] Neumann, Thomas; Leis, Viktor; Kemper, Alfons: The Complete Story of Joins (in HyPer). In (Mitschang, Bernhard; Nicklas, Daniela; Leymann, Frank; Schöning, Harald; Herschel, Melanie; Teubner, Jens; Härder, Theo; Kopp, Oliver; Wieland, Matthias, eds): Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings. volume P-265 of LNI. GI, pp. 31–50, 2017.

[Pa17] Passing, Linnea; Then, Manuel; Hubig, Nina C.; Lang, Harald; Schreier, Michael; Günnemann, Stephan; Kemper, Alfons; Neumann, Thomas: SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017. OpenProceedings.org, pp. 84–95, 2017.

[Se96] Seshadri, Praveen; Hellerstein, Joseph M.; Pirahesh, Hamid; Leung, T. Y. Cliff; Ramakrishnan, Raghu; Srivastava, Divesh; Stuckey, Peter J.; Sudarshan, S.: Cost-Based Optimization for Magic: Algebra and Implementation. In (Jagadish, H. V.; Mumick, Inderpal Singh, eds): Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996. ACM Press, pp. 435–446, 1996.

[Zu03] Zuzarte, Calisto; Pirahesh, Hamid; Ma, Wenbin; Cheng, Qi; Liu, Linqi; Wong, Kwai: WinMagic : Subquery Elimination Using Window Aggregation. In (Halevy, Alon Y.; Ives, Zachary G.; Doan, AnHai, eds): Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003. ACM, pp. 652–656, 2003.