Unnesting Arbitrary Queries

Thomas Neumann and Alfons Kemper
Technische Universität München
Munich, Germany
neumann@in.tum.de, kemper@in.tum.de

Abstract: SQL-99 allows for nested subqueries at nearly all places within a query. From a user's point of view, nested queries can greatly simplify the formulation of complex queries. However, nested queries that are correlated with the outer queries frequently lead to dependent joins with nested loops evaluations and thus poor performance.

Existing systems therefore use a number of heuristics to *unnest* these queries, i.e., de-correlate them. These unnesting techniques can greatly speed up query processing, but are usually limited to certain classes of queries. To the best of our knowledge no existing system can de-correlate queries in the general case. We present a generic approach for unnesting arbitrary queries. As a result, the de-correlated queries allow for much simpler and much more efficient query evaluation.

1 Introduction

Subqueries are frequently used in SQL queries to simplify query formulation. Consider for our running examples the following schema:

- students: $\{[\text{id, name, major, year}, \dots]\}$
- exams: $\{[\text{sid, course, curriculum, date}, \dots]\}$

Then the following is a nested query to find for each student the best exams (according to the German grading system where lower numbers are better):

```
Q1: select s.name,e.course
    from   students s,exams e
    where  s.id=e.sid and
           e.grade=(select min(e2.grade)
                    from exams e2
                    where s.id=e2.sid)
```

Conceptually, for each student, exam pair (s, e) it determines, in the subquery, whether or not this particular exam e has the best grade of all exams of this particular student s.

From a performance point of view the query is not so nice, as the subquery has to be re-evaluated for every student, exam pair. From a technical perspective the query contains a

dependent join, i.e., a nested loop join where the evaluation of the right hand side depends on the current value of the left-hand side. These joins are highly inefficient, and lead to (at least) quadratic execution time.

Database management systems (DBMSs) therefore internally rewrite the query to eliminate the correlation. A SQL representation of this rewrite would look like this:

```
Q1': select s.name,e.course
     from   students s,exams e,
            (select e2.sid as id, min(e2.grade) as best
             from exams e2
             group by e2.sid) m
     where  s.id=e.sid and m.id=s.id and
            e.grade=m.best
```

Here, the evaluation of the subquery no longer depends on the values of `s`, and thus regular joins can be used. This kind of unnesting is very important for good query performance, but existing techniques cannot handle arbitrary queries. For example the subsequent SQL query is very hard to de-correlate. It determines the exams that a CS or Games Engineering student should repeat in the future because he or she underachieved in comparison to the average grade of exams taken by him/her or taken by elder peers:

```
Q2:
select s.name, e.course
from   students s, exams e
where  s.id=e.sid and
  (s.major = 'CS' or s.major = 'Games Eng') and
  e.grade>=(select avg(e2.grade)+1 --one grade worse
            from exams e2          --than the average grade
            where s.id=e2.sid or   --of exams taken by
                  (e2.curriculum=s.major and --him/her or taken
                  s.year>e2.date))           --by elder peers
```

To the best of our knowledge, no existing system can unnest such a query. And indeed, unnesting this query is hard: Standard unnesting techniques rely upon the fact that attributes available within the query can be used to substitute the free variables determined by the outer query. This is not the case here, `s.year` for example cannot be substituted.

So clearly this kind of complicated correlated query will be more expensive to evaluate than a more simple subquery. However, as we will show, it is indeed possible to unnest even this query. We will have to spend extra effort to derive the value of `s.year` and `s.major`, but we can do so without a dependent join. And the extra effort we will have to spend is bound by the cost of the dependent join. Most queries will be dramatically more efficient in the decorrelated form, in the worst case we will have the some join effort. That is, our unnesting approach will definitely not incur higher costs than the straightforward nested loops evaluation – and in the majority of cases improve the performance dramatically, often by several orders of magnitude. Furthermore, even the worst case is

most likely a win, as eliminating dependent joins allows for more efficient join implementations. Our contribution can thus be seen as a universally applicable technique for unnesting any kind of nested subquery – in contrast to the special case treatments published and implemented so far. The universal unnesting technique has been fully implemented in our main-memory database system HyPer [KN11] and can be experienced via our web interface `hyper-db.de` that visualizes the resulting query plans.

And the typical performance gains of query unnesting are immense: Depending on the query, it replaces an $O(n^2)$ algorithm (nested loop join) with an $O(n)$ algorithm (hash join, joining keys). Furthermore the dependent side is executed for every outer tuple in the nested case, but only once in the unnested case. On large data sets it is easy to get a factor 10 or even 100 performance improvement by unnesting, which makes unnesting an essential technique for query compilation. There are a few cases where nested evaluation is actually beneficial, in particular if the outer side is very small and the inner side can be evaluated using an index lookup, but that should be triggered by a conscious decision of the query optimizer, not by the way the query is formulated. By default, queries should be unnested completely.

The rest of this paper is structured as follows: We first define the notation used in this paper in Section 2. Then, in Section 3 the algebraic unnesting transformations are specified. Section 4 covers further optimisation rules that are applicable in special cases (e.g., when functional dependencies can be inferred). Section 5 is devoted to a "cursory" performance evaluation that analyses some other well-known DBMSs to our HyPer system which incorporates the unnesting described. Finally, we survey the related work and conclude the paper.

2 Preliminaries

Before looking at the unnesting techniques, we briefly repeat some definitions for relational algebra, as the notation is not standardized beyond the basic operators.

First, we have the regular *(inner) join*, which is simply defined as cross product followed by a selection:

$$T_1 \bowtie_p T_2 \quad := \quad \sigma_p(T_1 \times T_2).$$

It computes the combination of all matching entries from T_1 and T_2. It is used in most SQL queries, but its definition is not sufficient in the presence of correlated subqueries. The subquery has to be evaluated for every tuple of the outer query, therefore we define the *dependent join* as

$$T_1 \ltimes\!\!\!\!\!\rtimes_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$$

Here, the right hand side is evaluated for every tuple of the left hand side. We denote the attributes produced by an expression T by $\mathcal{A}(T)$, and free variables occurring in an expression T by $\mathcal{F}(T)$. To evaluate the dependent join, $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$ must hold, i.e., the attributes required by T_2 must be produced by T_1.

Note that in this paper we sometimes explicitly mention *natural join* in the join predicate to simplify the notation. We assume that all relations occuring in a query will have unique attribute names, even if they reference the same physical table, thus $A \bowtie B \equiv A \times B$. However, if we explicitly reference the same relation name twice, and call for the natural join, then the attribute columns with the same name are compared, and the duplicate columns are projected out. Consider, for example:

$$(A \bowtie C) \bowtie_{p \wedge \text{natural join } C} (B \bowtie C)$$

Here, the top-most join checks both the predicate p and compares the columns of C that come from both sides (and eliminates one of the two copies of C's columns).

For semi joins ($\ltimes$), anti joins ($\triangleright$), and outer joins ($⟕$, $⟗$) we define the dependent variants accordingly ($◣$, $▶$, $⟕$, $⟗$), again the right-hand side is evaluated for every tuple of the left-hand side.

Besides the join operators, we have the *group by* operator as additional important operator

$$\Gamma_{A;a:f}(e) \quad := \quad \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}$$

It groups its input e (i.e., a base relation or a relation computed from another algebra expression) by A, and evaluates one (or more comma separated) aggregation function(s) to compute aggregated attributes. If A is empty, just one aggregation tuple is produced – as in SQL with a missing group by-clause.

We can evaluate functions (and thus construct new attributes) by evaluating the *map* operator

$$\chi_{a:f}(e) \quad := \quad \{x \circ (a : f(x)) | x \in e\}.$$

Besides these, we need the regular relational algebra operators ($\sigma, \times, \Pi, \rho, \cup, \cap, \setminus$). Using these operators, we can translate SQL queries into relational algebra.

In the following we will often have to compare sets of attributes. As a shorthand notation, we define the attribute comparison operator $=_A$ as

$$t_1 =_A t_2 \quad := \quad \forall_{a \in A} : t_1.a = t_2.a.$$

Note that unless indicated otherwise this operator has *is* semantics, i.e., it compares NULL values as equal.

3 Unnesting

The algebraic representation of a query with correlated subqueries (initially) results in a dependent join, i.e., an expression of the form

$$T_1 \bowtie_p T_2.$$

As already mentioned, these dependent joins are very unfortunate from a performance perspective, and we want to eliminate them. Fundamentally, we manipulate the algebraic expression until the right hand side no longer depends on the left hand side, and thus the dependent join can be transformed into a regular join. We achieve this using two techniques that we will discuss in the following. First, we try a *simple* unnesting, that handles cases where dependencies are created just for syntactic reasons. If that is not sufficient to unnest the query, we use the *general* unnesting framework that can handle arbitrary complex queries.

3.1 Simple Unnesting

Sometimes queries contain correlated subqueries just because they are simpler to formulate in SQL. An example for that is TPC-H Query 21, which contains a construct similar to the fragment

```
select ...
from   lineitem l1 ...
where  exists (select *
               from lineitem l2
               where l2.l_orderkey = l1.l_orderkey)
...
```

This is translated into an algebra expression of the form

$$l_1 \ltimes (\sigma_{l_1.okey=l_2.okey}(l_2))$$

It is easy to see that this fragment can be unnested by moving the dependent predicate up the tree, transforming the dependent join into a regular join:

$$l_1 \ltimes_{l_1.okey=l_2.okey} (l_2)$$

In general the simple unnesting phase moves all dependent predicates up the algebra tree as far as possible, potentially beyond joins, selections, group by, etc., until it reaches a point where all its attributes are available from the input. If this happens the dependent join can be transformed into a regular join, as shown by the equivalence explained above. Note that this predicate pull-up happens purely for decorreleation reasons. Further optimization steps might push (parts of) the predicate back down again to filter tuples early on.

3.2 General Unnesting

Predicate movement is very easy to implement and already sufficient to handle frequently occuring simply nested queries. Therefore we try it first, but for the general case we need a more complex approach: First, we translate the dependent join into a "nicer" dependent join (i.e., one that is easier to manipulate), and second, we will push the new dependent join down into the query until we can transform it into a regular join.

Thus, in the first step, we use the following equivalence

$$T_1 \ltimes\!\!\!\rtimes_p T_2 \quad \equiv \quad T_1 \bowtie_{p \wedge T_1 =_{\mathcal{A}(D)} D} (D \ltimes\!\!\!\rtimes T_2)$$

where $D := \Pi_{\mathcal{F}(T_2) \cap \mathcal{A}(T_1)}(T_1)$.

At a first glance this transformation did not improve the query plan much, as we have replaced one dependent join by a regular join and another dependent join. However, at a second glance this transformation is very helpful: In the original expression, we had to evaluate T_2 for every tuple of T_1, which could be millions. Therefore, in the second expression, we first compute the domain D of all variable bindings, evaluate T_2 only once for every distinct variable binding, and then use a regular join to match the results to the original T_1 value. If there are a lot of duplicates, this already greatly reduces the number of invocations of T_2.

This benefit can be illustrated by considering our first example query for determining the best exam(s) for every student. The straightforward evaluation computes the student's best grade for every exam he or she has ever taken, i.e.:

$$\begin{array}{l}\sigma_{e.\text{grade}=m}((\text{students } s \bowtie_{s.id=e.sid} \text{exams } e) \ltimes\!\!\!\rtimes \\ \quad (\Gamma_{\emptyset;m:min(e2.\text{grade})}(\sigma_{s.\text{id}=e2.\text{sid}} \text{ exams } e2)))\end{array}$$

The equivalence rule allows to restrict the computation of the best grades to each student – instead of computing it redundantly for each (student, exam)-pair. Thus, the dependent join is executed on the projection of the students' id only, i.e.:

$$\begin{array}{l}\ldots \Pi_{d.id:s.id}((\text{students } s \bowtie_{s.id=e.sid} \text{exams } e) \ltimes\!\!\!\rtimes \\ \quad (\Gamma_{\emptyset;m:min(e2.\text{grade})}(\sigma_{d.\text{id}=e2.\text{sid}} \text{ exams } e2)))\end{array}$$

The application of the "Push-Down"-rule for our example query is shown in Figure 1 where the entire query evaluation plan is graphically depicted. In a way, this constitutes a side-ways information passing from the outer (left) join argument to the inner (right) argument in order to eliminate redundancy in the evaluation. Therefore, it is important to implement the projection in the true, duplicate-eliminating semantics and not in the duplicate-preserving multi-set semantics of SQL.

Even more importantly, we have transformed a generic dependent join into a dependent join of a *set* (i.e., a relation without duplicates). Knowing that D contains no duplicates

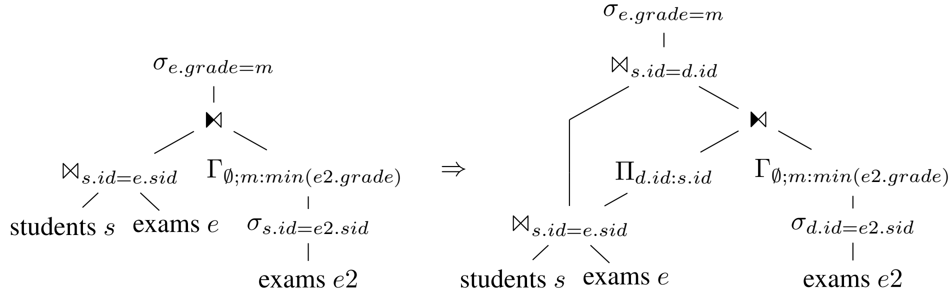


Figure 1: Example Application of Dependent Join "Push-Down"

helps in moving the dependent join further down into the query. In the following we will assume that any relation named D is duplicate free, and in the following equivalences we only consider dependent joins where the left hand side is a set. However, we emphasize that this optimization technique for nested queries does preserve the SQL multi-set semantics. All duplicates – contained in the base relations as well as generated by the query – are retained in the optimized plans; only the set D that constrains the evaluation work of the nested subquery is duplicate free. If duplicates are to be removed (because of a distinct clause in the query) we can further exploit this by pushing duplicate elimination down into the query evaluation plan.

The ultimate goal of our dependent join push-down is to reach a state where the right hand side no longer depends on the left hand side, i.e.,

$$D \ltimes\!\!\!\rtimes T \quad \equiv \quad D \bowtie T \quad \text{if} \quad \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset.$$

In this case we still have to perform a join, but at least we can perform a regular join instead of the highly inefficient dependent join. And, as we will see, we can always reach this state. An even nicer goal would be to reach a state where the resulting regular join can be substituted by existing attributes, eliminating the join altogether. We will discuss that in Section 4.

Having explained the start and the goal of our dependent join push down, we now look at individual operators. For selections, a push-down is very simple:

$$D \ltimes\!\!\!\rtimes \sigma_p(T_2) \quad \equiv \quad \sigma_p(D \ltimes\!\!\!\rtimes T_2).$$

This transformation might look unusual, as we usually want to push selections down, but that is besides the point of our unnesting transformation: We first push the dependent join down as far as possible, until it can either be eliminated completely due to substitution, or until it can be transformed into a regular join. Once all dependent joins have been eliminated we can use the regular techniques like selection push-down and join reordering to re-optimize the transformed query.

Pushing a dependent join down another join is more complex, as potentially both sides

could depend upon the dependent join

$$D \mathbin{⧑} (T_1 \bowtie_p T_2) \equiv \begin{cases} (D \mathbin{⧑} T_1) \bowtie_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ T_1 \bowtie_p (D \mathbin{⧑} T_2) & : \quad \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{⧑} T_1) \bowtie_{p \land \text{natural join } D} (D \mathbin{⧑} T_2) & : \quad \text{otherwise.} \end{cases}$$

If the values provided by the dependent join are only required on one side we push it to the corresponding side, otherwise we wave to replicate it in both sides. Note that this push-down rule is overly pessimistic, we can often simplify the parts below the join (see Section 4), but we stick to the basic push-down for now. If we pushed the dependent join to both sides we have to augment the join predicate such that both sides are matched on the D values. Note that the replication is not a performance penalty relative to the original expression, in both cases T_1 and T_2 are evaluated $|D|$ times.

For *outer joins* we always have to replicate the dependent join if the inner side depends on it, as otherwise we cannot keep track of unmatched tuples from the outer side.

$$D \mathbin{⧑} (T_1 \mathbin{⟕}_p T_2) \equiv \begin{cases} (D \mathbin{⧑} T_1) \mathbin{⟕}_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{⧑} T_1) \mathbin{⟕}_{p \land \text{natural join } D} (D \mathbin{⧑} T_2) & : \quad \text{otherwise.} \end{cases}$$

$$D \mathbin{⧑} (T_1 \mathbin{⟗}_p T_2) \equiv (D \mathbin{⧑} T_1) \mathbin{⟗}_{p \land \text{natural join } D} (D \mathbin{⧑} T_2).$$

Similar for *semi join* and *anti join*:

$$D \mathbin{⧑} (T_1 \ltimes_p T_2) \equiv \begin{cases} (D \mathbin{⧑} T_1) \ltimes_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{⧑} T_1) \ltimes_{p \land \text{natural join } D} (D \mathbin{⧑} T_2) & : \quad \text{otherwise.} \end{cases}$$

$$D \mathbin{⧑} (T_1 \triangleright_p T_2) \equiv \begin{cases} (D \mathbin{⧑} T_1) \triangleright_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{⧑} T_1) \triangleright_{p \land \text{natural join } D} (D \mathbin{⧑} T_2) & : \quad \text{otherwise.} \end{cases}$$

When pushing the dependent join down a *group-by* operator, the group-operator must preserve all attributes produces by the dependent join

$$D \mathbin{⧑} (\Gamma_{A;a:f}(T)) \quad \equiv \quad \Gamma_{A \cup \mathcal{A}(D);a:f}(D \mathbin{⧑} T)$$

Again, this makes use of the fact that D is a set.

The *projection* behaves similar to the group by operator

$$D \mathbin{⧑} (\Pi_A(T)) \quad \equiv \quad \Pi_{A \cup \mathcal{A}(D)}(D \mathbin{⧑} T)$$

The only missing operators are the set operations

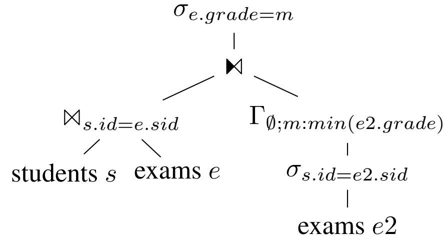


Figure 2: Original Query Q1

$$
\begin{aligned}
D \bowtie (T_1 \cup T_2) &\equiv (D \bowtie T_1) \cup (D \bowtie T_2) \\
D \bowtie (T_1 \cap T_2) &\equiv (D \bowtie T_1) \cap (D \bowtie T_2) \\
D \bowtie (T_1 \setminus T_2) &\equiv (D \bowtie T_1) \setminus (D \bowtie T_2)
\end{aligned}
$$

Using these transformations, each dependent join is either eliminated at some point by substitution, or ends up in front of a base relation, in which case it can be transformed into a non-dependent join. Thus, the dependent join can be eliminated from any query.

One potential concern for this approach could be that D might become very large, as it is the set of all variable bindings for the nested subquery. But fortunately that is not the case. Note that, if the original nested join was $T_1 \bowtie T_2$, then $|D| \leq |T_1|$. Thus, if (after decorrelating the subquery), the top-most join is a hash-join which stores T_1 in a hash table, the memory consumption for that join is at most doubled by computing D. And that is the absolute worst case. If we know that the values from T_1 are duplicate free, for example because they contain a key, we can even avoid materializing D and read the join hash table instead, removing any overhead. On the plus side we have transformed an $O(n^2)$ operation into an (ideally) $O(n)$ operation, which is well worth the memory overhead.

3.3 Optimization of Example Query Q1

As illustrational example, consider the algebraic translation of Query Q1 in Figure 2. It uses a dependent join to compute the nested subquery, and afterwards uses the produced attribute m to check the filter condition.

The subsequent transformations are shown in Figure 3 to 7. First, the top-most dependent join is transformed into a regular join plus a dependent join with the domain of the free

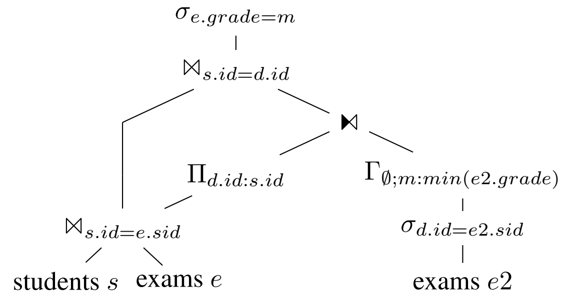


Figure 3: Query Q1, Transformation Step 1

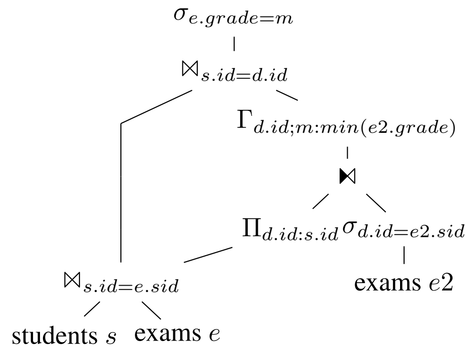


Figure 4: Query Q1, Transformation Step 2

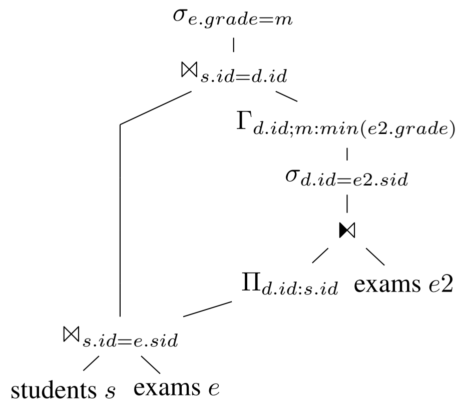


Figure 5: Query Q1, Transformation Step 3

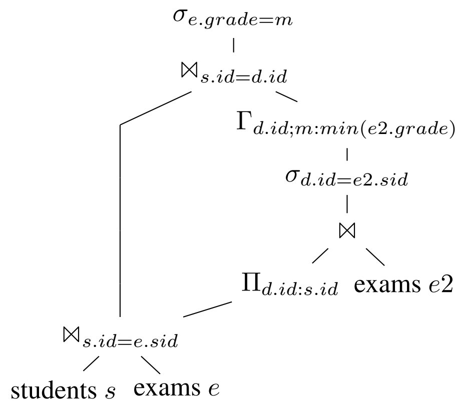


Figure 6: Query Q1, Transformation Step 4

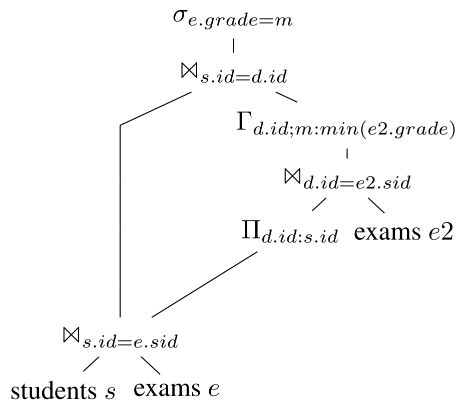


Figure 7: Query Q1, Transformation Step 5 (pushing selections back down)

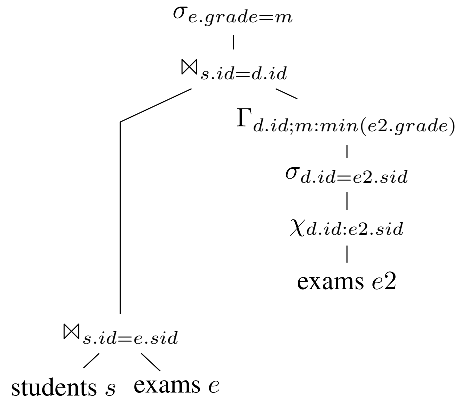


Figure 8: Query Q1, Optional Transformation Step 6 (decoupling both sides)

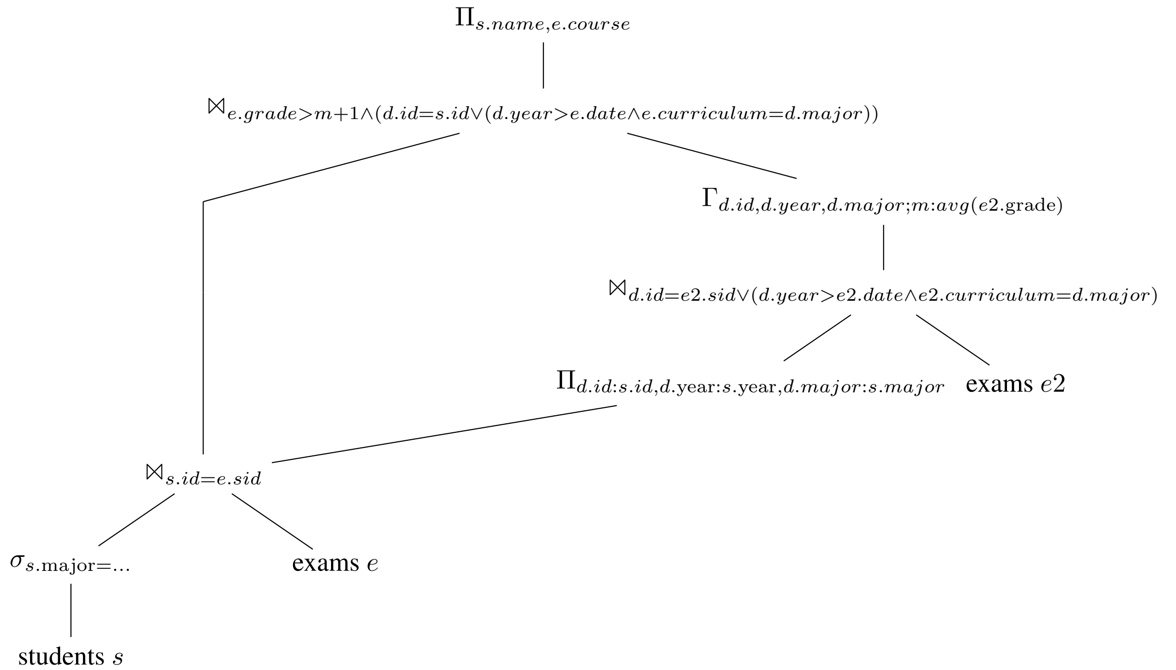


Figure 9: Query Q2, Optimized Form with Sideways Information Passing

variables. In the next step the dependent join is pushed down the group by operator, extending the aggregation attributes as needed. Afterwards, the dependent join is pushed down the selection, ending up in front of a table scan. Now we can transform it into a regular join, as the right hand side is not dependent on the left hand side. Subsequent optimizations will push the predicates down again, introducing a regular join, which results in the plan shown in Figure 7.

In many cases, and also in this example, it is possible to eliminate the join with the domain D altogether: If all attributes of the domain are (equi-)joined with existing attributes, we can instead derive the domain from the existing attributes. In this example, we know that $d.id = e2.sid$ holds after the join, therefore we can replace this by a map operator that substitutes $d.id$ with $e2.sid$, as shown in Figure 8. The nice thing about this substitution is that the parts of the query are completely independent, no trace of the original nesting remains. However, substitution creates a *superset* of the original tuples! At least in general, in this example referential integrity will most likely prevent that. But in general dropping the join with the domain and substituting instead can lead to larger intermediate results, as the filter effect of the join is removed, too. This does not affect the final result, because the filter will still happen at a later stage (at the original dependent join), but it can affect the query runtime. Therefore removing the join is a decision that has to be made by the query optimizer (see Section 4).

Note that some care is needed when trying to remove $\sigma_{d.id=e2.sid}$ after substitution. It would be tempting to simply drop this selection, as $d.id$ is derived from $e2.sid$. But this is only safe if $e2.sid$ is not nullable. If $e2.sid$ can assume NULL values, then the selection must be preserved (and in fact is simply a not-NULL check).

3.4 Optimization of Example Query Q2

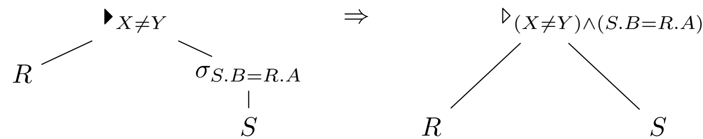
For our motivational query Q2 we illustrate the resulting plan in Figure 9. Note that here D cannot be eliminated, as there is a non-equi join with values from D, which prevents substitution. Here, decoupling the nested subquery evaluation is not possible because the value of $s.year$ is not equi-joinded with $e2.date$. Therefore, the domain of the outer query has to be transferred sideways to the nested query evaluation. However, note that all dependent joins have disappeared and were replaced by efficient regular algebraic operators.

3.5 Anti-Join Example

Let us discuss an example of a dependent anti-join which is used for transforming queries that use an SQL **all**-clause to compare a value against all values derived by a (possibly correlated) subquery. Such a query is formulated below for two abstract relations $R : \{[A, ..., X, ...]\}$ and $S : \{[B, ..., Y, ...]\}$.

```
Q3:  select R.*
     from   R
     where  R.X = all (select S.Y
                       from S
                       where S.B = R.A)
```

Obviously, it is not possible to translate this query into a dependent join; but it is possible to negate the predicate and use a dependent anti-join as shown on the left hand side of the subsequent figure. In the course of the unnesting optimization the dependent anti-join can be transformed into a "normal" anti-join. The resulting query plan is shown on the right-hand side of the subsequent figure.



4 Optimizations

When *simple unnesting* is successful, it completely eliminates any hint of correlations from the query. That is, the resulting algebra expression looks as if the query had been formulated without correlated subqueries. The *general unnesting* case however has to add the projection to compute the domain D, and the join with D, which causes some extra costs. Of course computing D is usually still much preferable to a nested evaluation, as the computation of D and the join with D cause one-time costs, while a nested evaluation results in quadratic runtime. But still, eliminating D completely is tempting, and sometimes possible, as seen in the previous example query Q1 (cf. Figure 8).

In general, we can eliminate D, if we can substitute it with values that already exist in the subtree anyway. This is commonly the case with equi-joins, for example the query contains the expression $D \bowtie_{D.a=R.b} R$, we can learn the possible values of $D.a$ *that can make it to the original dependent join* by inspecting the values of $R.b$. The emphasized part of the statement is important, of course D can contain values that do not exist in R, but these will never find a join partner and will thus never reach the original dependent join. We can therefore ignore them.

To decide about substitution we must first analyze the query tree to find equivalence classes that are induced by the join and filter conditions. For example a filter condition $\sigma_{a=b}$ implies that a and b are in the same equivalence class. We know that in the final result a and b have the same value, we can thus substitute a with b. Computing these equivalence is relatively straight forward. One potential cause for problems would be outer joins, which can cause a and b to not be equal in the example above, but as the top-most join on D is known to be NULL-rejecting this is not an issue here.

After having identified the equivalence classes C, we can decide about a possible substitution as shown below:

$$D \Join T \quad \subseteq \quad \chi_{\mathcal{A}(D):B}(T) \text{ if } \exists B \subseteq \mathcal{A}(T) : \mathcal{A}(D) \equiv_C B.$$

Thus, instead of joining with D, we can extend T (using the map operator) and compute the implied attribute value from D by using the equivalent attributes. Note that this only holds because D is a set. Note further that substitution might increase the size of intermediate results, the relationship between the two formulations is not $=$ but $\subseteq$. This cardinality increase is caused by losing the (potential) pruning power of the join with D. Instead of evaluating the formed correlated subtree with every tuple in T that will find a join partner in D, we evaluate with every tuple from T. The tuples that do not have join partners will (only) be eliminated higher up in the tree, when the original dependent join is executed, but the intermediate results can be larger.

Therefore, substitution only pays off if the join with D is unselective. In Query Q1 that will be the case, therefore it is a good idea to use substitution, but in general the query optimization has to compare the costs of both alternatives and chose the cheaper one. For selective joins, it is better to keep D and thus eliminate tuples as early as possible.

5 Evaluation

Unnesting correlated subqueries can lead to nearly arbitrary gains, as it can transform an $O(n^2)$ into an (ideally) $O(n)$ operation. Demonstrating a factor 100 improvement for example would be easy with a carefully constructed query. But the examples and the resulting factors would all be a bit arbitrary. Instead, we therefore first compare the expressiveness of our technique with that of other systems, and then give some performance numbers for TPC-H. All experiments were run on an Intel i7-3930K with 64GB main memory.

We have implemented the unnesting in our HyPer [KN11] system, and compared it to other approaches. First, we studied the unnesting capabilities of several database systems. Out of the commercial systems, SQL Server seems to have the best unnesting engine, which is to be expected based on the publication [GJ01] of the the MS SQL Server team. Unfortunately the licensing terms for SQL Server 2014 forbid publishing runtime numbers, therefore we will only qualitatively describe the result. We also ran experiments on PostgreSQL 9.1, where we are allowed to publish numbers. As test data for our two example queries we generated 1,000 student and 10,000 exam tuples, i.e., 10 exams per student. This is a small data set, but the effects of unnesting are so extreme that we did not want to increase the data set size. (Roughly speaking, the gains of our method grow quadratic with the size of the relations, so we could demonstrate arbitrary gains by increasing the data set size).

Query 1 is relatively simple to unnest. Our HyPer system unnests the query and uses substitution, which result in a runtime of <1ms. Without unnesting the query takes 51ms on Hyper. SQL Server 2014 also unnests this query. We are not allowed to report runtimes,

but the query plan is reasonable. PostgreSQL however was not able to unnest even this relatively simple case, which results in a runtime of 1,300ms. And this runtime grows very sharply as the data size grows. If, however, the query would have been formulated decorrelated, as formulated as query $Q1'$ in the Introduction, then PostgreSQL could have executed it in 17ms. This demonstrates how important it is to unnest subqueries.

Query 2 is more difficult to unnest, and we are not aware of any system besides our HyPer system that is able to unnest it. HyPer can execute that query in 42ms (408ms without unnesting). SQL Server 2014 is not able to unnest this query and generates an execution plan with nested loop joins. We are not allowed to report the runtime, but obviously one cannot expect good runtime in a plan with large nested loop joins. PostgreSQL needs 12,099ms for the query, again growing sharply with the data size.

The queries in TPC-H are not that difficult to unnest, and all of the large commercial systems are able to unnest them. This is indeed absolutely essential, as shown by the following performance numbers obtained from our HyPer system that allows to "switch on and off" the unnesting. Even at $SF = 1$, the runtime of Query 4 is 7ms with unnesting, and 157,616ms without. This is a difference of several orders of magnitude, and highlights that unnesting is absolutely essential. Other queries are affected, too: For example Query 17 takes 9ms with unnesting, and 4,664ms without. Of course every vendor makes sure that the well known TPC-H queries are correctly unnested in their system, but these numbers highlight that the performance impact is so large, that a system should be able to unnest arbitrary queries, as proposed in this paper.

6 Related Work

The first, seminal paper on optimization of nested subqueries was published by Won Kim [Kim82]. It provides "recipes" (i.e., transformation rules) for unnesting particular nested query patterns which were integrated in many commercial database systems. A good survey of those techniques was given by Jarke and Koch [JK84]. Werner Kiessling [Kie85] addressed the correctness problems of some of the suggested transformations when empty sets are encountered. For many, but not all, cases he was able to correctly formulate effective unnesting transformations. Thus unnesting remained an active field of research, in particular because of the SQL language extensions that orthogonally allowed nested subqueries in all of the select-from-where-clauses. In the Nineties, Seshadri et al. [SPL96] formulated complex query decorrelationa rules. Dayal's paper [Day87] focuses on efficiently evaluating subqueries utilizing outer joins. The outer joins (also extensively used in our transformation rules) can be optimized using transformation/equivalence rules developed by Galindo-Legaria and Rosenthal [GR97].

Oracle's subquery optimizations are described by Bellamkonda et. al. [BAW+09]. They developed a multitude of subquery optimization techniques, auch as unnesting, group-by merging, common subexpression elimination, join predicate pushdown, join factorization, anti-join formulations of set minus and intersection queries, etc. Many of these techniques are applied as preprocessing transformations in a heuristics pattern-matching way – and, as

it appears, are particularly tuned for optimizing "TPC-H-style" queries. In this paper, we developed a generalized approach that tackles the problem in a uniform way and achieves similar optimization effects as the special-purpose pattern-matching approaches – as our "best of breed" TPC-H performance results reveal [LBKN14].

The Tandem approach for unnesting queries is described by Celis and Zeller [CZ97]. At about the same time, in the context of the Microsoft SL Server, Galindo-Legaria and Joshi [GJ01] introduced the apply-operator, that is similar to our bind join, for "algebrizing" SQL queries with nested subqueries. However, their work fell short of being able to transform all possible nesting patterns: "*Achieving optimality and syntax-independence in this class (subqueries that are removed by introducing additional common subexpressions) requires an understanding of the plan space and mechanisms to generate plans of interest, for queries with common subexpressions, which we believe requires additional research*". We are confident that our paper closes this cited "research gap". Graefe's BTW-paper [Gra03] discusses Microsoft SQL Server's approach to evaluate nested queries that could not be unnested – which was needed when unnesting failed.

A particular query pattern, i.e., scalar subqueries in the presence of disjunctions, was optimized by Brantner, May and Moerkotte [BMM07] using an algebra with bypass operators [KMPS94]. Another important special case of unnesting queries involving large fact tables is addressed by Akinde and Böhlen [AB03]. They argue that outer joins are too costly in OLAP environments with very large fact tables and, instead, propose a generalized multidimensional join operator GMDJ.

In the late Nineties, several groups worked on algebra extension to capture nested subqueries. Some of the work was performed in the context of the object-oriented/object relational models [SAB94] that exhibits data nesting and therefore requires flattening and nesting of these structures. One of the first proposals for such an object query algebra was formulated by Cluet and Moerkotte [CM93] and later expanded by Wang, Maier and Shapiro [WMS99]. Cao and Badia [CB05] proposed the explicit use of nested relations to evaluate nested subqueries.

The Virtuoso system [Erl12] tries to address the problem of nested queries at runtime, by evaluating the nested query not once for every outer tuple, but once batch of outer tuples, similar to a block-wise nested loop join, greatly reducing the cost of nested evaluation. During batched execution, each tuple carries a "set number" between operators, maintaining an association between the intermediate results and the vector of inputs of the subquery [Erl14]. While not as powerful as a query-optimizer based solution, this is much more efficient than the standard nested-loop evaluation of nested queries.

Our side-ways information passing optimization to reduce the amount of work incurred by evaluating the nested subquery (tree) resembles the magic set transformations that were described by Seshadri et al [SHP^{+}96] in the context of the IBM DB2 system. Consequently, in our graphics interface (`www.hyper-db.de`) for displaying the query evaluation plans the corresponding operator is called *magic*.

7 Conclusion

Many proposals for unnesting SQL queries have been published in the past. However, so far their implementations fell short of comprehensively cover all possible patterns – as becomes evident by surveying the existing (even recent) literature on optimizing special cases. Also, our analyis of commercial and open source database systems revealed that their optimizations only cover special patterns that (arguably) are the most important use cases for query nesting. In this paper we have developed an algebraic transformation approach that covers all kinds of nested subqueries. The algebraic equivalences allow to completely replace the initially used dependent joins by "regular" join operators. In order to decrease the extent of the nested query evaluation work, side-ways information passing is employed that restricts the independent subquery evaluation plan to those tuple sets that are relevant for the outer query. The approach is fully implemented in our main-memory database system HyPer and can be experimented with via our web interface `www.hyper-db.de` that not only displays the running times of interactive queries but also shows the optimized query evaluation plans. Having integrated the unnesting transformations into our cost-based optimizer ensures that we are never too eager in, for example, decoupling the inner query from the outer query and thereby incurring higher costs – the optimizer will do so only if there is a cost benefit which in many practically relevant cases is dramatic.

Acknowledgements

This work was supported by the German Research Foundation DFG. We acknowldge the helpful comments of the anonymous BTW-reviewers that helped to improve the paper.

References

[AB03] Michael O. Akinde and Michael H. Böhlen. Efficient Computation of Subqueries in Complex OLAP. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 163–174. IEEE Computer Society, 2003.

[BAW+09] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zaït, and Chun Chieh Lin. Enhanced Subquery Optimizations in Oracle. *PVLDB*, 2(2):1366–1377, 2009.

[BMM07] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting Scalar SQL Queries in the Presence of Disjunction. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 46–55, 2007.

[CB05] Bin Cao and Antonio Badia. A Nested Relational Approach to Processing SQL Subqueries. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Con-*

ference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, pages 191–202. ACM, 2005.

[CM93] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[CZ97] Pedro Celis and Hansjörg Zeller. Subquery Elimination: A Complete Unnesting Algorithm for an Extended Relational Algebra. In W. A. Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, page 321. IEEE Computer Society, 1997.

[Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 197–208. Morgan Kaufmann, 1987.

[Erl12] Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[Erl14] Orri Erling. personal communication, 2014.

[GJ01] César A. Galindo-Legaria and Milind Joshi. Orthogonal Optimization of Subqueries and Aggregation. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 571–581. ACM, 2001.

[GR97] César A. Galindo-Legaria and Arnon Rosenthal. Outerjoin Simplification and Reordering for Query Optimization. *ACM Trans. Database Syst.*, 22(1):43–73, 1997.

[Gra03] Goetz Graefe. Executing Nested Queries. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, volume 26 of *LNI*, pages 58–77. GI, 2003.

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

[Kie85] Werner Kießling. On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. In Alain Pirotte and Yannis Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden.*, pages 241–250. Morgan Kaufmann, 1985.

[Kim82] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[KMPS94] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing Disjunctive Queries with Expensive Predicates. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 336–347. ACM Press, 1994.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011.

[LBKN14] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014.

[SAB94] Hennie J. Steenhagen, Peter M. G. Apers, and Henk M. Blanken. Optimization of Nested Queries in a Complex Object Model. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 337–350. Springer, 1994.

[SHP+96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-Based Optimization for Magic: Algebra and Implementation. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 435–446. ACM Press, 1996.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex Query Decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 450–458, 1996.

[WMS99] Quan Wang, David Maier, and Leonard Shapiro. Algebraic unnesting for nested queries. Technical report, Oregon Health & Science University, CSETech. Paper 252, 1999. http://digitalcommons.ohsu.edu/csetech/252.