Unnesting Arbitrary Queries

Thomas Neumann & Alfons Kemper

February 12, 2025

1 Introduction

```sql
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)
```

Listing 1: Q1

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 quadratic execution time.

A rewrite would look like this Here, the evaluation of the subquery no

```sql
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
```

Listing 2: Q1′

longer depends on the values of s, and thus regular joins can be used.

Now consider another query, by 2015, no system can unnests it

```sql
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
               from exams e2
               where s.id = e2.sid or
               (e2.curriculums = s.major and
                s.year > e2.date))
```

Listing 3: Q2

2 Preliminaries

(inner) join:

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

dependent join:

$$T_1 \Join_p T_2 := \{t_1 \circ t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}$$

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 dependent join, $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$ must hold.

Take Q1 as example (sort of). T_1 is

```sql
select s.name, s.id from students
```

T_2 is

```sql
select e.id, e.course
from exams e
where e.grade = (select min(e2.grade)
                 from exams e2
                 where X = e2.sid)
```

So T_1, T_2 here are actually all functions

We use **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)

- **semi join**:

$$T_1 \ltimes T_2 := \{t_1 \mid t_1 \in T_1 \wedge \exists t_2 \in T_2 : p(t_1 \circ t_2)\}$$

- **anti semi join**:

$$T_1 \triangleright_p T_2 := \{t_1 \mid t_1 \in T_1 \wedge \nexists t_2 \in T_2 : p(t_1 \circ t_2)\}$$

- **left outer join**:

$$T_1 \underset{p}{⟕} T_2 := (T_1 \bowtie_p T_2) \cup \{t_1 \circ_{a \in \mathcal{A}(T_2)} (a : null) \mid t_1 \in (T_1 \triangleright_p T_2)\}$$

- **full outer join**:

$$T_1 \underset{p}{⟗} T_2 := (T_1 \underset{p}{⟕} T_2) \cup \{t_2 \circ_{a \in \mathcal{A}(T_1)} (a : null) \mid t_2 \in (T_2 \triangleright_p T_1)\}$$

We define the dependent joins accordingly as $⋉$, $\triangleright'$, $⟕'$, $⟗'$
group by:

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

It groups its input e by A and evaluates one aggregation function. I guess a is the attribute of im f
map:

$$\chi_{a:f}(e) := \{x \circ (a : f(x)) \mid x \in e\}$$

We define the attribute comparison operator $=_A$ as

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

It compares NULL values as equal

3 Unnesting

The algebraic representation of a query with correlated subqueries results in a dependent join

3.1 Simple Unnesting

Consider

```sql
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

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

which is equivalent to

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

3.2 General Unnesting

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.

First,

$$T_1 \bowtie_p T_2 \equiv T_1 \bowtie_{p \wedge T_1 =_{\mathcal{A}(D)} D} (D \bowtie T_2)$$

where

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

In the original expression, we had to evaluate T_2 for every tuple of T_1. In the second expression, we first compute the domain D of all variables bindings, evaluate T_2 only once for every distinct variable binding the result of Π is a set, 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.

Consider the query for determining the worst exam for every student:

$$\sigma_{e.grade=m}((\text{student } s \bowtie_{s.id=e.id} \text{exams } e) \mathbin{⧑} \\ (\Gamma_{\emptyset;m:min(e2.grade)}(\sigma_{s.id=e2.sid}\text{exams } e2)))$$

$(\text{student } s \bowtie_{s.id=e.id} \text{ exams } e) \mathbin{⧑} (\Gamma_{\emptyset;m:min(e2.grade)}(\sigma_{s.id=e2.sid}\text{exams } e2))$ In left side, each student has many exams and grades. Then in the right side, for each student, its worst grade is taken. Therefore it generates students + students grades + student worst grade. Then we need σ to filter out those unrelated grades.

The equivalence rule allows to restrict the computation of the best grades to each student

$$\ldots (\Pi_{d.id,s.id}(\text{students } s \bowtie_{s.id=e.sid} \text{exams } e) \mathbin{⧑} \\ (\Gamma_{\emptyset;m:min(e2.grade)}(\sigma_{d.id=e2.sid}\text{exams } e2)))$$

Here $d.id : s.id$ means we replace $s.id$ with $d.id$.

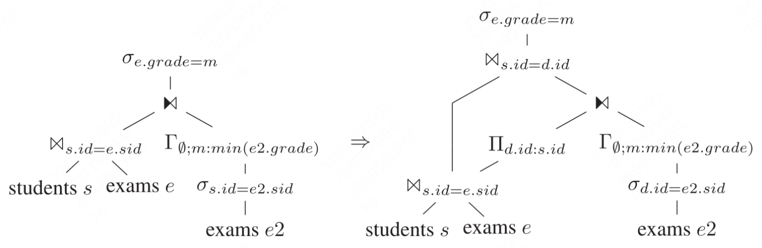

Figure 1: Example Application of Dependent Join "Push-Down"

Knowing that D contains no duplicates 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.

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

$$D \mathbin{⧑} T \equiv D \bowtie T \text{ if } \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset$$

For selections, a push-down is very simple:

$$D \mathbin{⧑} \sigma_p(T_2) \equiv \sigma_p(D \mathbin{⧑} T_2)$$

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.

$$D \Join (T_1 \bowtie_p T_2) = \begin{cases} (D \Join T_1) \bowtie_p T_2 & \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ T_1 \bowtie_p (D \Join T_2) & \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\ (D \Join T_1) \bowtie_{p \wedge \text{natural join } D} (D \Join T_2) & \text{otherwise} \end{cases}$$

If we pushed the dependent join to both sides we have to augment the join predicate s.t. both sides are matched on the D values.

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.

$$D \Join (T_1 ⟕_p T_2) \equiv \begin{cases} (D \Join T_1) ⟕_p T_2 & \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \Join T_1) ⟕_{p \wedge \text{natural join } D} (D \Join T_2) & \text{otherwise} \end{cases}$$

$$D \Join (T_1 ⟗_p T_2) \equiv (D \Join T_1) ⟗_{p \wedge \text{natural join } D} (D \Join T_2)$$

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

$$D \Join (T_1 \ltimes_p T_2) \equiv \begin{cases} (D \Join T_1) \ltimes_p T_2 & \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \Join T_1) \ltimes_{p \wedge \text{natural join } D} (D \Join T_2) & \text{otherwise} \end{cases}$$

$$D \Join (T_1 \triangleright_p T_2) \equiv \begin{cases} (D \Join T_1) \triangleright_p T_2 & \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \Join T_1) \triangleright_{p \wedge \text{natural join } D} (D \Join T_2) & \text{otherwise} \end{cases}$$

group by

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

projection

$$D \Join (\Pi_A(T)) \equiv \Pi_{A \cup \mathcal{A}(D)}(D \Join T)$$

set operation

$$D \Join (T_1 \cup T_2) \equiv (D \Join T_1) \cup (D \Join T_2)$$
$$D \Join (T_1 \cap T_2) \equiv (D \Join T_1) \cap (D \Join T_2)$$
$$D \Join (T_1 \setminus T_2) \equiv (D \Join T_1) \setminus (D \Join T_2)$$

3.3 Optimization of Example Query Q1

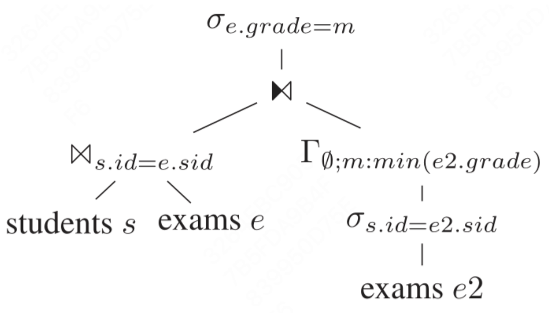

Figure 1: Original Query Q1

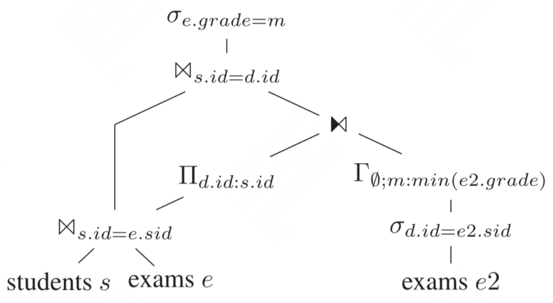

Figure 2: Query Q1, Transformation Step 1

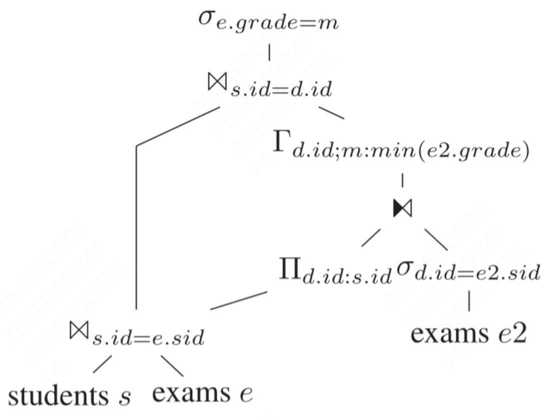

Figure 3: Query Q1, Transformation Step 2

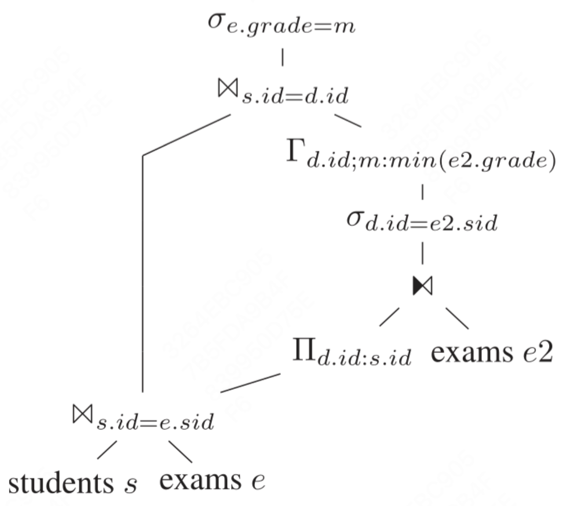

Figure 4: Query Q1, Transformation Step 3

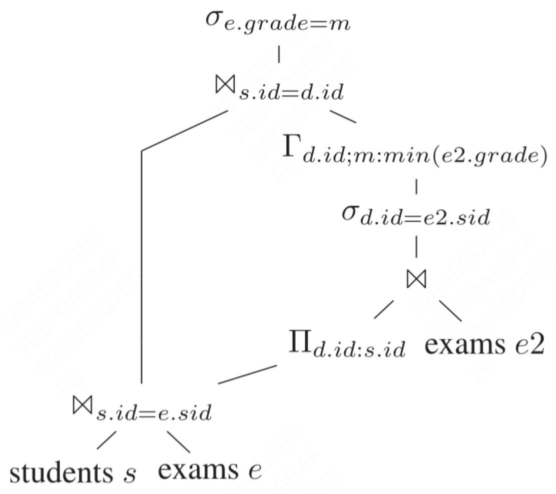

Figure 5: Query Q1, Transformation Step 4

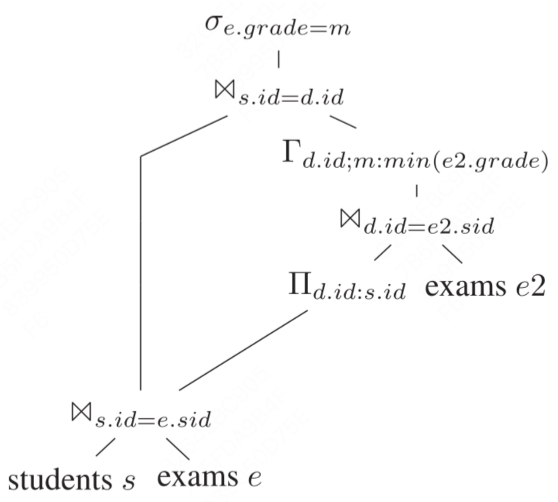

Figure 6: Query Q1, Transformation Step 5

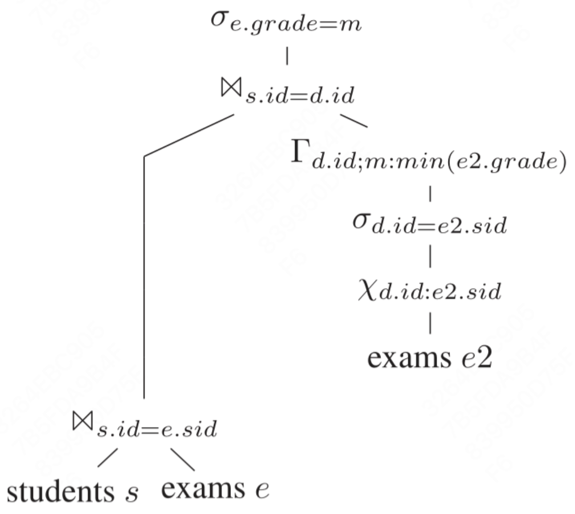

Figure 7: Query Q1, Optional Transformation Step 6 (decoupling both sides)

3.4 Optimization of Example Query Q2

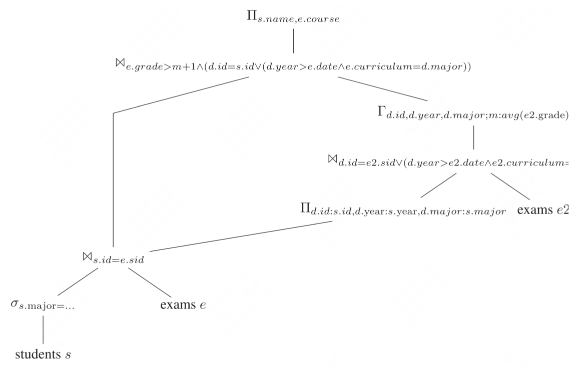
$\Pi_{s.name,e.course}$
$\bowtie_{e.grade>m+1\wedge(d.id=s.id\vee(d.year>e.date\wedge e.curriculum=d.major))}$
$\Gamma_{d.id,d.year,d.major;m:avg(e2.\mathrm{grade})}$
$\bowtie_{d.id=e2.sid\vee(d.year>e2.date\wedge e2.curriculum=}$
$\Pi_{d.id:s.id,d.\mathrm{year}:s.\mathrm{year},d.major:s.major}$
exams $e2$
$\bowtie_{s.id=e.sid}$
$\sigma_{s.\mathrm{major}=\ldots}$
exams e
students s

Figure 8: Query Q2, Optimized From with Sideways Information Passing

3.5 Anti-Join Example

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

4 Optimizations

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.

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:

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

Thus, instead of joining with D, we can extend T and compute the implied attribute value from D by using the equivalent attributes. Note that this only holds because D is a set.

5 Problems

6 References

References