Technische Universität München

Unnesting Arbitrary Queries

Thomas Neumann and Alfons Kemper

Technische Universität München

March 5, 2015

Motivation

Technische Universität München

Often queries are simpler to formulate using subqueries

```
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)
```

- here, subquery depends on outer query (correlated)
- nested loop evaluation, $O(n^2)$
- easy to formulate, very inefficient to execute!

Motivation (2)

Same query without correlated subquery:

```
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
```

- much more efficient to execute, no longer $O(n^2)$
- but not as intuitive as the original query
- a database should *unnest* (i.e., de-correlate) automatically

Motivation (3)

Typically, DBMSs detect and unnest some simple cases. But correlations can be complex:

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

- “difficult” (non-equality, disjunction, etc.)
- we are not aware of any system that could unnest that
- but $O(n^2)$ is a deal breaker, a DBMS must avoid that if possible

Motivation (4)

SQL promised declarative queries

- the user writes what he wants, not what the system should do
- the DBMS finds a good (the best?) evaluation strategy
- failing to unnest queries often leads to catastrophic runtime

We want an generic approach that can handle arbitrary queries

- works on the algebra, on on the SQL representation
- can handle all relational operators

Extended Relational Algebra

We need some extra functionality

$$
\begin{aligned}
\chi_{a:f}(e) &:= \{x \circ (a : f(x)) | x \in e\} \\
T_1 \bowtie_p T_2 &:= \sigma_p(T_1 \times T_2) \\
T_1 ⧑_p T_2 &:= \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\} \\
\Gamma_{A;a:f}(e) &:= \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}
\end{aligned}
$$

Additional notation:

$$
\begin{aligned}
\mathcal{A}(T) &:= \text{the attributes produced by } T \\
\mathcal{F}(T) &:= \text{the free variables of } T
\end{aligned}
$$

Unnesting

Canonical translation turns correlated subqueries into

$$(\textit{outer query}) \ltimes\!\!\!\rtimes_p (\textit{subquery}).$$

- $\ltimes\!\!\!\rtimes$ is a dependent join (evaluates right hand side for every tuple)
- nested loop evaluation, very expensive

The goal of unnesting is to eliminate all dependent joins.

Simple Unnesting

Technische Universität München

Some cases are simple

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

This results in an algebra expression of the form

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

We can unnest by pulling the predicate up, eliminating the dependency.

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

- pull predicates up to eliminate correlations

General Unnesting

Technische Universität München

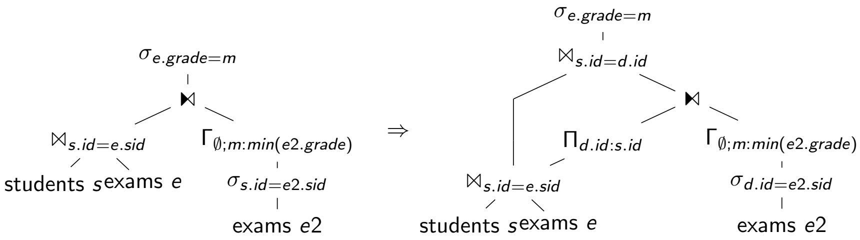
General idea: Evaluate subquery for all possible bindings simultaneously.

$$T_1 \mathbin{\blacktriangleright\!\!\triangleleft}_p T_2 \quad \equiv \quad T_1 \bowtie_{p \wedge T_1 =_{\mathcal{A}(D)} D} \left(D \mathbin{\blacktriangleright\!\!\triangleleft} T_2\right)$$

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

- D provides all possible bindings of free variables
- $|D| \leq |T_1|$
- D is a set (i.e., duplicate free)
- D being a set allow for equivalence that do not hold in general
- allows us to move D until subquery no longer dependent

General Unnesting (2)



Using D might already improve runtime sometimes, but in general is only the first step for full unnesting.

General Unnesting (3)

A dependent join with a set D can be manipulated much more easily. We push D down until the join is no longer dependent:

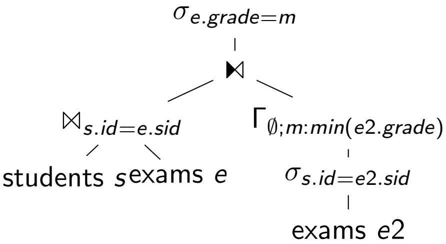
$$D \ltimes\!\!\!\rtimes T \quad \equiv \quad D \bowtie T \quad \text{if} \quad \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset.$$

Push down rules very between operators:

$$\begin{aligned}
D \ltimes\!\!\!\rtimes \sigma_p(T_2) \quad &\equiv \quad \sigma_p(D \ltimes\!\!\!\rtimes T_2) \\
D \ltimes\!\!\!\rtimes (T_1 \bowtie_p T_2) \quad &\equiv \quad \begin{cases} (D \ltimes\!\!\!\rtimes T_1) \bowtie_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ T_1 \bowtie_p (D \ltimes\!\!\!\rtimes T_2) & : \quad \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\ (D \ltimes\!\!\!\rtimes T_1) \bowtie_{p \wedge \text{natural } D} (D \ltimes\!\!\!\rtimes T_2) & : \quad \text{otherwise.} \end{cases} \\
D \ltimes\!\!\!\rtimes (T_1 ⟗_p T_2) \quad &\equiv \quad (D \ltimes\!\!\!\rtimes T_1) ⟗_{p \wedge \text{natural } D} (D \ltimes\!\!\!\rtimes T_2) \\
D \ltimes\!\!\!\rtimes (\Gamma_{A;a:f}(T)) \quad &\equiv \quad \Gamma_{A \cup \mathcal{A}(D);a:f}(D \ltimes\!\!\!\rtimes T) \\
\ldots \quad & \quad\ \ \text{(see the paper)}
\end{aligned}$$

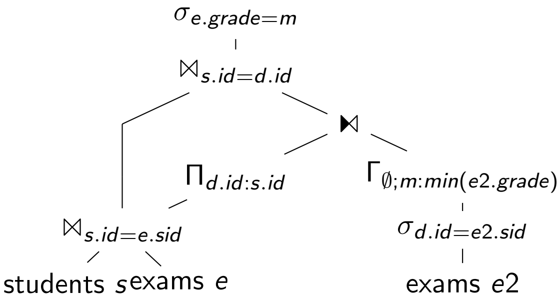
Examples

Technische Universität München



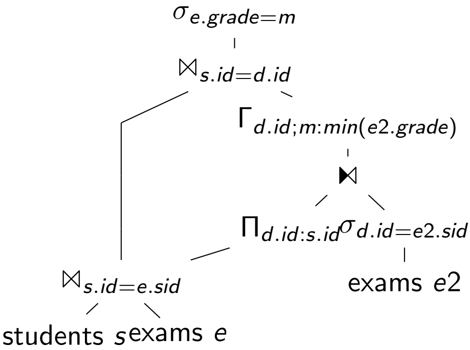
Original Query 1

Examples (2)



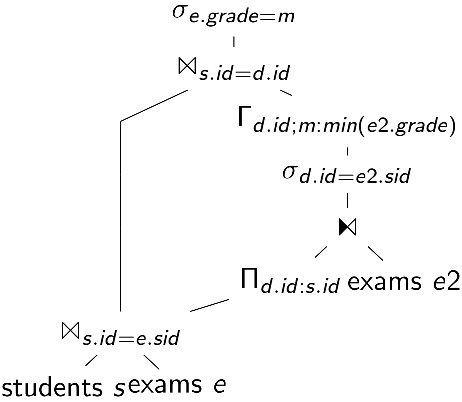
Query 1, Transformation Step 1

Examples (3)



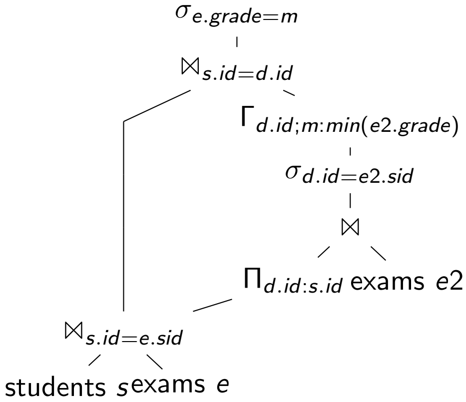
Query 1, Transformation Step 2

Examples (4)



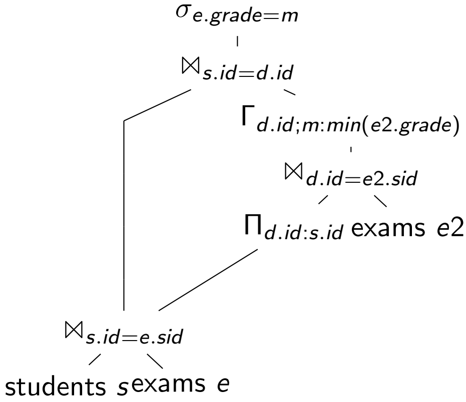
Query 1, Transformation Step 3

Examples (5)



Query 1, Transformation Step 4

Examples (6)



Query 1, Transformation Step 5 (pushing selections back down)

Optimizations

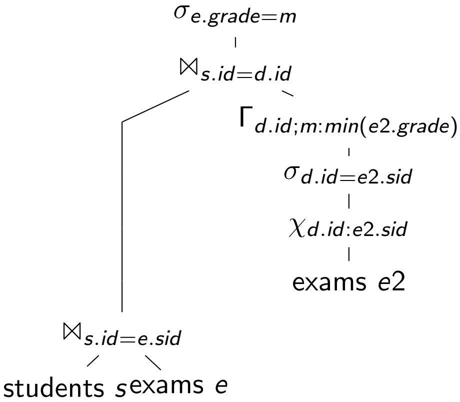
Instead of joining with D, we can often *infer* the attributes from D

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

- "perfect" unnesting, totally independent query parts afterwards
- but: this computes a *superset* of the join with D
- does not matter for correctness (final join will eliminate non-D values), but for performance
- we avoid computing D, but we potential lose pruning power
- a good idea if the join is unselective, otherwise keep D
- cost-base decision

Optimizations (2)

Technische Universität München



Query 1, Optional Transformation Step 6 (decoupling both sides)

Evaluation

- unnesting transforms an $O(n^2)$ into an (ideally) $O(n)$ operation
- arbitrary gains possible

Toy database, 1,000 students, 10,000 exams (i7-3930K)

	Q1	Q2
HyPer	$<$ 1ms	42ms
HyPer without unnesting	51ms	408ms
PostgreSQL 9.1	1,300ms	12,099ms
SQL Sever 2014	can unnest	cannot unnest

We cannot publish absolute runtime for SQL Server 2014, but you can guess from the asymptotics.

Conclusion

Unnesting is essential for good performance

- improves the asymptotics
- can lead to arbitrary gains

We present a generic approach for unnesting

- works on the algebra level, not on the SQL
- exploit set semantics, push down until no longer dependent
- can handle arbitrary queries
- virtually always beneficial, worst case memory overhead factor 2
- could often completely eliminate overhead, but that is a trade off