Unnesting Arbitrary Queries

Thomas Neumann and Alfons Kemper

Technische Universität München

March 5, 2015
Motivation

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:

```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.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

\[
\chi_{a:f}(e) := \{x \circ (a : f(x)) | x \in e\}
\]

\[
T_1 \bowtie_p T_2 := \sigma_p(T_1 \times T_2)
\]

\[
\Gamma_{A;a:f}(e) := \{x \circ (a : f(y)) | x \in \Pi_A(e) \land y = \{z | z \in e \land \forall a \in A : x.a = z.a\}\}
\]

Additional notation:

\[
\mathcal{A}(T) := \text{the attributes produced by } T
\]

\[
\mathcal{F}(T) := \text{the free variables of } T
\]
Unnesting

Canonical translation turns correlated subqueries into

\[(\text{outer query}) \bowtie_p (\text{subquery})\].

- $\bowtie$ 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

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

```
\text{l}_1 \bowtie (\sigma_{\text{l}_1\.okey = \text{l}_2\.okey} (\text{l}_2))
```

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

```
\text{l}_1 \bowtie \text{l}_1\.okey = \text{l}_2\.okey (\text{l}_2)
```

- pull predicates up to eliminate correlations
General Unnesting

General idea: Evaluate subquery for all possible bindings simultaneously.

\[ T_1 \bowtie_p T_2 \equiv T_1 \bowtie_p T_1 =_{A(D)} D \bowtie T_2 \]

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

- \( D \) 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
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:

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

Push down rules very between operators:

$$D \bowtie \sigma_p(T_2) \equiv \sigma_p(D \bowtie T_2)$$

$$D \bowtie (T_1 \bowtie_p T_2) \equiv \begin{cases} (D \bowtie T_1) \bowtie_p T_2 & : \mathcal{F}(T_2) \cap A(D) = \emptyset \smallskip \quad \quad \quad \quad \quad \quad \quad T_1 \bowtie_p (D \bowtie T_2) & : \mathcal{F}(T_1) \cap A(D) = \emptyset \\ (D \bowtie T_1) \bowtie_p \text{natural } D(D \bowtie T_2) & : \text{otherwise.} \end{cases}$$

$$D \bowtie (T_1 \bowtie_p T_2) \equiv (D \bowtie T_1) \bowtie_p \text{natural } D(D \bowtie T_2)$$

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

... (see the paper)
Examples

Original Query 1

\[ \sigma_{e.\text{grade}=m} \]

\[ \land \]

\[ \land_{s.\text{id}=e.\text{sid}} \]

\[ \land \]

\[ \land_{\emptyset; m:\text{min}(e2.\text{grade})} \]

\[ \sigma_{s.\text{id}=e2.\text{sid}} \]

\[ \land \]

exams e2

students sexams e
Examples (2)

Query 1, Transformation Step 1
Examples (3)

\[
\begin{align*}
\sigma_{e\text{.}grade=m} \\
\bowtie s\text{.}id=d\text{.}id \\
\Gamma_{d\text{.}id;m:\min(e\text{.}grade)} \\
\bowtimes_{s\text{.}id=e\text{.}sid} \\
\Pi_{d\text{.}id:s\text{.}id} \sigma_{d\text{.}id=e\text{.}sid} \\
\bowtie s\text{.}id=e\text{.}sid \\
\bowtie_{s\text{.}id=e\text{.}sid} \\
\text{students } s \bowtie \text{exams } e \\
\text{exams } e2
\end{align*}
\]

Query 1, Transformation Step 2
Query 1, Transformation Step 3
 Examples (5)

\[ \sigma_{e.\text{grade}=m} \]
\[ \land_{s.\text{id}=d.\text{id}} \]
\[ \Gamma_{d.\text{id};m:\text{min}(e2.\text{grade})} \]
\[ \sigma_{d.\text{id}=e2.\text{sid}} \]
\[ \land \]
\[ \Pi_{d.\text{id}:s.\text{id}} \text{exams } e2 \]
\[ \land_{s.\text{id}=e.\text{sid}} \]

students s exams e

Query 1, Transformation Step 4
Examples (6)

\[
\sigma_{e.grade=m} \quad \Gamma_{d.id;m: \text{min}(e2.grade)} \quad \Pi_{d.id:s.id \text{ exams } e2} \\
\land_{s.id=d.id} \quad \land_{d.id=e2.sid} \\
\land_{s.id=e.sid} \\
\land_{d.id=e2.sid} \\
\land_{s.id=e.sid} \\
\land_{d.id=e2.sid} \\
\land_{s.id=e.sid} \\
\land_{d.id=e2.sid}
\]

students s \text{ exams } e

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

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

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

- “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)

\[ \sigma_{e.\text{grade}=m} \]
\[ \land s.id=d.id \]
\[ \Gamma_{d.id;m:\min(e2.\text{grade})} \]
\[ \sigma_{d.id=e2.sid} \]
\[ \land \chi_{d.id:e2.sid} \]
\[ \land \text{exams } e2 \]
\[ \land s.id=e.sid \]

students \( s \) exams \( e \)

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)

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HyPer</td>
<td>&lt; 1ms</td>
<td>42ms</td>
</tr>
<tr>
<td>HyPer without unnesting</td>
<td>51ms</td>
<td>408ms</td>
</tr>
<tr>
<td>PostgreSQL 9.1</td>
<td>1,300ms</td>
<td>12,099ms</td>
</tr>
<tr>
<td>SQL Sever 2014</td>
<td>can unnest</td>
<td>cannot unnest</td>
</tr>
</tbody>
</table>

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