Evaluation of Recursive Queries
Part 2: Pushing Selections
Aggregate Operators

The `< ... >` notation in the head indicates grouping; the remaining arguments (Part, in this example) are the GROUP BY fields.

In order to apply such a rule, must have all of Assembly relation available.

Stratification with respect to use of `< ... >` is the usual restriction to deal with this problem; similar to negation.

NumParts(Part, SUM(<Qty>)) :- Assembly(Part, Subpt, Qty).

SELECT A.P, SUM(A.Qty) FROM Assembly A GROUP BY A.P
Don’t let the rule syntax of Datalog fool you: a collection of Datalog rules can be rewritten in SQL syntax, if recursion is allowed.

WITH RECURSIVE Comp(Part, Subpt) AS

UNION
(SELECT A2.Part, C1.Subpt
FROM Assembly A2, Comp C1
WHERE A2.Subpt=C1.Part)

SELECT * FROM Comp C2
Evaluation of Datalog Programs

- **Repeated inferences:** When recursive rules are repeatedly applied in the naïve way, we make the same inferences in several iterations.

- **Unnecessary inferences:** Also, if we just want to find the components of a particular part, say wheel, computing the fixpoint of the Comp program and then selecting tuples with wheel in the first column is wasteful, in that we compute many irrelevant facts.
Avoiding Repeated Inferences

- **Semiaive Fixpoint Evaluation:** Avoid repeated inferences by ensuring that when a rule is applied, at least one of the body facts was generated in the most recent iteration. (Which means this inference could not have been carried out in earlier iterations.)

  - For each recursive table \( P \), use a table \( \delta_P \) to store the \( P \) tuples generated in the previous iteration.
  - Rewrite the program to use the delta tables, and update the delta tables between iterations.

  \[
  \text{Comp}(\text{Part, Subpt}) :- \text{Assembly}(\text{Part, Part2, Qty}), \delta_{\text{Comp}}(\text{Part2, Subpt}).
  \]
Avoiding Unnecessary Inferences

\[
\text{SameLev}(S_1, S_2) \leftarrow \text{Assembly}(P_1, S_1, Q_1), \text{Assembly}(P_1, S_2, Q_2).
\]
\[
\text{SameLev}(S_1, S_2) \leftarrow \text{Assembly}(P_1, S_1, Q_1), \text{SameLev}(P_1, P_2), \text{Assembly}(P_2, S_2, Q_2).
\]

- There is a tuple \((S_1, S_2)\) in \(\text{SameLev}\) if there is a path up from \(S_1\) to some node and down to \(S_2\) with the same number of up and down edges.

```
trike
  +-----+  tire
  |    |    |    
  +-----+    +-----+
  3      1  1  1
  wheel  spoke  rim
  2      1  1
```

```
frame
  +-----+
  |    |
  +-----+
  1    1
  seat  pedal
```

CS 286, UC Berkeley, Spring 2007, R. Ramakrishnan
Avoiding Unnecessary Inferences

- Suppose that we want to find all SameLev tuples with *spoke* in the first column. We should “push” this selection into the fixpoint computation to avoid unnecessary inferences.
- But we can’t just compute SameLev tuples with *spoke* in the first column, because some other SameLev tuples are needed to compute all such tuples:

\[
\text{SameLev}(\text{spoke}, \text{seat}) \leftarrow \text{Assembly}(\text{wheel}, \text{spoke}, 2), \text{SameLev}(\text{wheel}, \text{frame}), \text{Assembly}(\text{frame}, \text{seat}, 1).
\]
“Magic Sets” Idea

Idea: Define a “filter” table that computes all relevant values, and restrict the computation of SameLev to infer only tuples with a relevant value in the first column.

Magic_SL(P1) :- Magic_SL(S1), Assembly(P1,S1,Q1).
Magic_SL(spoke).

SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1), Assembly(P1,S2,Q2).
SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1), SameLev(P1,P2), Assembly(P2,S2,Q2).
The Magic Sets Algorithm

- Generate an “adorned” program
  - Program is rewritten to make the pattern of bound and free arguments in the query explicit; evaluating SameLevel with the first argument bound to a constant is quite different from evaluating it with the second argument bound
  - This step was omitted for simplicity in previous slide
- Add filters of the form “Magic_P” to each rule in the adorned program that defines a predicate P to restrict these rules
- Define new rules to define the filter tables of the form Magic_P
Generating Adorned Rules

- The adorned program for the query pattern SameLev\textsuperscript{bf}, assuming a left-to-right order of rule evaluation:

\begin{align*}
\text{SameLev}^{bf}\ (S1,S2) & \text{ :- Assembly}(P1,S1,Q1),\ \text{Assembly}(P1,S2,Q2). \\
\text{SameLev}^{bf}\ (S1,S2) & \text{ :- Assembly}(P1,S1,Q1), \\
& \quad \text{SameLev}^{bf}\ (P1,P2),\ \text{Assembly}(P2,S2,Q2).
\end{align*}

- An argument of (a given body occurrence of) SameLev is \textit{b} if it appears to the left in the body, or in a \textit{b} arg of the head of the rule.

- Assembly is not adorned because it is an explicitly stored table.
Defining Magic Tables

- After modifying each rule in the adorned program by adding filter “Magic” predicates, a rule for Magic_P is generated from each occurrence O of P in the body of such a rule:
  - Delete everything to the right of O
  - Add the prefix “Magic” and delete the free columns of O
  - Move O, with these changes, into the head of the rule

\[
\text{SameLev}^{bf} (S1,S2) :\text{- Magic}_SL^{bf}(S1), \text{Assembly}(P1,S1,Q1), \\
\text{SameLev}^{bf} (P1,P2), \text{Assembly}(P2,S2,Q2).
\]

\[
\text{Magic}_SL^{bf}(P1) :\text{- Magic}_SL^{bf}(S1), \text{Assembly}(P1,S1,Q1).
\]
Nested Queries in SQL (No Recursion)

```
SELECT E, Sal, Avg, Ecnt
FROM emp(E, Sal, D, J),
     dinfo(D, Avg, Ecnt)
WHERE J = "Sr pgmer"

```

dinfo(D, A, C) AS
SELECT D, AVG(Sal), count(*)
FROM emp
GROUPBY D

“Find senior programmers and their salary, and also average salary and headcount in their depts.”
Example – Datalog and Magic

- Datalog
  
  Einfo(E, Sal, Avg, Ecnt) :- J="Sr pgmer", emp(E, Sal, D, J), dinfo(D, Avg, Ecnt).
  
  dinfo(D, A, C) :- ...

- MAGIC
  
  m_emp^{fff}(J) :- J="Sr pgmer".
  
  m_dinfo^{bff}(D) :- { J = “Sr pgmer” }, m_emp^{fff}(J), emp(E, Sal, D, J).
  
  dinfo^{bff}(D, A, C) :- m_dinfo^{bff}(D), …
Magic

1. **Identifies subqueries**
   
   Idea: Use rules of the form:
   
   If <…> is a (sub)query and also conditions <…> hold,
   
   Then <…> are also subqueries.

2. **Restricts computation**
   
   Idea: Modify the view definition by joining with the table of queries. 
   
   This join acts as a “Filter”.

3. **Classify queries**
   
   Idea: Using “Adornments”, or “Query forms”. All queries of the form
   
   p_{bf}(c, y)? are “MAGIC” tuples m_{p_{bf}}(c).

4. **Magic on subqueries**
   
   Can use magic for q_1 subqueries:
   
   m_{q_1}(y) :- ... p(x, y, z).
   
   q_1(x, y) :- m_{q_1}(x), ...
   
   q_2 subqueries handled some other way.

So, suitable for rule-based optimizer.
Dealing with Subqueries – Other ways

**CORRELATION**

When a subquery is generated, compute all answers, then continue.
  - Not (gasp!) set-oriented.
  - Current DB solution. (DB2 etc.)

**PROLOG**

When a subquery is generated, compute one answer, then continue.
  - May have to “BACKTRACK”.
Example – Recursion, Duplicates

```
SELECT P, S, count(*)
FROM contains(P, S)
GROUP BY [P, S]

Contains(p, s)   AS
   ( SELECT P, S
       FROM subpart(P, S) ) UNION
   ( SELECT P, S
       FROM subpart(P, T),
           contains(T, S) )
```

“Find all subparts of a part along with a count of how often the subpart is used.”
Correlation

\[
\text{SELECT} \quad \text{Ename} \\
\text{FROM} \quad \text{emp e1} \\
\text{WHERE} \quad \text{Job = “Sr pgmer” AND Sal > ( SELECT AVG(e2,Sal) FROM emp e2 WHERE e2.D = e1.D )}
\]

For each senior programmer, the average salary of her/his department is computed.

- Not set-oriented.
- Possible redundancy.
Select Ename from emp, dep_avgsal where Job = "Sr pgmer" and Sal > Asal and emp.D = dep_avgsal.D

-- Set-oriented, no redundancy.
-- But..., irrelevant computation.
Voila! Magic!

msg(D) AS
SELECT DISTINCT D
FROM emp
WHERE Job = “Sr pgmer”

dep_avgsal(D, ASal) AS
SELECT D, AVG(Sal)
FROM msg, emp
WHERE msg.D = emp.D
GROUPY D

SELECT Ename
FROM emp, dep_avgsal
WHERE Job = “Sr pgmer” AND Sal > Asal AND emp.D = dep_avgsal.D
From Datalog to SQL

• Conditions
  • $X + Y > 10$

• Grouping and aggregation

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>toy</td>
<td>20</td>
</tr>
<tr>
<td>Joe</td>
<td>toy</td>
<td>30</td>
</tr>
<tr>
<td>Susan</td>
<td>cs</td>
<td>50</td>
</tr>
<tr>
<td>David</td>
<td>cs</td>
<td>*</td>
</tr>
</tbody>
</table>

$\text{avgsal} = 25$

• Multisets
  • If you don’t remove duplicates. That is a feature!
Example - Conditions

SELECT Ename, Mgr
FROM emp, dept
WHERE Job = "Sr pgmer" AND Sal > 50000 AND emp.D = dept.D

Cast Magic

m_emp^fib(Sal, Job) AS
Job = "Sr Pgmer" AND
Sal > 50000

What really happens?

\[ \sigma_{\text{job="Sr Pgmer"}} \left( \begin{array}{c}
\text{emp} \\
\text{Sal>50000}
\end{array} \right) \otimes_{\text{D=D}} \text{dept} \] "Grounding"
Datalog to SQL: A summary

Conditions
- Magic transformation is followed by some “GROUNDING” steps.

Multisets (Duplicates)

Semantics
- # copies of a tuple = # of derivations

Operationally
- Just skip duplicate checks

Magic
- All “magic” tables are DISTINCT

Groupby, Aggregates
- Must check if restrictions (selections, conditions) can be “pushed down”
- With recursion, may need stratification.
Comparing Magic and Correlation

We must consider three factors:
1. Binding propagation
2. Repeated work (duplicates)
3. Set-Orientation

<table>
<thead>
<tr>
<th>Correlation*</th>
<th>Magic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. √</td>
<td>√</td>
</tr>
<tr>
<td>2. × (√)</td>
<td>√</td>
</tr>
<tr>
<td>3. ×</td>
<td>√</td>
</tr>
</tbody>
</table>
Experiments

Experiments run on DB V2R2 DBMS.

Benchmark DB

<table>
<thead>
<tr>
<th>Table</th>
<th>Tuple Size</th>
<th>#Tuples</th>
<th>#Column</th>
<th>#4k Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>itm</td>
<td>34</td>
<td>170,000</td>
<td>4</td>
<td>1,850</td>
</tr>
<tr>
<td>wkc</td>
<td>28</td>
<td>500</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Itl</td>
<td>78</td>
<td>2,550,000</td>
<td>13</td>
<td>57,980</td>
</tr>
<tr>
<td>itp</td>
<td>43</td>
<td>339,440</td>
<td>14</td>
<td>4,250</td>
</tr>
</tbody>
</table>
## Results

### Experiment 1

Binding propagation, no duplicates, set-orientation not significant.

<table>
<thead>
<tr>
<th>Query</th>
<th>Time</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Correlated</td>
<td>0.40</td>
<td>0.06</td>
</tr>
<tr>
<td>Magic</td>
<td>0.46</td>
<td>0.25</td>
</tr>
</tbody>
</table>

### Experiment 2

Binding set contains duplicates (~100), set-orientation not significant.

<table>
<thead>
<tr>
<th>Query</th>
<th>Time</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Correlated</td>
<td>2.10</td>
<td>0.005</td>
</tr>
<tr>
<td>Magic</td>
<td>0.25</td>
<td>0.069</td>
</tr>
</tbody>
</table>
### Results – Cont.

#### Experiment 3

Binding set has some duplicates, set-orientation is significant. (Bindings on non-index column)

<table>
<thead>
<tr>
<th>Query</th>
<th>Time</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Correlated</td>
<td>513</td>
<td>453</td>
</tr>
<tr>
<td>Magic</td>
<td>55</td>
<td>46</td>
</tr>
</tbody>
</table>

#### Experiment 4

Variant of experiments 3 with more expensive subquery. (10 binding).

<table>
<thead>
<tr>
<th>Query</th>
<th>Time</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Correlated</td>
<td>52.5</td>
<td>22.7</td>
</tr>
<tr>
<td>Magic</td>
<td>8.6</td>
<td>5.2</td>
</tr>
</tbody>
</table>
Conclusions

Magic is:

- Applicable to full SQL.
- Suitable for rule-based optimization.
- Efficient.
- Stable.
- Parallelizable.