Evaluation of Recursive Queries

Part 2: Pushing Selections

Aggregate Operators

```
SELECT A.Part, SUM(A.Qty)
FROM Assembly A
GROUP BY A.Part
```

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

- 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.

Datalog vs. SQL Notation

- 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

(SELECT A1.Part, A1.Subpt FROM Assembly A1)
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

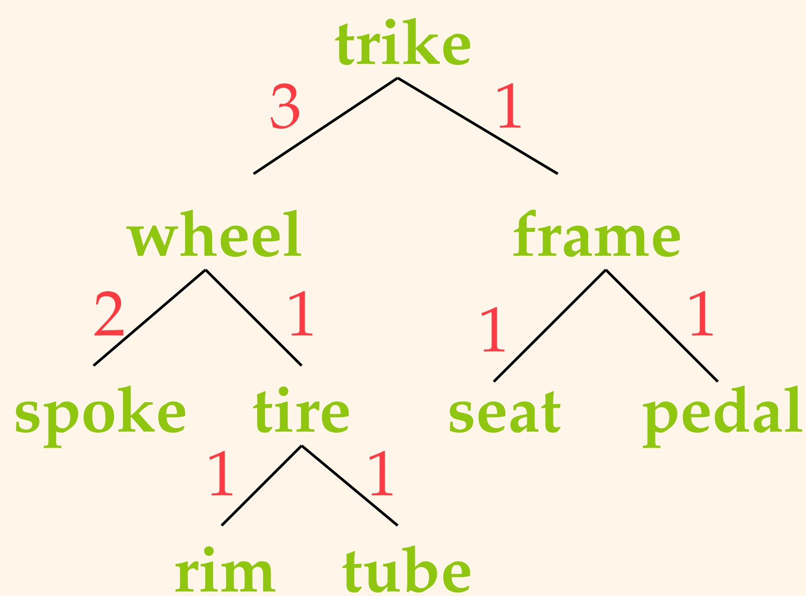
- Seminaive Fixpoint Evaluation: Avoid repeated inferences by ensuring that when a rule is applied, at least one of the body facts was generated in the most recent iteration. (Which means this inference could not have been carried out in earlier iterations.)
  - For each recursive table P, use a table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

Comp(Part, Subpt) :- Assembly(Part, Part2, Qty),
delta_Comp(Part2, Subpt).

Avoiding Unnecessary Inferences

```
SameLev(S1,S2) :- Assembly(P1,S1,Q1), Assembly(P1,S2,Q2).
SameLev(S1,S2) :- Assembly(P1,S1,Q1),
                    SameLev(P1,P2), Assembly(P2,S2,Q2).
```

- There is a tuple (S1,S2) in SameLev if there is a path up from S1 to some node and down to S2 with the same number of up and down edges.



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:

SameLev(spoke,seat) :- Assembly(wheel,spoke,2),
<u>SameLev(wheel,frame)</u>, Assembly(frame,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^{bf}, *assuming a left-to-right order of rule evaluation* :

SameLev^{bf} (S1,S2) :- Assembly(P1,S1,Q1), Assembly(P1,S2,Q2).

SameLev^{bf} (S1,S2) :- Assembly(P1,S1,Q1),

SameLev^{bf} (P1,P2), Assembly(P2,S2,Q2).

- An argument of (a given body occurrence of) SameLev is b if it appears to the left in the body, or in a 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

SameLev^{bf} (S1,S2) :- Magic_SL^{bf}(S1), Assembly(P1,S1,Q1),
SameLev^{bf} (P1,P2), Assembly(P2,S2,Q2).

Magic_SL^{bf}(P1) :- Magic_SL^{bf}(S1), 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^{fffb}(J) :- J="Sr pgmer".

  m_dinfo^{bff}(D) :- { J = "Sr pgmer" }, m_emp^{fffb}(J), emp(E, Sal, D, J).

  $dinfo^{bff}$(D, A, C) :- m_dinfo^{bff}(D), …

Magic

1. <u>Identifies subqueries</u>

   Idea: Use rules of the form:

   If <…> is a (sub)query and also conditions <…> hold,

   Then <…> are also subqueries.

2. <u>Restricts computation</u>

   Idea: Modify the view definition by joining with the table of queries.

   This join acts as a "Filter".

3. <u>Classify queries</u>

   Idea: Using "Adornments", or "Query forms". All queries of the form

   $p^{bf}(c,y)$? are "MAGIC" tuples $m\_p^{bf}(c)$.

4. <u>Magic on subqueries</u> :- …, $p(x, \underline{y}, \underline{z})$, $q_1(\underline{y}, u)$, $q_2(\underline{z}, y)$ …

   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)
GROUPBY       [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

```
OUTER   SELECT    Ename
        FROM      emp e1
        WHERE     Job = “Sr pgmer” AND
                  Sal >
INNER             (      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.

Decorrelation

```
SELECT    Ename
FROM      emp, dep_avgsal
WHERE     Job = "Sr pgmer"        AND
          Sal > Asal              AND
          emp.D = dep_avgsal.D

     dep_avgsal    AS
          SELECT     D, AVG(Sal)
          FROM       emp
          GROUPBY    D
```

- Set-oriented, no redundancy.
- But…, irrelevant computation.

Voila! Magic!

```
SELECT  Ename
FROM    emp, dep_avgsal
WHERE   Job = "Sr pgmer"  AND
        Sal > Asal AND emp.D = dep_avgsal.D
```

```
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
```

From Datalog to SQL

- Conditions
  - X + Y >10
- Grouping and aggregation

John	toy	20
Joe	toy	30
Susan	cs	50
David	cs	*

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^{fbfb}(Sal, Job) AS

Job = “Sr Pgmer” AND

Sal > 50000

What really happens?

$$\left\{ \left(\sigma_{\substack{job=\text{“Sr Pgmer”} \\ Sal>50000}} emp \right) \bowtie_{D=D} dept \right\}$$

“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

	Correlation*	Magic
1.	√	√
2.	× (√)	√
3.	×	√

Experiments

Experiments run on DB V2R2 DBMS.

Benchmark DB

Table	Tuple Size	#Tuples	#Column	#4k Pages
itm	34	170,000	4	1,850
wkc	28	500	10	5
Itl	78	2,550,000	13	57,980
itp	43	339,440	14	4,250

Results

Experiment 1

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

Query	Time	I/O
Original	100	100
Correlated	0.40	0.06
Magic	0.46	0.25

Experiment 2

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

Query	Time	I/O
Original	100	100
Correlated	2.10	0.005
Magic	0.25	0.069

Results – Cont.

Experiment 3

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

10 bindings

Query	Time	I/O
Original	100	100
Correlated	513	453
Magic	55	46

100 bindings

Query	Time	I/O
Original	100	100
Correlated	5136	4526
Magic	111	62

Experiment 4

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

Query	Time	I/O
Original	100	100
Correlated	52.5	22.7
Magic	8.6	5.2

Conclusions

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