Recursive SQL, Deductive Databases, Query Evaluation

Book Chapter of
Ramakrishnan and Gehrke
DBMS Systems, 3rd ed.
Motivation

- Can SQL-92 express queries:
  - Are we running low on any parts needed to build a ZX600 sports car?
  - What is total component and assembly cost to build ZX600 at today's part prices?

- Can we extend the query language to cover such queries?
  - Yes, by adding recursion.
Towards Semantics : Datalog

- SQL queries can be read as follows:

  "If some tuples exist in From tables that satisfy Where conditions, then Select tuple is in answer."

- Datalog is query language with same \textbf{if-then} flavor:
  - \textbf{New}: Answer table can appear in From clause, i.e., be defined recursively.
  - Prolog style syntax is commonly used.
Example

**Assembly instance**

<table>
<thead>
<tr>
<th>part</th>
<th>subpart</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>wheel</td>
<td>3</td>
</tr>
<tr>
<td>trike</td>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>seat</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>pedal</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>spoke</td>
<td>2</td>
</tr>
<tr>
<td>wheel</td>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>
Find components of “trike”.

Can you write relational algebra query to compute answer on the given instance of Assembly?

<table>
<thead>
<tr>
<th>part</th>
<th>subpart</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>wheel</td>
<td>3</td>
</tr>
<tr>
<td>trike</td>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>seat</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>pedal</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>spoke</td>
<td>2</td>
</tr>
<tr>
<td>wheel</td>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- Find components of ‘trike’
- There is no R.A. (or SQL-92) query that computes answer on all Assembly instances.

<table>
<thead>
<tr>
<th>part</th>
<th>subpart</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>wheel</td>
<td>3</td>
</tr>
<tr>
<td>trike</td>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>seat</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>pedal</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>spoke</td>
<td>2</td>
</tr>
<tr>
<td>wheel</td>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>

Assembly instance
Find the components of a trike?

We can write a relational algebra query to compute the answer on the given instance of Assembly.

But there is no R.A. (or SQL-92) query that computes the answer on all Assembly instances.
Problem with R.A. and SQL-92

Intuitively, we must join Assembly with itself to deduce that *trike* contains *spoke* and *tire*.
- Takes us one level down Assembly hierarchy.
- To find components that are one level deeper (e.g., rim), need another join.
- To find all components, need as many joins as there are levels in the given instance!

For any RA expression, we can create an Assembly instance for which some answers are not computed by including more levels than number of joins in expression!
Datalog Query that Does the Job

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

Can read second rule as follows:

“\textbf{For all} values of Part, Subpt and Qty, \textbf{if} there is a tuple (Part, Part2, Qty) in Assembly \textbf{and} a tuple (Part2, Subpt) in Comp, \textbf{then} there must be a tuple (Part, Subpt) in Comp.”
Datalog

Datalog : Relational QL inspired by prolog

Program : a collection of rules

Rule : if RHS exists, must be in LHS result.
Using Rule to Deduce New Tuples

Each rule is a template for making inferences: by assigning constants to variables so that each body “literal” is tuple in corresponding relation, we identify tuple that must be in head relation.

Ex: \( \text{Comp}(\text{Part}, \text{Subpt}) \) :- \( \text{Assembly}(\text{Part}, \text{Subpt}, \text{Qty}) \).

- By setting (Part=trike, Subpt=wheel, Qty=3) in rule, we deduce that tuple <trike,wheel> is in relation Comp.
- This is called an inference using the rule.
- Given a set of tuples, we apply rule by making all possible inferences with tuples in body.
Example

\[
\text{Comp}(\text{Part}, \text{Subpt}) \leftarrow \text{Assembly}(\text{Part}, \text{Subpt}, \text{Qty}). \\
\text{Comp}(\text{Part}, \text{Subpt}) \leftarrow \text{Assembly}(\text{Part}, \text{Part2}, \text{Qty}), \text{Comp}(\text{Part2}, \text{Subpt}).
\]

- For any instance of Assembly, we compute all Comp tuples by repeatedly applying two rules.
- Actually: we can apply Rule 1 just once, then apply Rule 2 repeatedly.
- Rule 1 ~ projection
- Rule2 ~ cross-product with equality join
<table>
<thead>
<tr>
<th>Part</th>
<th>Subpt</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>wheel</td>
<td>3</td>
</tr>
<tr>
<td>trike</td>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>seat</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>pedal</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>spoke</td>
<td>2</td>
</tr>
<tr>
<td>wheel</td>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>

Assembly instance

Comp tuples by applying Rule 2 once

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>spoke</td>
</tr>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
</tbody>
</table>

Comp tuples by applying Rule 2 twice

Comp(Part, Subpt) :-

Assembly(Part, Part2, Qty),

Comp(Part2, Subpt).
Example

For any instance of Assembly, we can compute all Comp tuples by repeatedly applying the two rules. (Actually, we can apply Rule 1 just once, then apply Rule 2 repeatedly.)

<table>
<thead>
<tr>
<th>trike</th>
<th>spoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
</tbody>
</table>

Comp tuples got by applying Rule 2 once

<table>
<thead>
<tr>
<th>trike</th>
<th>spoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
</tbody>
</table>

Comp tuples got by applying Rule 2 twice
Datalog vs. SQL Notation

- A collection of Datalog rules can be rewritten in SQL syntax with recursion

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
Datalog vs. SQL Notation

- Or, modify query to have selection:

```sql
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
  WHERE C2.part = trike.
```
Theoretical Foundations

(least fixpoint semantics – conceptual evaluation strategy a la relational algebra)
Fixpoint

Let $f$ be a function that takes values from domain $D$ and returns values from $D$.

A value $v$ in $D$ is a fixpoint of $f$ if $f(v) = v$. 
Fixpoints

Consider fn \textit{double}+ : applied to a set of integers and returns a set of integers, with domain D set of all sets of integers.

– E.g., \( \textit{double}+\{1,2,5\} = \{2,4,10\} \text{ Union } \{1,2,5\} \)

What are example fixpoints for \textit{double}+?
Fixpoints

- Consider fn \textit{double}+:  
  \textit{double}+({1,2,5})={2,4,10} \text{ Union } {1,2,5}

- Example Fixpoints (input sets):
  - The set of all integers is a fixpoint of \textit{double}+.
  - The set of all even integers is another fixpoint.
  - The set of integer zero is another fixpoint.
Least Fixpoint Semantics for Datalog

- **Least fixpoint** of a function $f$ is a fixpoint $v$ of $f$ such that every other fixpoint of $f$ is larger than or equal to $v$.
- There may be no (one) least fixpoint.
- We could have two minimal fixpoints, neither of which is smaller than other.
- Least fixpoint of double+?
Least Fixpoint Semantics for Datalog

- For Datalog program (without set-difference) as a function that applied to a set of tuples and returns another set of tuples, this function (fortunately!) always has a least fixpoint.
- Datalog Function ~~ defined by relational algebra.
Least Fixpoint Semantics for Datalog

- Comp = PROJECT [1,5] (PROJECT[1,2] (Assembly) UNION (Assembly JOIN[2=1] Comp) )

- with Comp = function (Comp, Assembly) defined by RA expression.

- Least Fixpoint ~ Is instance of Comp that satisfies this query (query answer).
Least Fixpoint Semantics for Datalog

- The **least fixpoint** of a function $f$ is a fixpoint $v$ of $f$ such that every other fixpoint of $f$ is smaller than or equal to $v$.
- Eg., Big depends on Small table.
- In general, there may be no least fixpoint (we could have two minimal fixpoints, neither of which is smaller than the other).
- If we think of a Datalog program as a function that is applied to a set of tuples and returns another set of tuples, this function (fortunately!) always has a least fixpoint.
Unsafe/Safe Datalog Program

- If one unbound variable, then program is unsafe:

  Price-Parts (Part, Price) :-
  Assembly(Part, Subpart, Qty), Qty>2.

  Infinite number of different values for price would all make the rule correct.

- If least model of program is not finite, then program is unsafe.

- So all variables in head of rule must also appear in body (range-restricted).
Negation/Set-Difference

What’s the problem below?

Big(Part) :- Assembly(Part, Subpt, Qty), Qty > 2, not Small(Part).
Small(Part) :- Assembly(Part, Subpt, Qty), not Big(Part).

- If rules contain not,
- then there may not be a least fixpoint.
Negation

Big(\text{Part}) := \text{Assembly}(\text{Part}, \text{Subpt}, \text{Qty}), \text{Qty} > 2, \text{not Small}(\text{Part}).
Small(\text{Part}) := \text{Assembly}(\text{Part}, \text{Subpt}, \text{Qty}), \text{not Big}(\text{Part}).

- One least fixpoint?
- Consider Assembly instance
  - What is intuitive answer?
    - \text{trike} is the only part that has 3 or more copies of some subpart. Intuitively, it should be in Big()!
    - If we apply Rule 1 first, we have Big(\text{trike}).
    - If we apply Rule 2 first, we have Small (\text{trike}).
    - Which one is right answer?
Negation

- If rules contain **not**, then there may not be a least fixpoint.
- Order of applying rules determines answer because:
  - Addition of tuples into one output relation may disallow inference of other tuples
- Need method to choose intended fixpoint.

```
Big(Part) :- Assembly(Part, Subpt, Qty), Qty > 2, not Small(Part).
Small(Part) :- Assembly(Part, Subpt, Qty), not Big(Part).
```
If rules contain not there may not be a least fixpoint. Consider the Assembly instance; trike is the only part that has 3 or more copies of some subpart. Intuitively, it should be in Big, and it will be if we apply Rule 1 first.

– But we have Small(trike) if Rule 2 is applied first!
– There are two minimal fixpoints for this program: Big is empty in one, and contains trike in the other (and all other parts are in Small in both fixpoints).

Need a way to choose the intended fixpoint.
Not in Body?

- range-restricted program:
  every variable in head of rule appears in some positive (non-negated) relation occurrence in body.
Stratification

- T **depends on** S if some rule with T in the head contains S or (recursively) some predicate that depends on S, in the body.
- Example: Big depends on Small.
- **Stratified program:** If T depends on **not** S, then S cannot depend on T (or **not** T).
Stratification

- If a program is stratified, tables in program can be partitioned into strata (fully order dependencies):
  - **Stratum 0**: All database tables.
  - **Stratum I**: Tables defined in terms of tables in Stratum I and lower strata.

If T depends on **not** S,
S is in lower stratum than T.
Or, table in stratum I depends negatively only on tables in stratum I-1.
Stratification

- Question:
  Is Big/Small program stratified?

- Big/Small: Mutually recursive tables
Fixpoint Semantics for Stratified Programs

- Semantics of stratified program given by one of its minimal fixpoints.

- This fixpoint identified by operational definition:
  - Stratum 0 tables are fixed
  - First compute least fixpoint of all tables in Stratum 1.
  - Then, compute least fixpoint of tables in Stratum 2.
  - Then, compute least fixpoint of tables in Stratum 3, and so on, stratum-by-stratum.
Fixpoint Semantics for Stratified Programs

- This evaluation strategy is guaranteed to find one minimal fixpoint (but several may exist).

- RA : corresponds to range-restricted stratified datalog.

- SQL3 requires stratified programs.
The < ... > notation in head indicates grouping; remaining arguments (Part, in this example) are GROUP BY fields.

In order to apply such rule, must have all of Assembly relation available. (not on partial computed relation).

Stratification with respect to use of < ... > is restriction to deal with this problem; similar to negation.
Query Optimization
Evaluation of Datalog Programs

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

- **Unnecessary inferences:**
  - If we just want to find components of a particular part, say `wheel`, then first computing general fixpoint of Comp program and then at end selecting tuples with `wheel` in the first column is wasteful. This would compute many irrelevant facts.
Query Optimization #1.
Avoiding Repeated Inferences

- **Semi-naive Fixpoint Evaluation:**
- Ensure that when rule is applied, at least one of body facts used was generated in most recent iteration.
- Such new inference could not have been carried out in earlier iterations.
Avoiding Repeated Inferences

**Idea:** For each recursive table $P$, use table $\text{delta}_P$ to store $P$ tuples generated in previous iteration.

- 1. Rewrite program to use delta tables
- 2. Update delta tables between iterations.

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

Comp(Part, Subpt) :- Assembly(Part, Part2, Qty),
delta_Comp(Part2, Subpt).
```
Query Optimization #2.
Avoiding Unnecessary Inferences

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
Where C2.part = trike.
Avoiding Unnecessary Inferences

Non-recursive program:

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

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

Semantics?
Avoiding Unnecessary Inferences

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

Tuple \((S_1, S_2)\) in \(\text{SameLev}\) if there is path up from \(S_1\) to some node and down to \(S_2\) with same number of up and down edges.
Avoiding Unnecessary Inferences

- Want all SameLev tuples with *spoke* in first column.
- **Intuition**: “Push” this selection into fixpoint computation.
- How do that?

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

\[
\text{SameLev}(\text{spoke},S2) :-
\text{Assembly}(P1,\text{spoke},Q1),
\text{SameLev}(P1=\text{spoke}? ,P2),
\text{Assembly}(P2,S2,Q2).
\]
Avoiding Unnecessary Inferences

**Intuition:** “Push” this selection with **spoke** into fixpoint computation.

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

\[
\text{SameLev}(\text{spoke},S2) :-
\text{Assembly}(P1,\text{spoke},Q1),
\text{SameLev}(\text{spoke},P2),
\text{Assembly}(P2,S2,Q2).
\]

\[
\text{SameLev}(\text{spoke},\text{seat}) :-
\text{Assembly}(\text{wheel},\text{spoke},2),
\text{SameLev}(\text{wheel},\text{frame}),
\text{Assembly}(\text{frame},\text{seat},1).
\]

- Other SameLev tuples are needed to compute all such tuples with **spoke**, e.g. **wheel**
“Magic Sets” Idea

1. Define a “filter” table that computes all relevant values
2. Restrict computation of SameLev to infer only tuples with relevant value in first column.
Intuition

- **Relevant** values: contains all tuples m for which we have to compute all same-level tuples with m in first column to answer query.

- **Relevant** values: compute all Same-Level tuples whose first field contains value on path from spoke up to root.

- We call it Magic-SameLevel (Magic-SL)
“Magic Sets” Idea

- **Idea:** Define “filter” table that computes all relevant values
- Make Magic table as Magic-SameLevel.

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

- Collect all parents of `spoke`.
“Magic Sets” Idea

**Idea:** Use “filter” table to restrict the computation of SameLev.

\[
\text{Magic}_{-}\text{SL}(P1) \ :- \ \text{Magic}_{-}\text{SL}(S1), \ \text{Assembly}(P1,S1,Q1).
\]

\[
\text{Magic}(\text{spoke}).
\]

\[
\text{SameLev}(S1,S2) \ :- \ \text{Magic}_{-}\text{SL}(S1), \ \text{Assembly}(P1,S1,Q1), \ \text{Assembly}(P1,S2,Q2).
\]

\[
\text{SameLev}(S1,S2) \ :- \ \text{Magic}_{-}\text{SL}(S1), \ \text{Assembly}(P1,S1,Q1), \ \text{SameLev}(P1,P2), \ \text{Assembly}(P2,S2,Q2).
\]
“Magic Sets” Idea

- **Idea:** Define “filter” table that computes all relevant values, and restrict the computation of SameLev correspondingly.

\[
\text{Magic}_\text{SL}(P1) \leftarrow \text{Magic}_\text{SL}(S1), \text{Assembly}(P1,S1,Q1).
\]

\[
\text{SameLev}(S1,S2) \leftarrow \text{Magic}_\text{SL}(S1), \text{Assembly}(P1,S1,Q1), \text{Assembly}(P1,S2,Q2).
\]

\[
\text{SameLev}(S1,S2) \leftarrow \text{Magic}_\text{SL}(S1), \text{Assembly}(P1,S1,Q1), \text{SameLev}(P1,P2), \text{Assembly}(P2,S2,Q2).
\]
The Magic Sets Algorithm

1. Generate an “adorned” program
   - Program is rewritten to make pattern of **bound and free arguments** in query explicit

2. Add magic filters of form “Magic_P”
   - for each rule in adorned program add a Magic condition to body that acts as filter on set of tuples generated  (**predicate P** to restrict these rules)

3. Define new rules to define **filter tables**
   - Define new rules to define **filter tables** of form Magic_P
Step 1: Generating Adorned Rules

Adorned program for query pattern SameLev_{bf}, assuming right-to-left order of rule evaluation:

\[
\begin{align*}
\text{SameLev}_{bf} (S_1, S_2) & :\text{ Assembly}(P_1, S_1, Q_1), \text{ Assembly}(P_1, S_2, Q_2), \\
\text{SameLev}_{bf} (S_1, S_2) & :\text{ Assembly}(P_1, S_1, Q_1), \\
\text{SameLev}_{bf} (P_1, P_2) & , \text{ Assembly}(P_2, S_2, Q_2).
\end{align*}
\]

- Argument of (a given body occurrence of) \text{SameLev} is:
  - \text{b} if it appears to the left in body,
  - or if it is a \text{b} argument of head of rule,
  - Otherwise it is \text{free}.

- Assembly not adorned because explicitly stored table.
Step 1: Generating Adorned Rules

Adorned program for query pattern SameLev_{bf}, assuming right-to-left order of rule evaluation:

\[
\text{SameLev}_{bf} (S_1, S_2) :\text{Assembly} (P_1, S_1, Q_1), \text{Assembly} (P_1, S_2, Q_2) .
\]

- Argument of (a given body occurrence of) SameLev is:
  - \( b \) if it appears to the left in body,
  - or if it is a \( b \) argument of head of rule,
  - Otherwise it is \text{free}.

- Assembly not adorned because explicitly stored table.
Step 2: Add Magic Filters

- For every rule in adorned program add a ‘magic filter’ predicate

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

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

- Filter predicate: copy of head of rule, Magic prefix, and delete free variable
Step 3: Defining Magic Tables

Rule for Magic_P is generated from each occurrence of recursive P in body of rule:
– Delete everything to right of P
– Add prefix “Magic” and delete free columns of P
– Move P, with these changes, into head of rule
Step 3: Defining Magic Tables

- Rule for Magic_P is generated from each occurrence O of recursive P in body of rule:
  - Delete everything to right of P

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

- Add prefix “Magic” and delete free columns of P

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

- Move P, with these changes, into head of rule

\[
\text{Magic}_\text{SL}(P1) :\text{ Magic}_\text{SL}(S1), \text{ Assembly}(P1,S1,Q1).
\]
Step 3: Defining Magic Tables

Rule for Magic_P is generated from each occurrence of P in body of such rule:

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

\[
\text{Magic_SL}(P1) :- \text{Magic_SL}(S1), \text{Assembly}(P1,S1,Q1).
\]
Summary

- Adding recursion extends relational algebra and SQL-92 in a fundamental way.
- Recursion included in SQL:1999.
- Semantics based on iterative fixpoint evaluation.
- Programs with negation are restricted to being stratified to ensure semantics is intuitive and unambiguous.
- Evaluation must avoid repeated and unnecessary inferences.
  - “Semi-naive” fixpoint evaluation
  - “Magic Sets” query transformation