Database Theory: Datalog, Views

CS 645
Mar 8, 2006
Coming lectures

• TODAY:
  – Adding recursion: datalog
  – Summary of Containment Complexity
  – Views
Expressive Power of FO

- Let $I = \{R(x,y)\}$ represent a graph
- Query $\text{path}(x,y) =$
  - all $x,y$ such that there is path from $x$ to $y$

- Theorem: $\text{path}(x,y)$ cannot be expressed in FO.
Datalog Programs

• A Datalog program is a collection of rules.
• In a program, subgoals can be either
  1. EDB = Extensional Database = stored table.
  2. IDB = Intensional Database = computed table.
• No EDB in heads.
• (Recall: in a CQ, subgoals are always EDBs)
Non-recursive rules

Graph: \( R(x,y) \)

\[
\begin{align*}
P(x,y) & :\ - R(x,u), R(u,v), R(v,y) \\
A(x,y) & :\ - P(x,u), P(u,y)
\end{align*}
\]

Can “unfold” it into:

\[
\begin{align*}
A(x,y) & :\ - R(x,u), R(u,v), R(v,w), R(w,m), R(m,n), R(n,y)
\end{align*}
\]
Example: Datalog Program

• Using EDB $\text{Sells(bar, beer, price)}$ and $\text{Beers(name, manf)}$,
• find the manufacturers of beers Joe doesn’t sell.

$\text{JoeSells(b)} \leftarrow \text{Sells(’Joes Bar’, b, p)}$
$\text{Answer(m)} \leftarrow \text{Beers(b,m)} \& \neg \text{JoeSells(b)}$
Evaluating Datalog Programs

• As long as there is no recursion, we can pick an order to evaluate the IDB predicates, so that all the predicates in the body of its rules have already been evaluated.

• If an IDB predicate has more than one rule, each rule contributes tuples to its relation.
Recursive example

Two forms of transitive closure:

Graph: $R(x,y)$

Path($x,y$) :- $R(x,y)$
Path($x,y$) :- Path($x,u$), $R(u,y)$

Path($x,y$) :- $R(x,y)$
Path($x,y$) :- Path($x,u$), Path($u,y$)
Recursive Example

- **EDB:** \( \text{Par}(c,p) = p \) is a parent of \( c \).
- **Generalized cousins:** people with common ancestors one or more generations back:

\[
\text{Sib}(x,y) :- \text{Par}(x,p), \text{Par}(y,p), x \neq y \\
\text{Cousin}(x,y) :- \text{Sib}(x,y) \\
\text{Cousin}(x,y) :- \text{Par}(x,x_p), \text{Par}(y,y_p), \text{Cousin}(x_p,y_p)
\]
Definition of Recursion

- Form a **dependency graph** whose nodes = IDB predicates.
- Arc $X \rightarrow Y$ if and only if there is a rule with $X$ in the head and $Y$ in the body.
- Cycle = recursion; no cycle = no recursion.
Example: Dependency Graphs

- **Recursive**
  - `Sib(x, y) :- Par(x, p), Par(y, p), x ≠ y`
  - `Cousin(x, y) :- Sib(x, y)`
  - `Cousin(x, y) :- Par(x, x_p), Par(y, y_p), Cousin(x_p, y_p)`

- **Nonrecursive**
  - `JoeSells(b) <- Sells('Joes Bar', b, p)`
  - `Answer(m) <- Beers(b, m) & ¬JoeSells(b)`
Evaluating Recursive Rules

- The following works when there is no negation:
  1. Start by assuming all IDB relations are empty.
  2. Repeatedly evaluate the rules using the EDB and the previous IDB, to get a new IDB.
  3. End when no change to IDB.

“Fixed Point”
The “Naïve” Evaluation Algorithm

Start:
IDB = 0

Apply rules to IDB, EDB

Change to IDB?

yes

no
done

Start:
IDB = 0

Apply rules to IDB, EDB

Change to IDB?

yes

no
done
Example: Evaluation of Cousin

- We’ll proceed in rounds to infer Sib facts (red) and Cousin facts (green).
- Remember the rules:

- $\text{Sib}(x,y) \leftarrow \text{Par}(x,p) \& \text{Par}(y,p) \& x \neq y$
- $\text{Cousin}(x,y) \leftarrow \text{Sib}(x,y)$
- $\text{Cousin}(x,y) \leftarrow \text{Par}(x,xp) \& \text{Par}(y,yp) \& \text{Cousin}(xp,yp)$
Seminaive Evaluation

• Since the EDB never changes, on each round we only get new IDB tuples if we use at least one IDB tuple that was obtained on the previous round.

• Saves work; lets us avoid rediscovering most known facts.
  - A fact could still be derived in a second way.
Par Data: Parent Above Child

Sibling

Cousin

Round 1
Round 2
Round 3
Round 4
Datalog

(Recursion, no negation)

• There are three equivalent meanings for a datalog rule
  - least fixed point
  - (unique) minimal model
  - set of facts derivable from EDBs
Recursion in SQL

Graph: \( R(x,y) \)

\[
\text{Path}(x,y) :: R(x,y) \\
\text{Path}(x,y) :: \text{Path}(x,u), R(u,y)
\]

WITH RECURSIVE Path(x,y) AS

( SELECT R1.x, R1.y FROM R as R1)

UNION

( SELECT P1.x, R2.y

  FROM R as R2, Path as P1

  WHERE P1.y = R2.x )

SELECT * FROM Path
## Variants of Datalog

<table>
<thead>
<tr>
<th></th>
<th>without recursion</th>
<th>with recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>without ¬</td>
<td>Non-recursive Datalog</td>
<td>Datalog</td>
</tr>
<tr>
<td></td>
<td>= UCQ</td>
<td></td>
</tr>
<tr>
<td>with ¬</td>
<td>Non-recursive Datalog ¬</td>
<td>Datalog ¬</td>
</tr>
<tr>
<td></td>
<td>= FO</td>
<td></td>
</tr>
</tbody>
</table>
Query language classes

- FO queries
- Recursive Queries
- Conjunctive Queries

Algebra
- RA

Logic
- (safe) RC
- UCQ^-

SQL
- SFW +
- UNION
- EXCEPT

Conjunctive Queries
- RA: σ,π,×
- single datalog rule

Datalog
- nr-Datalog^-
- Datalog^-
Containment for Datalog

• **Theorem** Containment for datalog programs is undecidable.
# Summary of complexity of containment

<table>
<thead>
<tr>
<th></th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQ</td>
<td>NP-complete</td>
</tr>
<tr>
<td>UCQ</td>
<td>NP-complete</td>
</tr>
<tr>
<td>nr-Datalog (no (\neg))</td>
<td>(\Pi^p_2)</td>
</tr>
<tr>
<td>CQ(\neg)</td>
<td>(\Pi^p_2)</td>
</tr>
<tr>
<td>CQ(&lt;)</td>
<td>(\Pi^p_2)</td>
</tr>
<tr>
<td>FO (nr-Datalog)</td>
<td>undecidable</td>
</tr>
<tr>
<td>Datalog</td>
<td>undecidable</td>
</tr>
</tbody>
</table>

same expressive power
Non-recursive Datalog

• Union of Conjunctive Queries = UCQ
  – Containment is decidable, and NP-complete

• Non-recursive Datalog (no ¬)
  – Is equivalent to UCQ
  – Hence containment is decidable here too
  – Is it still NP-complete ?
Non-recursive Datalog

• A non-recursive datalog:

\[
\begin{align*}
T_1(x,y) & : R(x,u), R(u,y) \\
T_2(x,y) & : T_1(x,u), T_1(u,y) \\
& \quad \cdots \\
T_n(x,y) & : T_{n-1}(x,u), T_{n-1}(u,y) \\
Answer(x,y) & : T_n(x,y)
\end{align*}
\]

• Its unfolding as a CQ:

\[
Answer(x,y) : R(x,u_1), R(u_1, u_2), R(u_2, u_3), \ldots R(u_m, y)
\]

• How big is this query?
Views
Views

• A **view** is a relation defined by a query.
• The query defining the view is called the **view definition**
• For example:

```
CREATE VIEW Developers AS
   SELECT name, project
   FROM Employee
   WHERE department = "Development"
```

SQL

```
CREATE VIEW Developers AS
   SELECT name, project
   FROM Employee
   WHERE department = "Development"
```

CQ

```
v1(y,z) :- P(x,y) & P(y,z)
```
Virtual and Materialized Views

• A view may be:
  – **virtual**: the view relation is defined, but not computed or stored.
    • Computed only on-demand – slow at runtime
    • Always up to date
  – **materialized**: the view relation is computed and stored in system.
    • Pre-computed offline – fast at runtime
    • May have stale data
Virtual view example

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)

CREATE VIEW Seattle-view AS

SELECT buyer, seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer

We have a new virtual table:
Seattle-view(buyer, seller, product, store)
We can use the view in a query as we would any other relation:

```
SELECT name, store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
     Product.category = "shoes"
```
Querying a virtual view

SELECT  name, Seattle-view.store
FROM    Seattle-view, Product
WHERE   Seattle-view.product = Product.name  AND
        Product.category = “shoes”

“View expansion”

SELECT  name, Purchase.store
FROM    Person, Purchase, Product
WHERE   Person.city = “Seattle”    AND
        Person.name = Purchase.buyer AND
        Purchase.product = Product.name  AND
        Product.category = “shoes”
Views and query minimization

- Users usually don’t write non-minimal queries
- However, non-minimal queries arise when using views intensively
- Good motivation for query minimization
CREATE VIEW HappyBoaters

   SELECT DISTINCT E1.name, E1.manager
   FROM Employee E1, Employee E2
   WHERE E1.manager = E2.name
           and E1.boater='YES'
           and E2.boater='YES'

This query is minimal
Query Minimization for Views

Now compute the Very-Happy-Boaters

```
SELECT DISTINCT  H1.name
FROM HappyBoaters H1, HappyBoaters H2
WHERE H1.manager = H2.name
```

This query is also minimal

What happens in SQL when we run a query on a view?
Query Minimization for Views

View Expansion

```
SELECT DISTINCT  E1.name
FROM Employee E1, Employee E2, Employee E3, Employee E4
WHERE E1.manager = E2.name and E1.boater = 'YES' and E2.boater = 'YES'
      and E3.manager = E4.name and E3.boater = 'YES' and E4.boater = 'YES'
      and E1.manager = E3.name
```

This query is no longer minimal!

E1 -> E2 -> E3 -> E4

E2 is redundant
The great utility of views

• Data independence
• Efficient query processing
  – materializing certain results can improve query execution
  – esp. data warehousing
• Controlling access
  – Grant access to views only to filter data
• Data integration
  – Combine data sources using views
View-related issues

1. View selection
   • which views to materialize, given workload

2. View maintenance
   • when base relations change, views need to be refreshed.

3. Updating virtual views
   • can users update relations that don’t exist?

4. Answering queries using views
   • when only views are available, what queries over base relations are answerable?
Issues in View Materialization

• What views should we materialize? What indexes should we build?
• Given a query and a set of materialized views, can we use the materialized views to answer the query?
• When do we refresh materialized views to make them consistent with the underlying tables? (And how can we do this incrementally?)
View Maintenance

• Two steps:
  ▪ **Propagate:** Compute changes to view when data changes.
  ▪ **Refresh:** Apply changes to the materialized view table.

• **Maintenance policy:** Controls when we do refresh.
  ▪ **Immediate:** As part of the transaction that modifies the underlying data tables. (+ Materialized view is always consistent; - updates are slowed)
  ▪ **Deferred:** Some time later, in a separate transaction. (- View becomes inconsistent; + can scale to maintain many views without slowing updates)
Deferred Maintenance

• Three flavors:
  - **Lazy**: Delay refresh until next query on view; then refresh before answering the query.
  - **Periodic (Snapshot)**: Refresh periodically. Queries possibly answered using outdated version of view tuples. Widely used, especially for asynchronous replication in distributed databases, and for warehouse applications.
  - **Event-based**: E.g., Refresh after a fixed number of updates to underlying data tables.
Updating Views

How can I insert a tuple into a table that doesn’t exist?

Employee(ssn, name, department, project, salary)

CREATE VIEW Developers AS
   SELECT name, project
   FROM Employee
   WHERE department = “Development”

If we make the following insertion:

   INSERT INTO Developers VALUES(“Joe”, “Optimizer”)

It becomes:

   INSERT INTO Employee(ssn, name, department, project, salary)
Non-Updatable Views

Person(name, city)
Purchase(buyer, seller, product, store)

```
CREATE VIEW City-Store AS
    SELECT Person.city, Purchase.store
    FROM     Person, Purchase
    WHERE  Person.name = Purchase.buyer
```

How can we add the following tuple to the view?

(“Seattle”, “Nine West”)

We don’t know the name of the person who made the purchase; cannot set to NULL (why?)
Troublesome examples

CREATE VIEW OldEmployees AS
   SELECT name, age
   FROM Employee
   WHERE age > 30

INSERT INTO OldEmployees VALUES ("Joe", 28)

If this tuple is inserted into view, it won’t appear! Allowed by default in SQL.
## Ambiguous updates

<table>
<thead>
<tr>
<th>Name</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>fac</td>
</tr>
<tr>
<td>Bob</td>
<td>fac</td>
</tr>
<tr>
<td>Bob</td>
<td>cvs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>group</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac</td>
<td>foo.txt</td>
</tr>
<tr>
<td>fac</td>
<td>bar.txt</td>
</tr>
<tr>
<td>cvs</td>
<td>foo.txt</td>
</tr>
</tbody>
</table>

Join

**view**

<table>
<thead>
<tr>
<th>Name</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>foo.txt</td>
</tr>
<tr>
<td>Alice</td>
<td>bar.txt</td>
</tr>
<tr>
<td>Bob</td>
<td>foo.txt</td>
</tr>
<tr>
<td>Bob</td>
<td>bar.txt</td>
</tr>
</tbody>
</table>

Delete ("Alice", "foo.txt")
Updating views in practice

- Updates on views highly constrained:
  - SQL-92: updates only allowed on single-table views with projection, selection, no aggregates.
  - SQL-99: takes into account primary keys; updates on multiple table views may be allowed.
  - SQL-99: distinguishes between updatable and insertable views.
Data integration

• A data integration system provides a uniform interface to multiple autonomous, heterogeneous data sources.
  – enterprise integration
  – integrating sources on the WWW
  – integrating scientific databases
• Uniform interface is offered by constructing a mediated schema over sources (views)
• Query answering means expressing query over sources using only views.
Data integration
Answering queries using views

Available views:

\[ v1(y,z) :- R(x,y), R(y,z) \]
\[ v2(x,z) :- R(x,y), R(y,z) \]

Query we want to answer:

\[ q(w) :- R(0,u), R(u,v), R(v,w) \]

“Great-grand parents of person 0 ”

\[ A(s) :- v2(0,t), v1(t,s) \]
Basic strategy

Given views $V_1, \ldots, V_n$ and query $Q$,

- Guess a rewriting
  - i.e. query $A$ which refers to views
- Expand $A$ using view definitions
- Check equivalence of $A$ and $Q$
Finding a Rewriting

**Theorem** Given views $V_1, \ldots, V_n$ and query $Q$, the problem whether $Q$ has an equivalent rewriting in terms of $V_1, \ldots, V_n$ is NP complete.
Rewritings

For language L

• Q’ is a maximally-contained rewriting of Q if:
  - Q’ ⊆ Q
  - no other rewriting contains Q’
Certain Answers

**Definition.** Given $V_1, \ldots, V_n$, their answers $A_1, \ldots, A_n$ and a query $Q$, a tuple $t$ is a *certain* tuple for $Q$ iff for every database instance $D$:

- **CWD (Closed World Assumption)**
  
  \[
  \text{if } A_1 = V_1(D) \text{ and } \ldots \text{ and } A_n = V_n(D) \text{ then } t \in Q(D)
  \]

- **OWD (Open World Assumption)**
  
  \[
  \text{if } A_1 \subseteq V_1(D) \text{ and } \ldots \text{ and } A_n \subseteq V_n(D) \text{ then } t \in Q(D)
  \]