CSEP 544: Lecture 08

Datalog
Announcements

- Homework 4 due tomorrow
- Homework 5 is posted
- Reading assignment due next Monday
- Reading assignment due on March 11:
  - C-stores (long), NoSQL (medium), blog (short)
Outline for Tday

• Optimistic Concurrency Control

• Datalog
Review

- Schedule
- Serializable/conflict-serializable
- 2PL
- Strict 2PL
- Phantoms

SQL isolation levels:
- Read uncommitted
- Read committed
- Repeatable reads
- Serializable
Optimistic Concurrency Control Mechanisms

• Pessimistic:
  – Locks

• Optimistic
  – Timestamp based: basic, multiversion
  – Validation
  – Snapshot isolation: a variant of both
Timestamps

• Each transaction receives a unique timestamp \( TS(T) \)

Could be:

• The system’s clock
• A unique counter, incremented by the scheduler
Timestamps

Main invariant:

The timestamp order defines the serialization order of the transaction

Will generate a schedule that is view-equivalent to a serial schedule, and recoverable
Main Idea

• For any two conflicting actions, ensure that their order is the serialized order:
Check WT, RW, WW conflicts

• \( w_U(X) \ldots r_T(X) \)
• \( r_U(X) \ldots w_T(X) \)
• \( w_U(X) \ldots w_T(X) \)

When \( T \) requests \( r_T(X) \), need to check \( TS(U) \leq TS(T) \)
Timestamps

With each element X, associate

• $RT(X) =$ the highest timestamp of any transaction U that read X
• $WT(X) =$ the highest timestamp of any transaction U that wrote X
• $C(X) =$ the commit bit: true when transaction with highest timestamp that wrote X committed

If element = page, then these are associated with each page X in the buffer pool
Simplified Timestamp-based Scheduling

Start discussion with transactions that do not abort

Transaction wants to read element X
   If WT(X) > TS(T) then ROLLBACK
   Else READ and update RT(X) to larger of TS(T) or RT(X)

Transaction wants to write element X
   If RT(X) > TS(T) then ROLLBACK
   Else if WT(X) > TS(T) ignore write & continue (Thomas Write Rule)
   Otherwise, WRITE and update WT(X) = TS(T)
Details

Read too late:

• T wants to read X, and $WT(X) > TS(T)$

START(T) … START(U) … $w_U(X)$ … $r_T(X)$

Need to rollback T!
Write too late:

- T wants to write X, and $RT(X) > TS(T)$

```
START(T) … START(U) … $r_U(X)$ … $w_T(X)$
```

Need to rollback T!
Details

Write too late, but we can still handle it:

- T wants to write X, and
  \[ RT(X) \leq TS(T) \quad \text{but} \quad WT(X) > TS(T) \]

START(T) … START(V) … \( w_V(X) \) … \( w_T(X) \)

Don’t write X at all!
(Thomas’ rule)
View-Serializability

• By using Thomas’ rule we do not obtain a conflict-serializable schedule

• But we obtain a view-serializable schedule
Ensuring Recoverable Schedules

• Review:
  – Schedule that *avoids* *cascading aborts*

• Use the commit bit $C(X)$ to keep track if the transaction that last wrote $X$ has committed
Ensuring Recoverable Schedules

Read dirty data:
• T wants to read X, and $WT(X) < TS(T)$
• Seems OK, but…

If $C(X)=$false, $T$ needs to wait for it to become true
Ensuring Recoverable Schedules

Thomas’ rule needs to be revised:

- T wants to write X, and \( WT(X) > TS(T) \)
- Seems OK not to write at all, but …

If \( C(X) = \text{false} \), T needs to wait for it to become true
Transaction wants to READ element X
    If WT(X) > TS(T) then ROLLBACK
    Else If C(X) = false, then WAIT
    Else READ and update RT(X) to larger of TS(T) or RT(X)

Transaction wants to WRITE element X
    If RT(X) > TS(T) then ROLLBACK
    Else if WT(X) > TS(T)
        Then If C(X) = false then WAIT
            else IGNORE write (Thomas Write Rule)
    Otherwise, WRITE, and update WT(X)=TS(T), C(X)=false
Summary of Timestamp-based Scheduling

• View-serializable

• Recoverable
  – Even avoids cascading aborts

• Does NOT handle phantoms
Multiversion Timestamp

• When transaction T requests r(X) but WT(X) > TS(T), then T must rollback

• Idea: keep multiple versions of X: X\textsubscript{t}, X\textsubscript{t-1}, X\textsubscript{t-2}, \ldots

\[ TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots \]

• Let T read an older version, with appropriate timestamp
Details

• When $w_T(X)$ occurs,
create a new version, denoted $X_t$ where $t = TS(T)$

• When $r_T(X)$ occurs,
find most recent version $X_t$ such that $t < TS(T)$

Notes:
– $WT(X_t) = t$ and it never changes
– $RT(X_t)$ must still be maintained to check legality of writes

• Can delete $X_t$ if we have a later version $X_{t1}$ and all active transactions $T$ have $TS(T) > t1$
Example (in class)

\[
X_3 \quad X_9 \quad X_{12} \quad X_{18}
\]

R6(X) -- what happens?
W14(X) – what happens?
R15(X) – what happens?
W5(X) – what happens?

When can we delete \(X_3\)?
Summary of Timestamp-based Scheduling

• View-serializable

• Recoverable
  – Even avoids cascading aborts

• DOES handle phantoms
Concurrency Control by Validation

• Each transaction T defines a **read set** RS(T) and a **write set** WS(T)
• Each transaction proceeds in three phases:
  – Read all elements in RS(T). Time = START(T)
  – Validate (may need to rollback). Time = VAL(T)
  – Write all elements in WS(T). Time = FIN(T)

Main invariant: the serialization order is VAL(T)
Avoid $r_T(X)$ - $w_U(X)$ Conflicts

IF $RS(T) \cap WS(U)$ and $FIN(U) > START(T)$
   (U has validated and U has not finished before T begun)
Then ROLLBACK(T)
Avoid $w_T(X) - w_U(X)$ Conflicts

$U$: Read phase  Validate  Write phase

$T$: Read phase  Validate  Write phase ?

IF $WS(T) \cap WS(U)$ and $FIN(U) > VAL(T)$
(U has validated and U has not finished before T validates)
Then ROLLBACK(T)
Snapshot Isolation

• Another optimistic concurrency control method

• Very efficient, and very popular
  – Oracle, Postgres, SQL Server 2005

WARNING: Not serializable, yet ORACLE uses it even for SERIALIZABLE transactions!
Snapshot Isolation Rules

• Each transactions receives a timestamp $TS(T)$

• $Tnx$ sees the snapshot at time $TS(T)$ of database

• When $T$ commits, updated pages written to disk

• Write/write conflicts are resolved by the “first committer wins” rule
Snapshot Isolation (Details)

• Multiversion concurrency control:
  – Versions of X: \(X_{t1}, X_{t2}, X_{t3}, \ldots\)
• When T reads X, return \(X_{TS(T)}\).
• When T writes X (to avoid lost update):
  • If latest version of X is \(TS(T)\) then proceed
  • If \(C(X) = \text{true}\) then \text{abort}
  • If \(C(X) = \text{false}\) then \text{wait}
What Works and What Not

• No dirty reads (Why ?)
• No inconsistent reads (Why ?)
• No lost updates (“first committer wins”)

• Moreover: no reads are ever delayed

• However: read-write conflicts not caught!
Write Skew

T1:
READ(X);
if X >= 50
then Y = -50; WRITE(Y)
COMMIT

T2:
READ(Y);
if Y >= 50
then X = -50; WRITE(X)
COMMIT

In our notation:
R₁(X), R₂(Y), W₁(Y), W₂(X), C₁, C₂

Starting with X=50, Y=50, we end with X=-50, Y=-50.
Non-serializable !!!
Write Skews Can Be Serious

- ACIDland had two viceroys, Delta and Rho
- Budget had two registers: ta**xes**, and spend**y**ng
- They had HIGH taxes and LOW spending…

Delta:
```
READ(X);
if X= ‘HIGH’
  then { Y= ‘HIGH’;
       WRITE(Y) }
COMMIT
```

Rho:
```
READ(Y);
if Y= ‘LOW’
  then {X= ‘LOW’;
       WRITE(X) }
COMMIT
```

… and they ran a deficit ever since.
Tradeoffs

• Pessimistic Concurrency Control (Locks):
  – Great when there are many conflicts
  – Poor when there are few conflicts

• Optimistic Concurrency Control (Timestamps):
  – Poor when there are many conflicts (rollbacks)
  – Great when there are few conflicts

• Compromise
  – READ ONLY transactions → timestamps
  – READ/WRITE transactions → locks
Commercial Systems

• **DB2**: Strict 2PL

• **SQL Server**:
  – Strict 2PL for standard 4 levels of isolation
  – Multiversion concurrency control for snapshot isolation

• **PostgreSQL, Oracle**
  – Snapshot isolation even for SERIALIZABLE
  – Postgres introduced novel, serializable scheduler in postgres 9.1
Datalog
Queries + Iterations

• For 30 years: a backwater of SQL

• Today: huge interest due to *big data analytics*

• Very few commercial datalog systems (e.g. Logicblox)

• Much larger number of hand-crafted applications (e.g. iteration + map-reduce)
Datalog

Review (from Lecture 2)
- Fact
- Rule
- Head and body of a rule
- Existential variable
- Head variable
Review

Facts

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules

Facts = tuples in the database
Rules = queries
Review

Facts

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules

Q1(y) :- Movie(x,y,z), z='1940'.

Facts = tuples in the database
Rules = queries
Facts

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules

Q1(y) :- Movie(x, y, z), z='1940'.
Q2(f, l) :- Actor(z, f, l), Casts(z, x), Movie(x, y, '1940').

Facts = tuples in the database
Rules = queries
Facts

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules

Q1(y) :- Movie(x, y, z), z='1940'.

Q2(f, l) :- Actor(z, f, l), Casts(z, x),
          Movie(x, y, '1940').

Q3(f, l) :- Actor(z, f, l), Casts(z, x1), Movie(x1, y1, 1910),
          Casts(z, x2), Movie(x2, y2, 1940).

Facts = tuples in the database
Rules = queries
Review

Facts

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules

Q1(y) :- Movie(x,y,z), z='1940'.
Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
          Casts(z,x2), Movie(x2,y2,1940).

Facts = tuples in the database
Rules = queries
Extensional Database Predicates = EDB
Intensional Database Predicates = IDB
Review

Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,’1940’).

f, l = head variables
x,y,z= existential variables
Simple datalog programs

R encodes a graph

R =

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

What does it compute?

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
Simple datalog programs

R encodes a graph

\[
R = \\
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\hline
\end{array}
\]

Initially: T is empty.

T(x, y) :- R(x, y)
T(x, y) :- R(x, z), T(z, y)

What does it compute?
Simple datalog programs

R encodes a graph

R =

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Initially:
T is empty.

First iteration:

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

T =

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Simple datalog programs

R encodes a graph

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

What does it compute?

Initially:
T is empty.

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\hline
\end{array}
\]

First iteration:
T =

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\hline
\end{array}
\]

Second iteration:
T =

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
2 & 2 \\
1 & 3 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
\hline
\end{array}
\]
Simple datalog programs

R encodes a graph

R =

1 2
2 1
2 3
1 4
3 4
4 5

First iteration:
T =

1 2
2 1
2 3
1 4
3 4
4 5

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Second iteration:
T =

1 2
2 1
2 3
1 4
3 4
4 5

Done

What does it compute?

Initially:
T is empty.

Third iteration:
T =

1 2
2 1
2 3
1 4
3 4
4 5

1 1
2 2
1 3
2 4
1 5
3 5
2 5
Simple datalog programs

R encodes a graph

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

Initially:
T is empty.

First iteration:
T =

Second iteration:
T =

Third iteration:
T =

Done

Discovered twice
Discovered 3 times!
Simple datalog programs

R encodes a graph

```
1 -> 4
2 -> 1
2 -> 3
1 -> 4
3 -> 4
4 -> 5
```

Alternative ways to compute TC:

- **Right linear**
  - \( T(x,y) :\) R(x,y)
  - \( T(x,y) :\) R(x,z), T(z,y)

- **Left linear**
  - \( T(x,y) :\) R(x,y)
  - \( T(x,y) :\) T(x,z), R(z,y)

- **Non-linear**
  - \( T(x,y) :\) R(x,y)
  - \( T(x,y) :\) T(x,z), T(z,y)

Discuss pros/cons in class
Simple datalog programs

R encodes a colored graph

R encodes a colored graph

Compute TC (ignoring color):

Compute pairs of nodes connected by the same color (e.g. (2,4))

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Green</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Blue</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Yellow</td>
<td>5</td>
</tr>
</tbody>
</table>
Simple datalog programs

R encodes a colored graph

R=  

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Green</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Blue</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Yellow</td>
<td>5</td>
</tr>
</tbody>
</table>

Compute TC (ignoring color):

\[ T(x,y) :- R(x,c,y) \]
\[ T(x,y) :- R(x,c,z), T(z,y) \]

Compute pairs of nodes connected by the same color (e.g. (2,4))
Simple datalog programs

R encodes a colored graph

R=

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Blue</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Green</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Blue</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Yellow</td>
<td>5</td>
</tr>
</tbody>
</table>

Compute TC (ignoring color):

T(x,y) :- R(x,c,y)
T(x,y) :- R(x,c,z), T(z,y)

Compute pairs of nodes connected by the same color (e.g. (2,4))

T(x,c,y) :- R(x,c,y)
T(x,c,y) :- R(x,c,z), T(z,c,y)
Answer(x,y) :- T(x,c,y)
Simple datalog programs

R, G, B encodes a 3-colored graph

What does this program compute in general?

\[
\begin{align*}
S(x,y) & :- B(x,y) \\
S(x,y) & :- T(x,z), B(z,y) \\
T(x,y) & :- S(x,z), R(z,y) \\
T(x,y) & :- S(x,z), G(z,y) \\
Answer(x,y) & :- T(x,y)
\end{align*}
\]

\[
\begin{array}{cc}
R = & \begin{array}{cc}
1 & 2 \\
3 & 4 \\
4 & 5 \\
\end{array} \\
G = & \begin{array}{cc}
2 & 3 \\
\end{array} \\
B = & \begin{array}{cc}
2 & 1 \\
1 & 4 \\
\end{array}
\end{array}
\]
Simple datalog programs

R, G, B encodes a 3-colored graph

What does this program compute in general?

```
S(x,y) :- B(x,y)
S(x,y) :- T(x,z), B(z,y)
T(x,y) :- S(x,z), R(z,y)
T(x,y) :- S(x,z), G(z,y)
Answer(x,y) :- T(x,y)
```

Answer: it computes pairs of nodes connected by a path spelling out these regular expressions:

- \( S = (B.(R \text{ or } G))^*.B \)
- \( T = (B.(R \text{ or } G))^+ \)
Syntax of Datalog Programs

The schema consists of two sets of relations:

- **Extensional Database (EDB):** $R_1, R_2, \ldots$
- **Intentional Database (IDB):** $P_1, P_2, \ldots$

A datalog program $P$ has the form:

$$P: \quad P_i(\text{x}_{i1}, \text{x}_{i2}, \ldots) : \text{body}_1$$
$$P_{i2}(\text{x}_{i1}, \text{x}_{i2}, \ldots) : \text{body}_2$$

$$\ldots$$

- Each head predicate $P_i$ is an IDB
- Each body is a conjunction of IDB and/or EDB predicates
- See lecture 2

Note: no negation (yet)! Recursion OK.
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} :- \text{body}_1 \]
\[ P_{i2} :- \text{body}_2 \]
\[ \text{....} \]
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} : \text{body}_1 \]
\[ P_{i2} : \text{body}_2 \]
\[ \ldots \]

\[ P_1 : \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_2 : \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]
\[ \ldots \]

Group by IDB predicate
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} : - \text{body}_1 \]
\[ P_{i2} : - \text{body}_2 \]
\[ ... \]

\[ P_1 : - \text{body}_{11} \cup \text{body}_{12} \cup ... \]
\[ P_2 : - \text{body}_{21} \cup \text{body}_{22} \cup ... \]
\[ ... \]

Group by IDB predicate

\[ P_1 : - \text{SPJU}_1 \]
\[ P_2 : - \text{SPJU}_2 \]
\[ ... \]

Each rule is a Select-Project-Join-Union query
Naïve Datalog Evaluation Algorithm

Datalog program:

$$P_i \frac{\rightarrow}{\text{body}_i}$$

Group by IDB predicate

$$P_1 \frac{\rightarrow}{\text{body}_{11} \cup \text{body}_{12} \cup \ldots}$$

Each rule is a Select-Project-Join-Union query

$$P_2 \frac{\rightarrow}{\text{body}_{21} \cup \text{body}_{22} \cup \ldots}$$

Example:

$$T(x,y) : \text{R}(x,y)$$
$$T(x,y) : \text{R}(x,z), T(z,y)$$
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} :: body_{1} \]
\[ P_{i2} :: body_{2} \]
\[ \ldots \]

Group by IDB predicate

\[ P_{1} :: body_{11} \cup body_{12} \cup \ldots \]
\[ P_{2} :: body_{21} \cup body_{22} \cup \ldots \]
\[ \ldots \]

Each rule is a Select-Project-Join-Union query

\[ P_{1} :: SPJU_{1} \]
\[ P_{2} :: SPJU_{2} \]
\[ \ldots \]

Example:

\[ T(x,y) :: R(x,y) \]
\[ T(x,y) :: R(x,z), T(z,y) \]
\[ \Rightarrow \]
\[ T(x,y) :: R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y)) \]
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} :- \text{body}_{1} \]
\[ P_{i2} :- \text{body}_{2} \]

\[ \ldots \]

\[ P \_i \text{group by IDB predicate} \]

\[ P_{1} :- \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_{2} :- \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]

\[ \ldots \]

Each rule is a Select-Project-Join-Union query

Naïve datalog evaluation algorithm:

\[ P_{1} = P_{2} = \ldots = \emptyset \]

Loop

New\( P_{1} = \text{SPJU}_{1} \); New\( P_{2} = \text{SPJU}_{2} \); \ldots

if (New\( P_{1} = P_{1} \) and New\( P_{2} = P_{2} \) and \ldots)

then exit

\[ P_{1} = \text{NewP}_{1} ; P_{2} = \text{NewP}_{2} ; \ldots \]

Endloop

Example:

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

\[ \Rightarrow \]

\[ T(x,y) :- R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y)) \]
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} \leftarrow \text{body}_{1} \]
\[ P_{i2} \leftarrow \text{body}_{2} \]
\[ \ldots \]

\[ \Rightarrow \]

\[ P_{1} \leftarrow \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_{2} \leftarrow \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]
\[ \ldots \]

Group by IDB predicate

\[ \Rightarrow \]

Each rule is a Select-Project-Join-Union query

Naïve datalog evaluation algorithm:

\[ P_{1} = P_{2} = \ldots = \emptyset \]

Loop

\[ \text{NewP}_{1} = \text{SPJU}_{1}; \text{NewP}_{2} = \text{SPJU}_{2}; \ldots \]
\[ \text{if} \ (\text{NewP}_{1} = P_{1} \text{ and } \text{NewP}_{2} = P_{2} \text{ and } \ldots) \]
\[ \text{then exit} \]
\[ P_{1} = \text{NewP}_{1}; P_{2} = \text{NewP}_{2}; \ldots \]

Endloop

Example:

\[ T(x,y) \leftarrow R(x,y) \]
\[ T(x,y) \leftarrow R(x,z), T(z,y) \]

\[ \Rightarrow \]

\[ T(x,y) \leftarrow R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y)) \]

\[ T = \emptyset \]

Loop

\[ \text{NewT}(x,y) = R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y)) \]
\[ \text{if} \ (\text{NewT} = T) \]
\[ \text{then exit} \]
\[ T = \text{NewT} \]

Endloop
Discussion

• A datalog program always terminates (why?)

• What is the running time of a datalog program as a function of the input database?
Discussion

• A datalog program *always* terminates (why?)
  – Number of possible tuples in IDB is $|\text{Dom}|^{\text{arity}(R)}$

• What is the running time of a datalog program as a function of the input database?
  – Number of iteration is $\leq |\text{Dom}|^{\text{arity}(R)}$
  – Each iteration is a relational query
Problem with the Naïve Algorithm

• The same facts are discovered over and over again

• The *semi-naïve* algorithm tries to reduce the number of facts discovered multiple times
Let $V$ be a view computed by one datalog rule (no recursion)

$$V :\neg \text{ body}$$

If (some of) the relations are updated:  $R_1 \leftarrow R_1 \cup \Delta R_1$, $R_1 \leftarrow R_2 \cup \Delta R_2$, ...

Then the view is also modified as follows:  $V \leftarrow V \cup \Delta V$

**Incremental view maintenance:**
Compute $\Delta V$ without having to recompute $V$
Incremental View Maintenance

Example 1:

\[ V(x,y) \leftarrow R(x,z), S(z,y) \]

If \( R \leftarrow R \cup \Delta R \) then what is \( \Delta V(x,y) \) ?
Incremental View Maintenance

Example 1:

\[ V(x,y) :- R(x,z), S(z,y) \]

If \( R \leftarrow R \cup \Delta R \) then what is \( \Delta V(x,y) \) ?

\[ \Delta V(x,y) :- \Delta R(x,z), S(z,y) \]
Incremental View Maintenance

Example 2:

\[ V(x,y) :- R(x,z), S(z,y) \]

If \( R \leftarrow R \cup \Delta R \) and \( S \leftarrow S \cup \Delta S \)
then what is \( \Delta V(x,y) \) ?
Incremental View Maintenance

Example 2:

\[ V(x, y) :- R(x, z), S(z, y) \]

If \( R \leftarrow R \cup \Delta R \) and \( S \leftarrow S \cup \Delta S \) then what is \( \Delta V(x, y) \)?

\[
\begin{align*}
\Delta V(x, y) & :- \Delta R(x, z), S(z, y) \\
\Delta V(x, y) & :- R(x, z), \Delta S(z, y) \\
\Delta V(x, y) & :- \Delta R(x, z), \Delta S(z, y)
\end{align*}
\]
Incremental View Maintenance

Example 3:

\[ V(x,y) :- T(x,z), T(z,y) \]

If \( T \leftarrow T \cup \Delta T \) then what is \( \Delta V(x,y) \) ?
Incremental View Maintenance

Example 3:

\[ V(x,y) :- T(x,z), T(z,y) \]

If \( T \leftarrow T \cup \Delta T \)
then what is \( \Delta V(x,y) \)?

\[ \Delta V(x,y) :- \Delta T(x,z), T(z,y) \]
\[ \Delta V(x,y) :- T(x,z), \Delta T(z,y) \]
\[ \Delta V(x,y) :- \Delta T(x,z), \Delta T(z,y) \]
Semi-naïve Evaluation Algorithm

• Naïve algorithm:

\[
P_0 = \text{InitialValue} \\
\text{Repeat} \\
P_k = f(P_{k-1}) \\
\text{Until no-more-change}
\]

• Semi-naïve algorithm
Semi-naïve Evaluation Algorithm

- Naïve algorithm:

  \[ P_0 = \text{InitialValue} \]
  \[ \text{Repeat} \]
  \[ P_k = f(P_{k-1}) \]
  \[ \text{Until no-more-change} \]

- Semi-naïve algorithm

  \[ P_0 = \Delta_0 = \text{InitialValue} \]
  \[ \text{Repeat} \]
  \[ \Delta_k = \Delta f(P_{k-1}, \Delta_{k-1}) - P_{k-1} \]
  \[ P_k = P_{k-1} \cup \Delta_k \]
  \[ \text{Until no-more-change} \]
Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part. Each $P_i$ defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

$P_1 = \Delta P_1 = \text{non-recursive-}SPJU_1$, $P_2 = \Delta P_2 = \text{non-recursive-}SPJU_2$, ...

Loop

$\Delta P_1 = \Delta SPJU_1(P_1, P_2..., \Delta P_1, \Delta P_2 ...) - P_1$;
$\Delta P_2 = \Delta SPJU_2(P_1, P_2..., \Delta P_1, \Delta P_2 ...) - P_2$;
...

if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and ...) then break

$P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$; ...

Endloop
Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.
Each $P_i$ defined by non-recursive-$SPJU_i$ and (recursive-)SPJU$_i$.

$$P_1 = \Delta P_1 = \text{non-recursive}-SPJU_1, \quad P_2 = \Delta P_2 = \text{non-recursive}-SPJU_2, \ldots$$

Loop

$$\Delta P_1 = \Delta SPJU_1(P_1, P_2, \ldots, \Delta P_1, \Delta P_2, \ldots) - P_1;$$
$$\Delta P_2 = \Delta SPJU_2(P_1, P_2, \ldots, \Delta P_1, \Delta P_2, \ldots) - P_2;$$

... if $(\Delta P_1 = \emptyset \text{ and } \Delta P_2 = \emptyset \text{ and } \ldots)$

then break

$$P_1 = P_1 \cup \Delta P_1; \quad P_2 = P_2 \cup \Delta P_2; \quad \ldots$$

Endloop

Example:

$T(x, y) :- R(x, y)$
$T(x, y) :- R(x, z), T(z, y)$

$T = \Delta T = \ ?$ (non-recursive rule)

Loop

$\Delta T(x, y) = \ ?$ (recursive $\Delta$-rule)

if $(\Delta T = \emptyset)$

then break

$T = T \cup \Delta T$

Endloop
Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part. Each \( P_i \) defined by non-recursive-SPJU\(_i\) and (recursive-)SPJU\(_i\).

\[
P_1 = \Delta P_1 = \text{non-recursive-SPJU}_1, \ P_2 = \Delta P_2 = \text{non-recursive-SPJU}_2, \ ...
\]

Loop

\[
\Delta P_1 = \Delta \text{SPJU}_1(P_1, P_2, ..., \Delta P_1, \Delta P_2, ...) - P_1;
\]

\[
\Delta P_2 = \Delta \text{SPJU}_2(P_1, P_2, ..., \Delta P_1, \Delta P_2, ...) - P_2;
\]

... 

if (\( \Delta P_1 = \emptyset \) and \( \Delta P_2 = \emptyset \) and ...) then break

\[
P_1 = P_1 \cup \Delta P_1; \ P_2 = P_2 \cup \Delta P_2; \ ...
\]

Endloop

Example:

\[
T(x,y) : - R(x,y)
\]

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

\[
T(x,y) = \Delta T(x,y) = R(x,y)
\]

Loop

\[
\Delta T(x,y) = R(x,z), \ \Delta T(z,y), \ \text{not} \ T(x,y)
\]

if (\( \Delta T = \emptyset \)) then break

\[
T = T \cup \Delta T
\]

Endloop
Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part. Each $P_i$ defined by non-recursive-$\text{SPJU}_i$ and (recursive-)$\text{SPJU}_i$.

Example:

$$T(x,y) :- R(x,y)$$

$$T(x,y) :- R(x,z), T(z,y)$$

Note: for any linear datalog programs, the semi-naïve algorithm has only one $\Delta$-rule for each rule!
Simple datalog programs

R encodes a graph

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Initially:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ΔT =

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T =

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T = ΔT = R
Loop
ΔT(x,y) = R(x,z), ΔT(z,y), not T(x,y)
if (ΔT = ∅)
    then break
T = T ∪ ΔT
Endloop
Simple datalog programs

R encodes a graph

Initially:

T= \Delta T = R

Loop
\Delta T(x,y)= R(x,z), \Delta T(z,y), not T(x,y)
if (\Delta T = \emptyset)
then break
T = T \cup \Delta T
Endloop

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

First iteration:

\Delta T=

paths of length 2
Simple datalog programs

R encodes a graph

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Initially:

First iteration:

Second iteration:

T=
ΔT = R
Loop
ΔT(x,y) = R(x,z), ΔT(z,y), not T(x,y)
if (ΔT = ∅)
then break
T = T ∪ ΔT
Endloop
Simple datalog programs

R encodes a graph

\[ \begin{align*}
R &= \{ \langle 1, 2 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 5 \rangle \} \\
\Delta T &= \{ \langle 1, 2 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 5 \rangle \} \\
T(x, y) &: \text{paths of length } 2 \\
T(x, y) &: \text{paths of length } 3 \\
T(x, y) &: \text{paths of length } 4
\end{align*} \]

\[ \begin{align*}
T &= \{ \langle 1, 2 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 5 \rangle \} \\
\Delta T &= \{ \langle 1, 2 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 5 \rangle \} \\
T &= T \cup \Delta T
\end{align*} \]

\[ \begin{align*}
T &= \{ \langle 1, 2 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 5 \rangle \} \\
\Delta T &= \{ \langle 1, 2 \rangle, \langle 1, 4 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 5 \rangle \} \\
T &= T \cup \Delta T
\end{align*} \]

Initial conditions:

\[ \begin{align*}
T &= \{ \} \\
\Delta T &= \{ \} \\
\end{align*} \]

Loop

\[ \text{Loop} \]

\[ \text{Break} \]

Endloop
Discussion of Semi-Naïve Algorithm

• Avoids re-computing some tuples, but not all tuples

• Easy to implement, no disadvantage over naïve

• A rule is called *linear* if its body contains only one recursive IDB predicate:
  – A linear rule always results in a single incremental rule
  – A non-linear rule may result in multiple incremental rules
Summary So Far

• Simple syntax for expressing queries with recursion
• Bottom-up evaluation – always terminates
  – Naïve evaluation
  – Semi-naïve evaluation
• Next:
  – Datalog semantics
  – Datalog with negation
Semantics of a Datalog Program

Three different, equivalent semantics:

• Minimal model semantics
• Least fixpoint semantics
• Proof-theoretic semantics
Minimal Model Semantics

To each rule \( r \): \[ P(x_1 \ldots x_k) : - R_1(...), R_2(...), \ldots \]
Minimal Model Semantics

To each rule \( r: \)

\[
P(x_1 \ldots x_k) \leftarrow R_1(\ldots), R_2(\ldots), \ldots
\]

Associate the logical sentence \( \Sigma_r: \)

\[
\forall z_1 \ldots \forall z_n. [(R_1(\ldots) \land R_2(\ldots) \land \ldots) \Rightarrow P(\ldots)]
\]

All variables in the rule
Minimal Model Semantics

To each rule $r$: $P(x_1 \ldots x_k) \Leftarrow R_1(\ldots), R_2(\ldots), \ldots$

Associate the logical sentence $\Sigma_r$: $\forall z_1 \ldots \forall z_n. [(R_1(\ldots) \land R_2(\ldots) \land \ldots) \Rightarrow P(\ldots)]$

Same as: $\forall x_1 \ldots \forall x_k. [\exists y_1 \ldots \exists y_m. (R_1(\ldots) \land R_2(\ldots) \land \ldots) \Rightarrow P(\ldots)]$

Head variables

Existential variables

All variables in the rule
Minimal Model Semantics

To each rule $r$: $P(x_1...x_k) :- R_1(...), R_2(...), ...$

Associate the logical sentence $\Sigma_r$: $\forall z_1...\forall z_n. [(R_1(...)^{\land} R_2(...) ^{\land} ...) \implies P(...)]$

Same as: $\forall x_1...\forall x_k. [\exists y_1...\exists y_m. (R_1(...) \land R_2(...) ^{\land} ...) \implies P(...)]$

**Definition.** If $P$ is a datalog program, $\Sigma_P$ is the set of all logical sentences associated to its rules.
Minimal Model Semantics

To each rule r:

\[ P(x_1 \ldots x_k) \leftarrow R_1(...), R_2(...), \ldots \]

Associate the logical sentence \( \Sigma_r \):

\[ \forall z_1 \ldots \forall z_n. \ [(R_1(...) \land R_2(...) \land \ldots) \Rightarrow P(...)] \]

Same as:

\[ \forall x_1 \ldots \forall x_k. \ [(\exists y_1 \ldots \exists y_m. (R_1(...) \land R_2(...) \land \ldots) \Rightarrow P(...)] \]

**Definition.** If \( P \) is a datalog program, \( \Sigma_P \) is the set of all logical sentences associated to its rules.

Example. Rule:

\[ T(x,y) \leftarrow R(x,z), T(z,y) \]

Sentence: \( \forall x. \forall y. \forall z. (R(x,z) \land T(z,y) \Rightarrow T(x,y)) \]

\[ \equiv \forall x. \forall y. (\exists z. R(x,z) \land T(z,y) \Rightarrow T(x,y)) \]
Minimal Model Semantics

**Definition.** A pair $(I,J)$ where $I$ is an EDB and $J$ is an IDB is a *model* for $P$, if $(I,J) \models \Sigma_P$

**Definition.** Given an EDB database instance $I$ and a datalog program $P$, the minimal model, denoted $J = P(I)$ is a minimal database instance $J$ s.t. $(I,J) \models \Sigma_P$

**Theorem.** The minimal model always exists, and is unique.
**Definition.** A pair \((I,J)\) where \(I\) is an EDB and \(J\) is an IDB is a *model* for \(P\), if \((I,J) \models \Sigma_P\).

**Definition.** Given an EDB database instance \(I\) and a datalog program \(P\), the minimal model, denoted \(J = P(I)\) is a minimal database instance \(J\) s.t. \((I,J) \models \Sigma_P\).

**Theorem.** The minimal model always exists, and is unique.

**Example:**

Which of these IDBs are *models*?
Which are *minimal models*?

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
T(x,y) & - & R(x,y) & \quad & T(x,y) & - & R(x,z), \; T(z,y) \\
\hline
1 & 2 & \quad & \quad & \quad & \quad & \\
2 & 3 & \quad & \quad & \quad & \quad & \\
3 & 4 & \quad & \quad & \quad & \quad & \\
4 & 5 & \quad & \quad & \quad & \quad & \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
R= & 1 & 2 & \quad & 1 & 3 & \quad & \\
\hline
1 & 2 & \quad & \quad & 3 & 4 & \quad & \\
2 & 3 & \quad & \quad & 4 & 5 & \quad & \\
3 & 4 & \quad & \quad & 1 & 3 & \quad & \\
4 & 5 & \quad & \quad & 2 & 4 & \quad & \\
\hline
\end{array}
\]
**Minimal Model Semantics**

**Definition.** A pair \((I,J)\) where \(I\) is an EDB and \(J\) is an IDB is a *model* for \(P\), if \((I,J) \models \Sigma_P\).

**Definition.** Given an EDB database instance \(I\) and a datalog program \(P\), the minimal model, denoted \(J = P(I)\) is a minimal database instance \(J\) s.t. \((I,J) \models \Sigma_P\).

**Theorem.** The minimal model always exists, and is unique.

**Example:**

<table>
<thead>
<tr>
<th>R=</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T=</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Which of these IDBs are *models*? Which are *minimal models*?

\(T(x,y) \leftarrow R(x,y)\)
\(T(x,y) \leftarrow R(x,z), T(z,y)\)
**Minimal Model Semantics**

**Definition.** A pair \((I, J)\) where \(I\) is an EDB and \(J\) is an IDB is a *model* for \(P\), if \((I, J) \models \Sigma_P\).

**Definition.** Given an EDB database instance \(I\) and a datalog program \(P\), the minimal model, denoted \(J = P(I)\) is a minimal database instance \(J\) s.t. \((I, J) \models \Sigma_P\).

**Theorem.** The minimal model always exists, and is unique.

**Example:**

- \(T(x, y) \leftarrow R(x, y)\)
- \(T(x, y) \leftarrow R(x, z), T(z, y)\)

Which of these IDBs are *models*? Which are *minimal models*?

**Table:**

<table>
<thead>
<tr>
<th>(R)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(T)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(T=)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(4)</td>
<td>(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1)</td>
<td>(6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>(7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td>(9)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(5)</td>
<td>(10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(5)</td>
<td>(11)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All 25 pairs of nodes
**Definition.** Fix an EDB I, and a datalog program P. The *immediate consequence* operator $T_P$ is defined as follows. For any IDB J:

$$T_P(J) = \text{all IDB facts that are immediate consequences from I and J.}$$

**Fact.** For any datalog program P, the immediate consequence operator is monotone. In other words, if $J_1 \subseteq J_2$ then $T_P(J_1) \subseteq T_P(J_2)$. 

---

**Minimal Fixpoint Semantics**

---
Definition. Fix an EDB I, and a datalog program P. The immediate consequence operator $T_P$ is defined as follows. For any IDB $J$:

$$T_P(J) = \text{all IDB facts that are immediate consequences from I and } J.$$ 

Fact. For any datalog program P, the immediate consequence operator is monotone. In other words, if $J_1 \subseteq J_2$ then $T_P(J_1) \subseteq T_P(J_2)$.

Theorem. The immediate consequence operator has a unique, minimal fixpoint $J$: $\text{fix}(T_P) = J$, where $J$ is the minimal instance with the property $T_P(J) = J$.

Proof: using Knaster-Tarski’s theorem for monotone functions. The fixpoint is given by:

$$\text{fix } (T_P) = J_0 \cup J_1 \cup J_2 \cup \ldots \quad \text{where } J_0 = \emptyset, \quad J_{k+1} = T_P(J_k)$$
Minimal Fixpoint Semantics

\[ R = \]

\[ T = \]

\[ J_0 = \emptyset \]
\[ J_1 = T_P(J_0) \]
\[ J_2 = T_P(J_1) \]
\[ J_3 = T_P(J_2) \]
\[ J_4 = T_P(J_3) \]

\[ T(x,y) : - R(x,y) \]
\[ T(x,y) : - R(x,z), T(z,y) \]
Proof Theoretic Semantics

Every fact in the IDB has a *derivation tree*, or *proof tree* justifying its existence.

\[ R = \]

\[
\begin{array}{cc}
1 & 2 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

Derivation tree of \( T(1,4) \)

\[
T(x,y) :- R(x,y) \\
T(x,y) :- R(x,z), T(z,y)
\]
Adding Negation: Datalog\(^{-}\)

**Example:** compute the complement of the transitive closure

\[
\begin{align*}
T(x,y) & \leftarrow R(x,y) \\
T(x,y) & \leftarrow T(x,z), R(z,y) \\
\text{CT}(x,y) & \leftarrow \text{Node}(x), \text{Node}(y), \text{not } T(x,y)
\end{align*}
\]

What does this mean??
Recursion and Negation
Don’t Like Each Other

EDB: \( I = \{ R(a) \} \)

Which IDBs are models of \( P \)?

\[
\begin{align*}
J_1 &= \{ \} \\
J_2 &= \{ S(a) \} \\
J_3 &= \{ T(a) \} \\
J_4 &= \{ S(a), T(a) \}
\end{align*}
\]

\[
S(x) ::= R(x), \text{ not } T(x) \\
T(x) ::= R(x), \text{ not } S(x)
\]
Recursion and Negation
Don’t Like Each Other

EDB: \[ I = \{ R(a) \} \]

\[ S(x) :- R(x), \text{not } T(x) \]
\[ T(x) :- R(x), \text{not } S(x) \]

Which IDBs are models of \( P \)?

\[ J_1 = \{ \} \]
\[ J_2 = \{ S(a) \} \]
\[ J_3 = \{ T(a) \} \]
\[ J_4 = \{ S(a), T(a) \} \]

No: both rules fail
Recursion and Negation
Don’t Like Each Other

EDB: \( I = \{ \text{R(a)} \} \)

\[\begin{align*}
\text{S(x)} & \leftarrow \text{R(x)}, \neg \text{T(x)} \\
\text{T(x)} & \leftarrow \text{R(x)}, \neg \text{S(x)}
\end{align*}\]

Which IDBs are models of \( P \)?

\( J_1 = \{ \} \)  
\( J_2 = \{ \text{S(a)} \} \)  
\( J_3 = \{ \text{T(a)} \} \)  
\( J_4 = \{ \text{S(a)}, \text{T(a)} \} \)

Yes: the facts in \( J_2 \) are \( \text{R(a)}, \text{S(a)}, \neg \text{T(a)} \) and both rules are true.

No: both rules fail
Recursion and Negation
Don’t Like Each Other

EDB: \( I = \{ R(a) \} \)

\[ \begin{align*}
S(x) & :\ - R(x), \ not \ T(x) \\
T(x) & :\ - R(x), \ not \ S(x)
\end{align*} \]

Which IDBs are models of \( P \)?

\[ \begin{align*}
J_1 &= \{ \} \\
J_2 &= \{ S(a) \} \\
J_3 &= \{ T(a) \} \\
J_4 &= \{ S(a), T(a) \}
\end{align*} \]

No: both rules fail

Yes: the facts in \( J_2 \) are \( R(a), S(a), \neg T(a) \) and both rules are true.

Yes
Recursion and Negation
Don’t Like Each Other

EDB: \( I = \{ R(a) \} \)

\[
S(x) :- R(x), \neg T(x)
\]

\[
T(x) :- R(x), \neg S(x)
\]

Which IDBs are models of \( P \)?

\[ J_1 = \{ \} \]
\[ J_2 = \{ S(a) \} \]
\[ J_3 = \{ T(a) \} \]
\[ J_4 = \{ S(a), T(a) \} \]

No: both rules fail

Yes: the facts in \( J_2 \) are \( R(a), S(a), \neg T(a) \) and both rules are true.

Yes

Yes
Recursion and Negation
Don’t Like Each Other

EDB: \( I = \{ R(a) \} \)

\[ \begin{align*}
S(x) & : - R(x), \text{not } T(x) \\
T(x) & : - R(x), \text{not } S(x)
\end{align*} \]

Which IDBs are models of \( P \)?

- \( J_1 = \{ \} \)
  - No: both rules fail

- \( J_2 = \{ S(a) \} \)
  - Yes: the facts in \( J_2 \) are \( R(a), S(a), \neg T(a) \) and both rules are true.

- \( J_3 = \{ T(a) \} \)
  - Yes

- \( J_4 = \{ S(a), T(a) \} \)
  - Yes

There is no \textit{minimal} model!
Recursion and Negation
Don’t Like Each Other

EDB: \( I = \{ R(a) \} \)

\[ S(x) : \sim R(x), \text{not } T(x) \]
\[ T(x) : \sim R(x), \text{not } S(x) \]

Which IDBs are models of \( P \)?

\( J_1 = \{ \} \)
\( J_2 = \{ S(a) \} \)
\( J_3 = \{ T(a) \} \)
\( J_4 = \{ S(a), T(a) \} \)

No: both rules fail

Yes: the facts in \( J_2 \) are \( R(a), S(a), \sim T(a) \) and both rules are true.

Yes

Yes

There is no minimal model!

There is no minimal fixpoint!
(Why does Knaster-Tarski’s theorem fail?)
Adding Negation: datalog

- **Solution 1: Stratified Datalog**
  - Insist that the program be *stratified*: rules are partitioned into strata, and an IDB predicate that occurs only in strata \( \leq k \) may be negated in strata \( \geq k+1 \)

- **Solution 2: Inflationary-fixpoint Datalog**
  - Compute the fixpoint of \( J \cup T_P(J) \)
  - Always terminates (why ?)

- **Solution 3: Partial-fixpoint Datalog**
  - Compute the fixpoint of \( T_P(J) \)
  - May not terminate
A datalog\textsuperscript{\neg} program is *stratified* if its rules can be partitioned into $k$ strata, such that:

- If an IDB predicate $P$ appears negated in a rule in stratum $i$, then it can only appear in the head of a rule in strata $1, 2, \ldots, i-1$.

Note: a datalog\textsuperscript{\neg} program either is stratified or it ain’t!

Which programs are stratified?

- $T(x,y) :- R(x,y)$
- $T(x,y) :- T(x,z), R(z,y)$
- $CT(x,y) :- \text{Node}(x), \text{Node}(y), \text{not } T(x,y)$
- $S(x) :- R(x), \text{not } T(x)$
- $T(x) :- R(x), \text{not } S(x)$
Stratified datalog\(^–\)

- Evaluation algorithm for stratified datalog\(^–\):
  - For each stratum \(i = 1, 2, \ldots,\) do:
    - Treat all IDB’s defined in prior strata as EBS
    - Evaluate the IDB’s defined in stratum \(i\), using either the naïve or the semi-naïve algorithm

\[
\begin{align*}
T(x,y) & : - R(x,y) \\
T(x,y) & : - T(x,z), R(z,y) \\
CT(x,y) & : - \text{Node}(x), \text{Node}(y), \text{not } T(x,y)
\end{align*}
\]

Does this compute a minimal model?
Stratified datalog

• Evaluation algorithm for stratified datalog:

  • For each stratum \( i = 1, 2, \ldots \), do:
    – Treat all IDB’s defined in prior strata as EBS
    – Evaluate the IDB’s defined in stratum \( i \), using either the naïve or the semi-naïve algorithm

Does this compute a minimal model?

NO:
\( J_1 = \{ T = \text{transitive closure}, CT = \text{its complement} \} \)
\( J_2 = \{ T = \text{all pairs of nodes}, CT = \text{empty} \} \)

\[
\begin{align*}
T(x,y) & : - R(x,y) \\
T(x,y) & : - T(x,z), R(z,y) \\
CT(x,y) & : - \text{Node}(x), \text{Node}(y), \text{not } T(x,y)
\end{align*}
\]
Inflationary-fixpoint datalog

Let $P$ be any datalog$^-$ program, and $I$ an EDB.
Let $T_P(J)$ be the immediate consequence operator.
Let $F(J) = J \cup T_P(J)$ be the inflationary immediate consequence operator.

Define the sequence: $J_0 = \emptyset$, $J_{n+1} = F(J_n)$, for $n \geq 0$.

**Definition.** The inflationary fixpoint semantics of $P$ is $J = J_n$ where $n$ is such that $J_{n+1} = J_n$.

Why does there always exist an $n$ such that $J_n = F(J_n)$?

Find the inflationary semantics for:

- $T(x,y) :- R(x,y)$
- $T(x,y) :- T(x,z), R(z,y)$
- $CT(x,y) :- Node(x), Node(y), \text{not } T(x,y)$

- $S(x) :- R(x), \text{not } T(x)$
- $T(x) :- R(x), \text{not } S(x)$
Inflationary-fixpoint datalog⁻

• Evaluation for Inflationary-fixpoint datalog⁻

• Use the naïve, of the semi-naïve algorithm

• Inhibit any optimization that rely on monotonicity (e.g. out of order execution)
Partial-fixpoint datalog\(^-\)*

Let \( \mathbf{P} \) be any datalog\(^-\) program, and \( I \) an EDB.
Let \( T_{\mathbf{P}}(J) \) be the *immediate consequence* operator.

Define the sequence: \( J_0 = \emptyset, J_{n+1} = T_{\mathbf{P}}(J_n) \), for \( n \geq 0 \).

**Definition.** The partial fixpoint semantics of \( \mathbf{P} \) is \( J = J_n \) where \( n \) is such that \( J_{n+1} = J_n \), if such an \( n \) exists, undefined otherwise.

Find the partial fixpoint semantics for:

- \( T(x,y) :\!\!:\!\!:\!\!:- R(x,y) \)
- \( T(x,y) :\!\!:\!\!:\!\!:- T(x,z), R(z,y) \)
- \( CT(x,y) :\!\!:\!\!:\!\!:- \text{Node}(x), \text{Node}(y), \text{not} T(x,y) \)

Note: there may not exist an \( n \) such that \( J_n = F(J_n) \)

- \( S(x) :\!\!:\!\!:\!\!:- R(x), \text{not} T(x) \)
- \( T(x) :\!\!:\!\!:\!\!:- R(x), \text{not} S(x) \)
Summary of Datalog

• Recursion = easy and fun
• Recursion + negation = nightmare
• Powerful optimizations:
  – Incremental view updates
  – Magic sets (did not discuss in class)
• SQL implements limited recursion