Introduction to Data Management
CSE 344

Lecture 15: NoSQL and JSon

Announcements

- Assignments:
  - WQ4 and HW4 due this week
  - HW5 will be out on Wednesday
    - Due on Friday, 11/11
  - [There is no Web Quiz 5]
  - Midterm next Monday in class

- Today's lecture:
  - Datalog review
  - JSon
    - The book covers XML instead (skim 11.1-11.3, 12.1)

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Review: Datalog program

A datalog program is a collection of one or more rules

Each rule tells us how to infer the contents of relations from others

Example: Find all actors with Bacon number ≤ 2

```
B0(x) :- Actor(x,'Kevin', 'Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B0(y)
B2(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B1(y)
 Q4(x) :- B0(x)
 Q4(x) :- B2(x)
```

Note: Q4 means the *union* of B0 and B2

We actually don't need Q4(x) :- B0(x); Why?

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Review: Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

```
U1(x,y) :- Movie(x,z,1994), y>1910
```

```
U2(x)   :- Movie(x,z,1994), not Casts(u,x)
```

> A datalog rule is *safe* if every variable appears in some positive relational atom

Simpler than in relational calculus

Datalog v.s. Relational Algebra

- Fact: Every expression in the basic relational algebra can be expressed as a Datalog query

- But operations in the extended relational algebra (grouping, aggregation, and sorting) have no corresponding features in the version of datalog that we discussed today

- Similarly, datalog can express recursion, which relational algebra cannot

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Union R(A,B,C) ∪ S(D,E,F)

U(x,y,z) :- R(x,y,z)
U(x,y,z) :- S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Intersection R(A,B,C) ∩ S(D,E,F)

I(x,y,z) :- R(x,y,z), S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Selection: $\sigma_{x>100 \text{ and } y=\text{‘foo’}}(R)$

L(x,y,z) :- R(x,y,z), x > 100, y=‘foo’

Selection $\sigma_{x>100 \textbf{ or } y=\text{‘foo’}}(R)$

L(x,y,z) :- R(x,y,z), x > 100

L(x,y,z) :- R(x,y,z), y=‘foo’

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Equi-join: $R \bowtie_{R.A=S.D \text{ and } R.B=S.E} S$

J(x,y,z,q) :- R(x,y,z), S(x,y,q)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Projection

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

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

To express difference, we add negation

D(x,y,z) :- R(x,y,z), NOT S(x,y,z)

Examples

R(A,B,C)

S(D,E,F)

T(G,H)

Translate: $\Pi_A(\sigma_{B=3}(R))$

A(a) :- R(a,3,_)

Underscore used to denote an "anonymous variable”

Each such variable is unique

Examples

R(A,B,C)

S(D,E,F)

T(G,H)

Translate: $\Pi_A(\sigma_{B=3}(R) \bowtie_{R.A=S.D} \sigma_{E=5}(S))$

A(a) :- R(a,3,_), S(a,5,_)

Friend(name1, name2)
Enemy(name1, name2)

More Examples

Find Joe's friends, and Joe's friends of friends.

```
A(x) :- Friend('Joe', x)
A(x) :- Friend('Joe', z), Friend(z, x)
```

Friend(name1, name2)
Enemy(name1, name2)

You try it!

Find all of Joe's friends who do not have any friends except for Joe:

```
JoeFriends(x) :- Friend('Joe',x)
NonAns(x) :- JoeFriends(x), Friend(x,y), y != ‘Joe’
A(x) :- JoeFriends(x), NOT NonAns(x)
```

Find all persons x that have a friend all of whose enemies are x's enemies.

```
Everyone(x) :- Friend(x,y)
NonAns(x) :- Friend(x,y), Enemy(y,z), NOT Enemy(x,z)
A(x) :- Everyone(x), NOT NonAns(x)
```

Friend(name1, name2)
Enemy(name1, name2)

More Examples

Find all people such that all their enemies' enemies are their friends

- Q: if someone doesn't have any enemies nor friends, do we want them in the answer?
- A: Yes!

```
Everyone(x) :- Friend(x,y)
Everyone(x) :- Friend(y,x)
Everyone(x) :- Enemy(x,y)
Everyone(x) :- Enemy(y,x)
NonAns(x) :- Enemy(x,y),Enemy(y,z), NOT Friend(x,z)
A(x) :- Everyone(x), NOT NonAns(x)
```

Translating queries

How to write a complex SQL query:

- Write it in RC
- Translate RC to datalog
- Translate datalog to SQL

Take shortcuts when you know what you're doing

(the next 8 slides are those that we didn't get to in class)

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog$^\neg$ to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog$^{\neg}$ to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$

$P \Rightarrow Q$ same as $\neg P \vee Q$

$\forall x\ P(x)$ same as $\neg \exists x\ \neg P(x)$

Step 1: Replace $\forall$ with $\exists$ using de Morgan's Laws

$\neg(\neg P \vee Q)$ same as $P \wedge \neg Q$

$Q(x) = \exists y.\ Likes(x, y) \wedge \neg \exists z.(Serves(z,y) \wedge \neg Frequents(x,z))$

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog$^{\neg}$ to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

$P \Rightarrow Q$ same as
$\neg P \vee Q$

$\forall x\ P(x)$ same as
$\neg \exists x\ \neg P(x)$

Step 1: Replace $\forall$ with $\exists$ using de Morgan's Laws

$\neg(\neg P \vee Q)$ same as
$P \wedge \neg Q$

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg \exists z.(Serves(z,y) \wedge \neg Frequents(x,z))$$

Step 2: Make sure the query is domain independent

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg \exists z.(Likes(x,y) \wedge Serves(z,y) \wedge \neg Frequents(x,z))$$

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog$^{\neg}$ to SQL

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg\ \underbrace{\exists z.(Likes(x,y) \wedge Serves(z,y) \wedge \neg Frequents(x,z))}_{H(x,y)}$$

Step 3: Create a datalog rule for each subexpression;
(shortcut: only for “important” subexpressions)

H(x,y) :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x) :- Likes(x,y), not H(x,y)

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog$^{\neg}$ to SQL

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE ……
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog¬ to SQL

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE … …)
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog¬ to SQL

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE L2.drinker=L.drinker and L2.beer=L.beer
         and L2.beer=S.beer
         and not exists (SELECT * FROM Frequents F
                         WHERE F.drinker=L2.drinker
                           and F.bar=S.bar))
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to Datalog$^{\neg}$ to SQL

H(x,y) :- ~~Likes(x,y),~~Serves(z,y), not Frequents(x,z)
Q(x) :- Likes(x,y), not H(x,y)

Unsafe rule

Improve the SQL query by using an unsafe datalog rule

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Serves S
   WHERE L.beer=S.beer
         and not exists (SELECT * FROM Frequents F
                         WHERE F.drinker=L.drinker
                               and F.bar=S.bar))
```

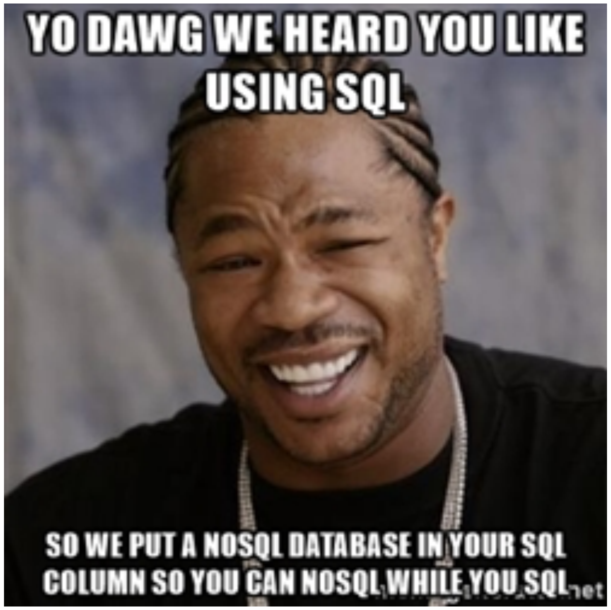
Datalog Summary

- EDB (base relations) and IDB (derived relations)
- Datalog program = set of rules
- Datalog is recursive
  - But we only focused on non-recursive datalog

- Some reminders about Datalog semantics:
  - Multiple atoms in a rule mean join (or intersection)
  - Variables with the same name are join variables
  - Multiple rules with same head mean union

The New Hipster: NoSQL

Where are we?



- Relational data model
  - Storage: file organization, indexes
  - Languages: SQL / RA / RC / Datalog
  - Query processing
- Non-relational data models (aka NoSQL)
  - Unstructured
  - Semi-structured
  - Hybrid?

What's Wrong with the Relational Data Model?

- Single server DBMS are too small for Web data

- Solution: scale out to multiple servers

- This is hard for relational DMBS
  - Do we copy entire relations to all servers? (expensive)
  - Divide relations into pieces and distribute?
    (break data model – how to execute queries?)

- NoSQL: reduce functionality for easier scale up
  - Simpler data model
  - Simpler query language

Non-Relational Data Models:

- ☞ Key-value stores (unstructured)
  - e.g., Project Voldemort, Memcached
- Document stores (semi-structured)
  - e.g., SimpleDB, CouchDB, MongoDB
- Extensible Record Stores (?)
  - e.g., HBase, Cassandra, PNUTS

Key-Value Data Model

- **Instance:** (key,value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)
- **Schema:** none (!)
- **Language:**
  - `get(key), put(key,value)`
  - Operations on value are not supported
- **How to scale up to multiple servers?**
  - No replication: key k is stored at server h(k)
  - N-way replication: key k stored at h1(k),h2(k),…,hn(k)

How does get(k) work? How does put(k,v) work?

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

Example

- How would you represent the Flights data as key, value pairs?
- Option 1: key=fid, value=entire flight record
- Option 2: key=date, value=all flights that day
- Option 3: key=(origin,dest), value=all flights between

How does query processing work?

Non-Relational Data Models

- Key-value stores (unstructured)
  - e.g., Project Voldemort, Memcached
- ☞ Document stores (semi-structured)
  - e.g., SimpleDB, CouchDB, MongoDB
- Extensible Record Stores (?)
  - e.g., HBase, Cassandra, PNUTS

Document Store Data Model

- **Instance**: (key,document) pairs
  - Key = string/integer, unique for the entire data
  - Document = JSon, or XML
- **Schema:** embedded in JSon / XML document
- **Language:**
  - `get(doc_key), put(doc_key,value)`
  - Limited, non-standard query language on Json (N1QL)
- **How to scale up to multiple servers?**
  - Replicate entire documents, just like key/value pairs

We will discuss JSon in this class

Non-Relational Data Models

- Key-value stores (unstructured)
  - e.g., Project Voldemort, Memcached
- Document stores (semi-structured)
  - e.g., SimpleDB, CouchDB, MongoDB
- ☞ Extensible Record Stores (?)
  - e.g., HBase, Cassandra, PNUTS

Extensible Record Stores

- Based on Google's BigTable
- **Instance:** Rows and columns, as in relational
- **Schema:** same as relational
- **Language:** Java/Python API for manipulating rows
  - `get(key), put(key,value)`
- **How to scale up to multiple servers?**
  - Splitting rows and columns over nodes
  - Rows partitioned using primary key
  - Columns of a table are distributed over multiple nodes by using "column groups"

- HBase is an open source implementation of BigTable