Introduction to Data Management
CSE 344

Lecture 15: NoSQL and JSon

Announcements

- Midterm on Monday
  - Covers everything include this lecture
- Review session: Saturday 4-5pm
  - Location TBD
- Today: NoSQL

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

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

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

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

$P \Rightarrow Q$ 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 $P \wedge \neg Q$

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

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$^{\neg}$ 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 not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE … …)
```

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 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¬ 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))
```

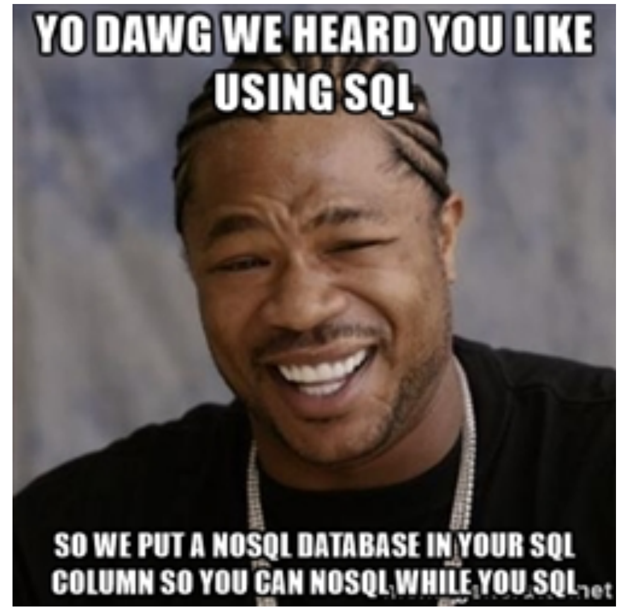
Datalog Summary: all these formalisms are equivalent!

- We have seen these translations:
  - RA → datalog¬
  - RC → datalog¬
- Practice at home, and read *Query Language Primer:*
  - Nonrecursive datalog¬ → RA
  - RA → RC
- Summary:
  - RA, RC, and non-recursive datalog¬ can express the same class of queries, called Relational Queries

End of Relational Data Model

(at least for now ☺)

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

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