Schedule

• Today: Feb. 28 (TH)
  ◆ Datalog and SQL Recursion, ODL.
  ◆ Read Sections 10.3-10.4, 4.1-4.4. Project Part 6 due.

• Mar. 5 (T)
  ◆ More ODL, OQL.
  ◆ Read Sections 9.1. Assignment 7 due.

• Mar. 7 (TH)
  ◆ More OQL.
  ◆ Read Sections 9.2-9.3.

• Mar. 12 (T)
  ◆ Semistructured Data, XML, XQuery.
  ◆ Read Sections 4.6-4.7. Assignment 8 due.
SQL Recursion

WITH

  stuff that looks like Datalog rules
  an SQL query about EDB, IDB

• Rule =

    [RECURSIVE] R(<arguments>) AS
    SQL query
Example

- Find Sally’s cousins, using EDB \text{Par}(\text{child}, \text{parent}).

\begin{verbatim}
WITH
  \text{Sib}(x,y) \text{ AS }
    \text{SELECT } p1.\text{child}, p2.\text{child} \\
    \text{FROM Par p1, Par p2} \\
    \text{WHERE } p1.\text{parent} = p2.\text{parent} \\
    \hspace{1cm} \text{AND } p1.\text{child} <> p2.\text{child},

  \text{RECURSIVE Cousin}(x,y) \text{ AS }
    \text{Sib}
    \hspace{1cm} \text{UNION}
    (\text{SELECT } p1.\text{child}, p2.\text{child} \\
    \text{FROM Par p1, Par p2, Cousin} \\
    \text{WHERE } p1.\text{parent} = Cousin.x \\
    \hspace{1cm} \text{AND } p2.\text{parent} = Cousin.y
    \hspace{1cm} )
  \end{verbatim}

\text{SELECT y} \\
\text{FROM Cousin} \\
\text{WHERE } x = 'Sally';
\end{verbatim}
Plan for Describing Legal SQL Recursion

1. Define “monotonicity,” a property that generalizes “stratification.”

2. Generalize stratum graph to apply to SQL queries instead of Datalog rules.
   - (Non)monotonicity replaces NOT in subgoals.

3. Define semantically correct SQL recursions in terms of stratum graph.

Monotonicity

If relation $P$ is a function of relation $Q$ (and perhaps other things), we say $P$ is monotone in $Q$ if adding tuples to $Q$ cannot cause any tuple of $P$ to be deleted.
Monotonicity Example

In addition to certain negations, an aggregation can cause nonmonotonicity.

\[ \text{Sells(}\text{bar, beer, price}) \]

\[ \text{SELECT AVG(price)} \]
\[ \text{FROM Sells} \]
\[ \text{WHERE bar = 'Joe''s Bar'}; \]

• Adding to \text{Sells} a tuple that gives a new beer Joe sells will usually change the average price of beer at Joe’s.

• Thus, the former result, which might be a single tuple like (2.78) becomes another single tuple like (2.81), and the old tuple is lost.
Generalizing Stratum Graph to SQL

- Node for each relation defined by a “rule.”
- Node for each subquery in the “body” of a rule.
- Arc $P \rightarrow Q$ if
  a) $P$ is “head” of a rule, and $Q$ is a relation appearing in the FROM list of the rule (not in the FROM list of a subquery), as argument of a UNION, etc.
  b) $P$ is head of a rule, and $Q$ is a subquery directly used in that rule (not nested within some larger subquery).
  c) $P$ is a subquery, and $Q$ is a relation or subquery used directly within $P$ [analogous to (a) and (b) for rule heads].
- Label the arc – if $P$ is not monotone in $Q$.
- Requirement for legal SQL recursion: finite strata only.
Example

For the Sib/Cousin example, there are three nodes: Sib, Cousin, and $SQ$ (the second term of the union in the rule for Cousin).

- No nonmonotonicity, hence legal.
A Nonmonotonic Example

Change the UNION to EXCEPT in the rule for Cousin.

```
RECURSIVE Cousin(x,y) AS
Sib
    EXCEPT
        (SELECT p1.child, p2.child
         FROM Par p1, Par p2, Cousin
         WHERE p1.parent = Cousin.x
         AND p2.parent = Cousin.y
         )
```

- Now, adding to the result of the subquery can delete Cousin facts; i.e., Cousin is nonmonotone in $SQ$.
- Infinite number of $-$'s in cycle, so illegal in SQL.
Another Example: NOT Doesn’t Mean Nonmonotone

Leave Cousin as it was, but negate one of the conditions in the where-clause.

RECURSIVE Cousin(x,y) AS
  Sib
  UNION
  (SELECT p1.child, p2.child 
   FROM Par p1, Par p2, Cousin 
   WHERE p1.parent = Cousin.x 
     AND NOT (p2.parent = Cousin.y)
  )

• You might think that SQ depends negatively on Cousin, but it doesn’t.
  ◆ If I add a new tuple to Cousin, all the old tuples still exist and yield whatever tuples in SQ they used to yield.
  ◆ In addition, the new Cousin tuple might combine with old p1 and p2 tuples to yield something new.
Object-Oriented DBMS’s

- ODMG = Object Data Management Group: an OO standard for databases.
- ODL = Object Description Language: design in the OO style.
- OQL = Object Query Language: queries an OO database with an ODL schema, in a manner similar to SQL.
ODL Overview

Class declarations include:

1. Name for the interface.
2. Key declaration(s), which are optional.
3. *Extent* declaration = name for the set of currently existing objects of a class.
4. *Element* declarations. An element is an attribute, a relationship, or a method.
ODL Class Declarations

class <name> { 
  elements = attributes, relationships, methods
}

Element Declarations
attribute <type> <name>;
relationship <rangetype> <name>;

• Relationships involve objects; attributes involve non-object values, e.g., integers.

Method Example
float gpa(in string) raises(noGrades)

• float = return type.
• in: indicates the argument (a student name, presumably) is read-only.
  ◆ Other options: out, inout.
• noGrades is an exception that can be raised by method gpa.
ODL Relationships

• Only binary relations supported.
  ◆ Multiway relationships require a “connecting” class, as discussed for E/R model.

• Relationships come in inverse pairs.
  ◆ Example: “Sells” between beers and bars is represented by a relationship in bars, giving the beers sold, and a relationship in beers giving the bars that sell it.

• Many-many relationships have a set type (called a collection type) in each direction.

• Many-one relationships have a set type for the one, and a simple class name for the many.

• One-one relations have classes for both.
Beers-Bars-Drinkers Example

class Beers {
  attribute string name;
  attribute string manf;
  relationship Set<Bars> servedAt
    inverse Bars::serves;
  relationship Set<Drinkers> fans
    inverse Drinkers::likes;
}

- An element from another class is indicated by
  \textlangle\text{class}\rangle::

- Form a set type with Set\langle\text{type}\rangle.
```java
class Bars {
    attribute string name;
    attribute Struct Addr
        {string street, string city, int zip}
        address;
    attribute Enum Lic {full, beer, none}
        licenseType;
    relationship Set<Drinkers> customers
        inverse Drinkers::frequents;
    relationship Set<Beers> serves
        inverse Beers::servedAt;
}

• Structured types have names and bracketed lists of field-type pairs.
• Enumerated types have names and bracketed lists of values.
```
class Drinkers {
    attribute string name;
    attribute Struct Bars::Addr address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
}

• Note reuse of Addr type.
ODL Type System

- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors: struct for structures and four collection types: Set, Bag, List, Array, and Dictionary.
- Relationship types may only be classes or a collection of a class.
Many-One Relationships

Don’t use a collection type for relationship in the “many” class.

Example: Drinkers Have Favorite Beers

class Drinkers {
    attribute string name;
    attribute Struct Bars::Addr address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Beers favoriteBeer
        inverse Beers::realFans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
}

• Also add to Beers:
  relationship Set<Drinkers> realFans
      inverse Drinkers::favoriteBeer;
Example: Multiway Relationship

Consider a 3-way relationship bars-beers-prices. We have to create a connecting class BBP.

```java
class Prices {
    attribute real price;
    relationship Set<BBP> toBBP
        inverse BBP::thePrice;
}
class BBP {
    relationship Bars theBar inverse ...
    relationship Beers theBeer inverse ...
    relationship Prices thePrice
        inverse Prices::toBBP;
}
```

- Inverses for `theBar`, `theBeer` must be added to `Bars`, `Beers`.
- Better in this special case: make no `Prices` class; make `price` an attribute of `BBP`.
- Notice that keys are optional.
  - `BBP` has no key, yet is not “weak.” Object identity suffices to distinguish different `BBP` objects.
Roles in ODL

Names of relationships handle “roles.”

Example: Spouses and Drinking Buddies

class Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
        address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
    relationship Drinkers husband
        inverse wife;
    relationship Drinkers wife
        inverse husband;
    relationship Set<Drinkers> buddies
        inverse buddies;
}

• Notice that Drinkers:: is optional when the inverse is a relationship of the same class.