Chapter 5: Other Relational Languages

- Query-by-Example (QBE)
- Datalog

Query-by-Example (QBE)

- Basic Structure
- Queries on One Relation
- Queries on Several Relations
- The Condition Box
- The Result Relation
- Ordering the Display of Tuples
- Aggregate Operations
- Modification of the Database
QBE — Basic Structure

- A graphical query language which is based (roughly) on the domain relational calculus
- Two dimensional syntax – system creates templates of relations that are requested by users
- Queries are expressed “by example”

QBE Skeleton Tables for the Bank
Example

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customer</th>
<th>customer-name</th>
<th>customer-street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QBE Skeleton Tables (Cont.)

<table>
<thead>
<tr>
<th>borrower</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Queries on One Relation

Find all loan numbers at the Perryridge branch.

- \_x is a variable (optional; can be omitted in above query)
- P. means print (display)
- duplicates are removed by default
- To retain duplicates use P.ALL

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_x</td>
<td>Perryridge</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.ALL</td>
<td>Perryridge</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Queries on One Relation (Cont.)

- Display full details of all loans
  - Method 1:
    - Method 2: Shorthand notation

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>P._x</td>
<td>P._y</td>
<td>P._z</td>
<td></td>
</tr>
</tbody>
</table>

- Find names of all branches that are not located in Brooklyn

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>→ Brooklyn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Find the loan number of all loans with a loan amount of more than $700

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td>&gt;700</td>
</tr>
</tbody>
</table>
Queries on One Relation (Cont.)

- Find the loan numbers of all loans made jointly to Smith and Jones.
  
<table>
<thead>
<tr>
<th>borrower</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Smith”</td>
<td>“Jones”</td>
<td>P._x</td>
</tr>
</tbody>
</table>

- Find all customers who live in the same city as Jones.
  
<table>
<thead>
<tr>
<th>customer</th>
<th>customer-name</th>
<th>customer-street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>P._x</td>
<td>Jones</td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>

Queries on Several Relations

- Find the names of all customers who have a loan from the Perryridge branch.
  
<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>_x</td>
<td></td>
<td>Perryridge</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>borrower</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>P._y</td>
<td></td>
<td>_x</td>
</tr>
</tbody>
</table>
Queries on Several Relations (Cont.)

Find the names of all customers who have both an account and a loan at the bank.

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>P._x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>borrower</td>
<td>customer-name</td>
<td>loan-number</td>
</tr>
<tr>
<td>_x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Negation in QBE

Find the names of all customers who have an account at the bank, but do not have a loan from the bank.

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>P._x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>borrower</td>
<td>customer-name</td>
<td>loan-number</td>
</tr>
<tr>
<td>_\neg_x</td>
<td>_x</td>
<td></td>
</tr>
</tbody>
</table>

\neg means “there does not exist”
**Negation in QBE (Cont.)**

- Find all customers who have at least two accounts.

```
<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_x</td>
<td>_x</td>
<td>y</td>
</tr>
<tr>
<td>_x</td>
<td></td>
<td>¬y</td>
</tr>
</tbody>
</table>
```

¬ means “not equal to”

**The Condition Box**

- Allows the expression of constraints on domain variables that are either inconvenient or impossible to express within the skeleton tables.
- Complex conditions can be used in condition boxes
- E.g. Find the loan numbers of all loans made to Smith, to Jones, or to both jointly

```
<table>
<thead>
<tr>
<th>borrower</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>_n</td>
<td></td>
<td>P_x</td>
</tr>
</tbody>
</table>

conditions

| _n = Smith or _n = Jones |
```
QBE supports an interesting syntax for expressing alternative values.

### Condition Box (Cont.)

Find all account numbers with a balance between $1,300 and $1,500.

- \( x \geq 1300 \)
- \( x \leq 1500 \)

Find all account numbers with a balance between $1,300 and $2,000 but not exactly $1,500.

- \( x = (\geq 1300 \text{ and } \leq 2000 \text{ and } \neg 1500) \)
Condition Box (Cont.)

Find all branches that have assets greater than those of at least one branch located in Brooklyn

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{-x}</td>
<td></td>
<td>Brooklyn</td>
<td>_y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_z</td>
</tr>
</tbody>
</table>

conditions

_y > _z

The Result Relation

Find the customer-name, account-number, and balance for all customers who have an account at the Perryridge branch.

- We need to:
  - Join depositor and account.
  - Project customer-name, account-number and balance.
- To accomplish this we:
  - Create a skeleton table, called result, with attributes customer-name, account-number, and balance.
  - Write the query.
The Result Relation (Cont.)

- The resulting query is:

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Perryridge</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>customer-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

Ordering the Display of Tuples

- AO = ascending order; DO = descending order.
- E.g. list in ascending alphabetical order all customers who have an account at the bank

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.AO.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- When sorting on multiple attributes, the sorting order is specified by including with each sort operator (AO or DO) an integer surrounded by parentheses.
- E.g. List all account numbers at the Perryridge branch in ascending alphabetic order with their respective account balances in descending order.

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.AO(1).</td>
<td>Perryridge</td>
<td>P.DO(2).</td>
<td></td>
</tr>
</tbody>
</table>
Aggregate Operations

- The aggregate operators are AVG, MAX, MIN, SUM, and CNT
- The above operators must be postfixed with “ALL” (e.g., SUM.ALL or AVG.ALL_x) to ensure that duplicates are not eliminated.
- E.g. Find the total balance of all the accounts maintained at the Perryridge branch.

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Perryridge</td>
<td>P.SUM.ALL</td>
</tr>
</tbody>
</table>

Aggregate Operations (Cont.)

- UNQ is used to specify that we want to eliminate duplicates
- Find the total number of customers having an account at the bank.

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>P.CNT.UNQ.</td>
</tr>
</tbody>
</table>
Query Examples

- Find the average balance at each branch.

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PG.</td>
<td>P.AVG.ALL...x</td>
</tr>
</tbody>
</table>

- The “G” in “P.G” is analogous to SQL’s group by construct.
- The “ALL” in the “P.AVG.ALL” entry in the balance column ensures that all balances are considered.
- To find the average account balance at only those branches where the average account balance is more than $1,200, we simply add the condition box:

```
conditions
AVG.ALL...x > 1200
```

Query Example

- Find all customers who have an account at all branches located in Brooklyn.

  - Approach: for each customer, find the number of branches in Brooklyn at which they have accounts, and compare with total number of branches in Brooklyn.
  - QBE does not provide subquery functionality, so both above tasks have to be combined in a single query.
  - Can be done for this query, but there are queries that require subqueries and cannot be expressed in QBE always be done.

- In the query on the next page

  - CNT.UNQ.ALL...w specifies the number of distinct branches in Brooklyn. Note: The variable _w is not connected to other variables in the query.
  - CNT.UNQ.ALL...z specifies the number of distinct branches in Brooklyn at which customer x has an account.
Modification of the Database – Deletion

- Deletion of tuples from a relation is expressed by use of a D. command. In the case where we delete information in only some of the columns, null values, specified by \(-\), are inserted.
- Delete customer Smith

<table>
<thead>
<tr>
<th>customer</th>
<th>customer-name</th>
<th>customer-street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.</td>
<td>Smith</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Delete the branch-city value of the branch whose name is “Perryridge”.

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>D.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Delete all loans with a loan amount between $1300 and $1500.

For consistency, we have to delete information from loan and borrower tables.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{loan} & \text{loan-number} & \text{branch-name} & \text{amount} \\
\hline
D. & .y & .x & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{ borrower} & \text{customer-name} & \text{loan-number} \\
\hline
D. & .y & .y \\
\hline
\end{array}
\]

conditions

\[
.x = (\geq 1300 \text{ and } \leq 1500)
\]

Delete all accounts at branches located in Brooklyn.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{account} & \text{account-number} & \text{branch-name} & \text{balance} \\
\hline
D. & .y & .x & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{depositor} & \text{customer-name} & \text{account-number} \\
\hline
D. & .y & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{branch} & \text{branch-name} & \text{branch-city} & \text{assets} \\
\hline
\_x & \text{Brooklyn} & \\
\hline
\end{array}
\]
Modification of the Database – Insertion

- Insertion is done by placing the I. operator in the query expression.
- Insert the fact that account A-9732 at the Perryridge branch has a balance of $700.

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>A-9732</td>
<td>Perryridge</td>
<td>700</td>
</tr>
</tbody>
</table>

Modification of the Database – Insertion (Cont.)

- Provide as a gift for all loan customers of the Perryridge branch, a new $200 savings account for every loan account they have, with the loan number serving as the account number for the new savings account.

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>_x</td>
<td>Perryridge</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>_y</td>
<td>_x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>_x</td>
<td></td>
<td>Perryridge</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>borrower</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>_y</td>
<td></td>
<td>_x</td>
</tr>
</tbody>
</table>
Modification of the Database – Updates

- Use the U. operator to change a value in a tuple without changing all values in the tuple. QBE does not allow users to update the primary key fields.
- Update the asset value of the Perryridge branch to $10,000,000.

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>U.10000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Increase all balances by 5 percent.

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U._x * 1.05</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microsoft Access QBE

- Microsoft Access supports a variant of QBE called Graphical Query By Example (GQBE)
- GQBE differs from QBE in the following ways
  - Attributes of relations are listed vertically, one below the other, instead of horizontally
  - Instead of using variables, lines (links) between attributes are used to specify that their values should be the same.
    - Links are added automatically on the basis of attribute name, and the user can then add or delete links
    - By default, a link specifies an inner join, but can be modified to specify outer joins.
  - Conditions, values to be printed, as well as group by attributes are all specified together in a box called the design grid
An Example Query in Microsoft Access QBE

- Example query: Find the customer-name, account-number and balance for all accounts at the Perryridge branch

An Aggregation Query in Access QBE

- Find the name, street and city of all customers who have more than one account at the bank
Aggregation in Access QBE

- The row labeled Total specifies
  - which attributes are group by attributes
  - which attributes are to be aggregated upon (and the aggregate function).
  - For attributes that are neither group by nor aggregated, we can still specify conditions by selecting where in the Total row and listing the conditions below
- As in SQL, if group by is used, only group by attributes and aggregate results can be output

Datalog

- Basic Structure
- Syntax of Datalog Rules
- Semantics of Nonrecursive Datalog
- Safety
- Relational Operations in Datalog
- Recursion in Datalog
- The Power of Recursion
**Basic Structure**

- Prolog-like logic-based language that allows recursive queries; based on first-order logic.
- A Datalog program consists of a set of **rules** that define views.
- Example: define a view relation \( v1 \) containing account numbers and balances for accounts at the Perryridge branch with a balance of over $700.

\[
v1(A, B) :– \text{account}(A, \text{“Perryridge”}, B), B > 700.
\]

- Retrieve the balance of account number “A-217” in the view relation \( v1 \).

\[
\]

- To find account number and balance of all accounts in \( v1 \) that have a balance greater than 800

\[
? v1(A, B), B > 800
\]

**Example Queries**

- Each rule defines a set of tuples that a view relation must contain.
  - E.g. \( v1(A, B) :– \text{account}(A, \text{“Perryridge”}, B), B > 700 \) is read as
    - for all \( A, B \)
    - if \( (A, \text{“Perryridge”}, B) \in \text{account and } B > 700 \)
    - then \( (A, B) \in v1 \)

- The set of tuples in a view relation is then defined as the union of all the sets of tuples defined by the rules for the view relation.

- Example:
  - \( \text{interest-rate}(A, 5) :– \text{account}(A, N, B), B < 10000 \)
  - \( \text{interest-rate}(A, 6) :– \text{account}(A, N, B), B >= 10000 \)
Negation in Datalog

- Define a view relation \( c \) that contains the names of all customers who have a deposit but no loan at the bank:

\[
c(N) :\neg \text{depositor}(N, A), \text{not is-borrower}(N).
\]

\[
is\text{-borrower}(N) :\neg \text{borrower}(N, L).
\]

**NOTE:** using **not borrower \((N, L)\)** in the first rule results in a different meaning, namely there is some loan \( L \) for which \( N \) is not a borrower.

- To prevent such confusion, we require all variables in negated "predicate" to also be present in non-negated predicates.

---

Named Attribute Notation

- Datalog rules use a positional notation, which is convenient for relations with a small number of attributes.

- It is easy to extend Datalog to support named attributes.

  - E.g., \( v1 \) can be defined using named attributes as

\[
v1(\text{account-number } A, \text{balance } B) :\neg
\]

\[
\text{account(}\text{account-number } A, \text{branch-name } \text{“Perryridge”}, \text{balance } B), B > 700.
\]
Formal Syntax and Semantics of Datalog

- We formally define the syntax and semantics (meaning) of Datalog programs, in the following steps:
  - We define the syntax of predicates, and then the syntax of rules
  - We define the semantics of individual rules
  - We define the semantics of non-recursive programs, based on a layering of rules
  - It is possible to write rules that can generate an infinite number of tuples in the view relation. To prevent this, we define what rules are “safe”. Non-recursive programs containing only safe rules can only generate a finite number of answers.
  - It is possible to write recursive programs whose meaning is unclear. We define what recursive programs are acceptable, and define their meaning.

Syntax of Datalog Rules

- A **positive literal** has the form
  \[ p(t_1, t_2, \ldots, t_n) \]
  - \( p \) is the name of a relation with \( n \) attributes
  - each \( t_i \) is either a constant or variable
- A **negative literal** has the form
  \[ \text{not } p(t_1, t_2, \ldots, t_n) \]

- Comparison operations are treated as positive predicates
  - E.g. \( X > Y \) is treated as a predicate \( >(X,Y) \)
  - “\( > \)” is conceptually an (infinite) relation that contains all pairs of values such that the first value is greater than the second value
- Arithmetic operations are also treated as predicates
  - E.g. \( A = B + C \) is treated as \( +(B, C, A) \), where the relation “\( + \)” contains all triples such that the third value is the sum of the first two
Syntax of Datalog Rules (Cont.)

- **Rules** are built out of literals and have the form:
  \[ p(t_1, t_2, \ldots, t_n) : – L_1, L_2, \ldots, L_m. \]
  - each of the \( L_i \)'s is a literal
  - head – the literal \( p(t_1, t_2, \ldots, t_n) \)
  - body – the rest of the literals
- A **fact** is a rule with an empty body, written in the form:
  \[ p(v_1, v_2, \ldots, v_n). \]
  - indicates tuple \( (v_1, v_2, \ldots, v_n) \) is in relation \( p \)
- A **Datalog program** is a set of rules

Semantics of a Rule

- A **ground instantiation** of a rule (or simply **instantiation**) is the result of replacing each variable in the rule by some constant.
  - Eg. Rule defining \( v1 \)
    \[ v1(A,B) : – \text{account}(A, "Perryridge", B), B > 700. \]
  - An instantiation above rule:
    \[ v1("A-217", 750) : – \text{account}("A-217", "Perryridge", 750), \]
    \[ 750 > 700. \]
- The body of rule instantiation \( R' \) is **satisfied** in a set of facts (database instance) \( I \) if
  1. For each positive literal \( q(v_1, \ldots, v_{m_1}) \) in the body of \( R' \), \( I \) contains the fact \( q(v_1, \ldots, v_{m_1}) \).
  2. For each negative literal **not** \( q(v_1, \ldots, v_{m_2}) \) in the body of \( R' \), \( I \) does not contain the fact \( q(v_1, \ldots, v_{m_2}) \).
Semantics of a Rule (Cont.)

- We define the set of facts that can be inferred from a given set of facts \( l \) using rule \( R \) as:
  \[
  \text{infer}(R, l) = \{ p(t_1, ..., t_n) \mid \text{there is a ground instantiation } R' \text{ of } R \\
  \text{where } p(t_1, ..., t_n) \text{ is the head of } R', \text{ and} \\
  \text{the body of } R' \text{ is satisfied in } l \}
  \]

- Given a set of rules \( \mathcal{R} = \{ R_1, R_2, ..., R_n \} \), we define
  \[
  \text{infer}(\mathcal{R}, l) = \text{infer}(R_1, l) \cup \text{infer}(R_2, l) \cup ... \cup \text{infer}(R_n, l)
  \]

Layering of Rules

- Define the interest on each account in Perryridge
  
  \[
  \begin{align*}
  \text{interest}(A, l) & :\text{ perryridge-account}(A, B), \\
  \text{interest-rate}(A, R), & \quad l = B \times R/100. \\
  \text{perryridge-account}(A, B) & :\text{account}(A, "Perryridge", B). \\
  \text{interest-rate}(A, 0) & :\text{account}(N, A, B), B < 2000. \\
  \text{interest-rate}(A, 5) & :\text{account}(N, A, B), B > 2000.
  \end{align*}
  \]

- Layering of the view relations

- [Diagram of layering of views]
Layering Rules (Cont.)

Formally:

- A relation is a layer 1 if all relations used in the bodies of rules defining it are stored in the database.
- A relation is a layer 2 if all relations used in the bodies of rules defining it are either stored in the database, or are in layer 1.
- A relation \( p \) is in layer \( i + 1 \) if
  - it is not in layers 1, 2, ..., \( i \)
  - all relations used in the bodies of rules defining a \( p \) are either stored in the database, or are in layers 1, 2, ..., \( i \)

Semantics of a Program

Let the layers in a given program be 1, 2, ..., \( n \). Let \( \mathcal{R}_i \) denote the set of all rules defining view relations in layer \( i \).

- Define \( I_0 = \) set of facts stored in the database.
- Recursively define \( I_{i+1} = I_i \cup \text{infer}(\mathcal{R}_{i+1}, I_i) \)
- The set of facts in the view relations defined by the program (also called the semantics of the program) is given by the set of facts \( I_n \) corresponding to the highest layer \( n \).

Note: Can instead define semantics using view expansion like in relational algebra, but above definition is better for handling extensions such as recursion.
Safety

- It is possible to write rules that generate an infinite number of answers.

\[
\text{gt}(X, Y) :- X > Y
\]
\[
\text{not-in-loan}(B, L) :- \text{not loan}(B, L)
\]

To avoid this possibility Datalog rules must satisfy the following conditions.

- Every variable that appears in the head of the rule also appears in a non-arithmetic positive literal in the body of the rule.
  - This condition can be weakened in special cases based on the semantics of arithmetic predicates, for example to permit the rule
    \[
    p(A) :- q(B), A = B + 1
    \]
- Every variable appearing in a negative literal in the body of the rule also appears in some positive literal in the body of the rule.

Relational Operations in Datalog

- Project out attribute \textit{account-name} from account.

\[
\text{query}(A) :- \text{account}(A, N, B).
\]

- Cartesian product of relations \(r_1\) and \(r_2\).

\[
\text{query}(X_1, X_2, ..., X_n, Y_1, Y_2, ..., Y_m) :-
\text{r}_1(X_1, X_2, ..., X_n), \text{r}_2(Y_1, Y_2, ..., Y_m).
\]

- Union of relations \(r_1\) and \(r_2\).

\[
\text{query}(X_1, X_2, ..., X_n) :-\text{r}_1(X_1, X_2, ..., X_n),
\text{query}(X_1, X_2, ..., X_n) :-\text{r}_2(X_1, X_2, ..., X_n),
\]

- Set difference of \(r_1\) and \(r_2\).

\[
\text{query}(X_1, X_2, ..., X_n) :-\text{r}_1(X_1, X_2, ..., X_n), \text{not r}_2(X_1, X_2, ..., X_n).
\]
Updates in Datalog

- Some Datalog extensions support database modification using + or – in the rule head to indicate insertion and deletion.
- E.g. to transfer all accounts at the Perryridge branch to the Johnstown branch, we can write
  
  ```datalog
  ```

Recursion in Datalog

- Suppose we are given a relation
  
  `manager(X, Y)`
  
  containing pairs of names X, Y such that Y is a manager of X (or equivalently, X is a direct employee of Y).
- Each manager may have direct employees, as well as indirect employees
  
  ⚫ Indirect employees of a manager, say Jones, are employees of people who are direct employees of Jones, or recursively, employees of people who are indirect employees of Jones
- Suppose we wish to find all (direct and indirect) employees of manager Jones. We can write a recursive Datalog program.
  
  ```datalog
  empl-jones(X) :- manager(X, Jones).
  empl-jones(X) :- manager(X, Y), empl-jones(Y).
  ```
Semantics of Recursion in Datalog

- Assumption (for now): program contains no negative literals
- The view relations of a recursive program containing a set of rules $R$ are defined to contain exactly the set of facts $l$ computed by the iterative procedure $Datalog-Fixpoint$

**procedure** $Datalog-Fixpoint$

$l = \text{set of facts in the database}$

repeat

$\text{Old}_l = l$

$l = l \cup \text{infer}(R, l)$

until $l = \text{Old}_l$

- At the end of the procedure, $\text{infer}(R, l) \subseteq l$
  - $\text{infer}(R, l) = l$ if we consider the database to be a set of facts that are part of the program
- $l$ is called a fixed point of the program.

Example of Datalog-FixPoint Iteration

<table>
<thead>
<tr>
<th>employee-name</th>
<th>manager-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alon</td>
<td>Barinsky</td>
</tr>
<tr>
<td>Barinsky</td>
<td>Estovar</td>
</tr>
<tr>
<td>Corbin</td>
<td>Duarte</td>
</tr>
<tr>
<td>Duarte</td>
<td>Jones</td>
</tr>
<tr>
<td>Estovar</td>
<td>Jones</td>
</tr>
<tr>
<td>Jones</td>
<td>Klinger</td>
</tr>
<tr>
<td>Rensal</td>
<td>Klinger</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Tuples in emp-l-jones</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(Duarte), (Estovar)</td>
</tr>
<tr>
<td>2</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin)</td>
</tr>
<tr>
<td>3</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)</td>
</tr>
<tr>
<td>4</td>
<td>(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)</td>
</tr>
</tbody>
</table>
A More General View

Create a view relation `empl` that contains every tuple `(X, Y)` such that `X` is directly or indirectly managed by `Y`.

- `empl(X, Y) :- manager(X, Y).`
- `empl(X, Y) :- manager(X, Y), empl(Z, Y)`

Find the direct and indirect employees of Jones.


The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.

  ★ Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of manager with itself
  - ✔ This can give only a fixed number of levels of managers
  - ✔ Given a program we can construct a database with a greater number of levels of managers on which the program will not work
Recursion in SQL

- SQL:1999 permits recursive view definition
- E.g. query to find all employee-manager pairs

```sql
with recursive empl (emp, mgr) as (  
  select emp, mgr  
  from manager  
  union  
  select emp, empl.mgr  
  from manager, empl  
  where manager.mgr = empl.emp  
)  
select *  
from empl
```

Monotonicity

- A view $V$ is said to be monotonic if given any two sets of facts $I_1$ and $I_2$ such that $I_1 \subseteq I_2$, then $E_v(I_1) \subseteq E_v(I_2)$, where $E_v$ is the expression used to define $V$.
- A set of rules $R$ is said to be monotonic if $I_1 \subseteq I_2$ implies $\text{infer}(R, I_1) \subseteq \text{infer}(R, I_2)$.
- Relational algebra views defined using only the operations: \( \Pi \), \( \sigma \), \( \times \), \( \cup \), \( \cap \), and \( \rho \) (as well as operations like natural join defined in terms of these operations) are monotonic.
- Relational algebra views defined using – may not be monotonic.
- Similarly, Datalog programs without negation are monotonic, but Datalog programs with negation may not be monotonic.
Non-Monotonicity

- Procedure *Datalog-Fixpoint* is sound provided the rules in the program are monotonic.
  - Otherwise, it may make some inferences in an iteration that cannot be made in a later iteration. E.g. given the rules
    
    \[ a \leftarrow \text{not } b. \]
    
    \[ b \leftarrow c. \]
    
    \[ c. \]

    Then \( a \) can be inferred initially, before \( b \) is inferred, but not later.

- We can extend the procedure to handle negation so long as the program is “stratified”: intuitively, so long as negation is not mixed with recursion.

Stratified Negation

- A Datalog program is said to be stratified if its predicates can be given layer numbers such that
  - For all positive literals, say \( q_i \), in the body of any rule with head, say, \( p \)
    \[ p(\ldots) \leftarrow \ldots, q_i(\ldots), \ldots \]
    then the layer number of \( p \) is greater than or equal to the layer number of \( q_i \).
  - Given any rule with a negative literal
    \[ p(\ldots) \leftarrow \ldots, \text{not } q_i(\ldots), \ldots \]
    then the layer number of \( p \) is strictly greater than the layer number of \( q_i \).

- Stratified programs do not have recursion mixed with negation.

- We can define the semantics of stratified programs layer by layer, from the bottom-most layer, using fixpoint iteration to define the semantics of each layer.

  - Since lower layers are handled before higher layers, their facts will not change, so each layer is monotonic once the facts for lower layers are fixed.
Non-Monotonicity (Cont.)

- There are useful queries that cannot be expressed by a stratified program
  - E.g., given information about the number of each subpart in each part, in a part-subpart hierarchy, find the total number of subparts of each part.
  - A program to compute the above query would have to mix aggregation with recursion
  - However, so long as the underlying data (part-subpart) has no cycles, it is possible to write a program that mixes aggregation with recursion, yet has a clear meaning
  - There are ways to evaluate some such classes of non-stratified programs

Forms and Graphical User Interfaces

- Most naive users interact with databases using form interfaces with graphical interaction facilities
  - Web interfaces are the most common kind, but there are many others
  - Forms interfaces usually provide mechanisms to check for correctness of user input, and automatically fill in fields given key values
  - Most database vendors provide convenient mechanisms to create forms interfaces, and to link form actions to database actions performed using SQL
Report Generators

Report generators are tools to generate human-readable summary reports from a database

- They integrate database querying with creation of formatted text and graphical charts
- Reports can be defined once and executed periodically to get current information from the database.
- Example of report (next page)
- Microsoft’s Object Linking and Embedding (OLE) provides a convenient way of embedding objects such as charts and tables generated from the database into other objects such as Word documents.

A Formatted Report

Acme Supply Company Inc.
Quarterly Sales Report

Period: Jan. 1 to March 31, 2001

<table>
<thead>
<tr>
<th>Region</th>
<th>Category</th>
<th>Sales</th>
<th>Subtotal</th>
</tr>
</thead>
<tbody>
<tr>
<td>North</td>
<td>Computer Hardware</td>
<td>1,000,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Computer Software</td>
<td>500,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All categories</td>
<td></td>
<td>1,500,000</td>
</tr>
<tr>
<td>South</td>
<td>Computer Hardware</td>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Computer Software</td>
<td>400,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All categories</td>
<td></td>
<td>600,000</td>
</tr>
<tr>
<td></td>
<td><strong>Total Sales</strong></td>
<td></td>
<td><strong>2,100,000</strong></td>
</tr>
</tbody>
</table>
QBE Skeleton Tables for the Bank Example

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customer</th>
<th>customer-name</th>
<th>customer-street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>borrower</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>depositor</th>
<th>customer-name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Example Query in Microsoft Access QBE

An Aggregation Query in Microsoft Access QBE
### The account Relation

<table>
<thead>
<tr>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-217</td>
<td>Perryridge</td>
<td>750</td>
</tr>
</tbody>
</table>

### The v1 Relation

<table>
<thead>
<tr>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>A-217</td>
<td>750</td>
</tr>
</tbody>
</table>
### Result of infer\( (R, I) \)

<table>
<thead>
<tr>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>A-217</td>
<td>750</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td>Perryridge</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan</th>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>_x</td>
<td></td>
<td>Perryridge</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch</th>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>Capital</td>
<td>Queens</td>
<td></td>
</tr>
</tbody>
</table>

### conditions

\[ y \geq 2 \times z \]