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

<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>
<tr>
<td>customer</td>
<td>customer-name</td>
<td>customer-street</td>
<td>customer-city</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loan</td>
<td>loan-number</td>
<td>branch-name</td>
<td>amount</td>
</tr>
<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>
<tr>
<td>account</td>
<td>account-number</td>
<td>branch-name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>depositor</td>
<td>customer-name</td>
<td>account-number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Queries on One Relation

- Find all loan numbers at 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>

- Method 1:
- Method 2: Shorthand notation

Queries on One Relation (Cont.)

- Display full details of all loans

<table>
<thead>
<tr>
<th>loan department</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 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 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>

- 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></td>
<td>~ Brooklyn</td>
<td></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>P, x</td>
<td>“Smith”</td>
<td>P, x</td>
</tr>
<tr>
<td>_x</td>
<td>“Jones”</td>
<td>_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>_y</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>P, x</td>
<td>Perryridge</td>
<td>_x</td>
</tr>
</tbody>
</table>

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

~ means “there does not exist”

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>_x</td>
<td></td>
<td>_x</td>
</tr>
</tbody>
</table>

conditions: _x = Smith or _x = Jones

Condition Box (Cont.)

- QBE supports an interesting syntax for expressing alternative values.

<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>_x</td>
<td>_x</td>
<td>_x</td>
</tr>
</tbody>
</table>

conditions: _x = (Brooklyn or Queens)

Condition Box (Cont.)

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

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>P, x</td>
<td>_x</td>
<td>_x</td>
<td>_x</td>
</tr>
</tbody>
</table>

conditions: _x = 1,300 and _x = 1,500

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

<table>
<thead>
<tr>
<th>account</th>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>P, x</td>
<td>_x</td>
<td>_x</td>
<td>_x</td>
</tr>
</tbody>
</table>

conditions: _x = ( $1,300 and $2,000 and ~ _x = 1,500)
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>
</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.

1. We need:
   - Join depositor and account.
   - Project customer-name and account-number.

2. 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>

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>. y</td>
<td>Perryridge</td>
<td>P.AO(1)</td>
<td></td>
</tr>
<tr>
<td>. y</td>
<td>Perryridge</td>
<td>PDO(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>PSUM.AX</td>
</tr>
</tbody>
</table>

The Result Relation (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>
<th>CNT.UNQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.CNT.UNQ</td>
<td></td>
<td></td>
<td></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>PA.x</td>
<td>PAVG.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:

<table>
<thead>
<tr>
<th>conditions</th>
<th>AVG.ALL.x &gt; 1200</th>
</tr>
</thead>
</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>
<th>P.CNT.UNQ</th>
</tr>
</thead>
</table>

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

```
customer | customer-name | customer-street | customer-city  
D.        | Smith         |                |               
```

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

```
branch    | branch-name   | branch-city    | assets        
Perryridge |                | Brooklyn       |               
```

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.

```
account | account-number | branch-name    | balance     
L.       | A-9732         | Perryridge     | 700         
```

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.

```
branch   | branch-name   | branch-city    | assets     
Perryridge|                |              | U.10000000 
```

- Increase all balances by 5 percent.

```
account | account-number | branch-name    | balance    
U.       | z*1.05         |                |             
```
An Example Query in Microsoft Access QBE

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

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

Example Queries

Each rule defines a set of tuples that a view relation must contain.

- E.g. v1(A, B) :- account(A, "Perryridge", B), B > 700 is read as
  - for all A, B
  - if (A, "Perryridge", B) ∈ account and B > 700
  - then (A, B) ∈ 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:

interest-rate(A, B) :- account(A, N, B), B < 10000
interest-rate(A, B) :- 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) :- depositor(N, A), not is-borrower(N).

is-borrower(N) :- 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(account-number A, balance B) :-
account(account-number A, branch-name "Perryridge", balance B), B > 700.
**Syntax of Datalog Rules**

- **A positive literal** has the form:
  \[ p(t_1, t_2, ..., t_n) \]
  - \( p \) is the name of a relation with \( n \) attributes
  - each \( t \) is either a constant or variable

- **A negative literal** has the form:
  \[ \neg p(t_1, t_2, ..., t_n) \]
  - Comparison operations are treated as positive predicates
    - E.g., \( X > Y \) is treated as a predicate \( X > Y \)
    - \( \neg X \) 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

**Semantics of a Rule (Cont.)**

- **Ground instantiation** of a rule (or simply instantiation) is the result of replacing each variable in the rule by some constant.
  - E.g., Rule defining \( v1 \):
    \[ v1(A, B) \rightarrow \text{account}(A, "Perryridge", B), B > 700. \]
  - An instantiation above rule:
    \[ \nu 1(A, 217), \nu 2(750) \rightarrow \text{account}(A, "Perryridge", 750), 750 > 700. \]

- The body of rule instantiation \( R' \) is satisfied in a set of facts (database instance) \( I \) if:
  1. For each positive literal \( q(v_1, ..., v_m) \) in the body of \( R' \), \( I \) contains the fact \( q(v_1, ..., v_m) \)
  2. For each negative literal \( \neg q(v_1, ..., v_m) \) in the body of \( R' \), \( I \) does not contain the fact \( q(v_1, ..., v_m) \)

**Layering Rules (Cont.)**

- 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 \( R \) is in layer \( i + 1 \) if:
  - it is not in layers \( 1, 2, ..., i \)
  - all relations used in the bodies of rules defining \( R \) 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 \( R_i \) denote the set of all rules defining view relations in layer \( i \).

- Define \( I_0 \) as set of facts stored in the database.
- Recursively define \( I_i = I_{i-1} \cup \text{infer}(R_i, I_{i-1}) \)
- 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.

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

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

1. Every variable that appears in the head of the rule also appears in a non-arithmetic positive literal in the body of the rule.
2. 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.

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

\[
\text{+ account}(A, "Perryridge", B) : \rightarrow \text{account}(A, "Johnstown", B).
\]

Relational Operations in Datalog

- Project out attribute account-name from account.

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

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

\[
\text{query}(X_1, X_2, ..., X_n) \leftarrow r_1(X_1, X_2, ..., X_n), r_2(Y_1, Y_2, ..., Y_m).
\]

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

\[
\text{query}(X_1, X_2, ..., X_n) \leftarrow r_1(X_1, X_2, ..., X_n), r_2(Y_1, Y_2, ..., Y_m).
\]

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

\[
\text{query}(X_1, X_2, ..., X_n) \leftarrow r_1(X_1, X_2, ..., X_n), \text{not}\, r_2(X_1, X_2, ..., X_n).
\]

Recursion in Datalog

- Suppose we are given a relation \( \text{manager}(X, Y) \)

- 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 employees 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.

\[
\text{empl-jones}(X) : \rightarrow \text{manager}(X, Y).
\]

\[
\text{empl-jones}(X) : \rightarrow \text{manager}(X, Y), \text{empl-jones}(Y).
\]

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 \( \mathfrak{R} \) are defined to contain exactly the set of facts \( \mathfrak{L} \) computed by the iterative procedure Datalog-Fixpoint

\[
\text{procedure} \quad \text{Datalog-Fixpoint} \\
\text{Repeat} \quad \text{Old} = \mathfrak{L}, \mathfrak{L} = \mathfrak{L} \cup \text{infer}(\mathfrak{R}, \mathfrak{L}) \\
\text{Until} \quad \text{Old} = \mathfrak{L}.
\]

- At the end of the procedure, \( \text{infer}(\mathfrak{R}, \mathfrak{L}) = \mathfrak{L} \)

- \( \mathfrak{L} \) is called a fixed point of the program.

Example of Datalog-FixPoint Iteration

A More General View

- Create a view relation \( \text{empl} \) that contains every tuple \((X, Y)\)

- Find the direct and indirect employees of Jones.

- Can define the view \( \text{empl} \) in another way too:

\[
\text{empl}(X, Y) \rightarrow \text{manager}(X, Y).
\]

\[
\text{empl}(X, Y) \rightarrow \text{empl}(X, Z), \text{manager}(Z, Y).
\]

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

```
with recursive empI (emp, mgr) as {
  select emp, mgr
  from manager
  union
  select manager.emp, empI.mgr
  from manager, empI
  where manager.mgr = empI.emp
}
```

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 :- b.
    - b :- 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

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

Monotonicity
- A view V is said to be monotonic if given any two sets of facts I₁ and I₂ such that I₁ ⊆ I₂, then E₁(I₁) ⊆ E₁(I₂), where E₁ is the expression used to define V.
- A set of rules R is said to be monotonic if I₁ ⊆ I₂ implies infer(R, I₁) ⊆ infer(R, I₂).
- Relational algebra views defined using only the operations: [ ], σ, ×, ∩, union, and ρ (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.

Stratified Negation
- A Datalog program is said to be stratified if its predicates can be given layer numbers such that
  1. For all positive literals, say q, in the body of any rule with head, say, p :- q₁,…,qₙ, then the layer number of p is greater than or equal to the layer number of q.
  2. Given any rule with a negative literal p :- q₁,…,qₙ, then the layer number of p is strictly greater than the layer number of q.
- 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.

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

<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>1,500,000</td>
<td></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>600,000</td>
<td></td>
</tr>
<tr>
<td>Total Sales</td>
<td></td>
<td>2,100,000</td>
<td></td>
</tr>
</tbody>
</table>
```
QBE Skeleton Tables for the Bank

Example

An Example Query in Microsoft Access QBE

An Aggregation Query in Microsoft Access QBE

The account Relation

The v1 Relation

Result of infer(R, I)