# 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

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| | | | |

| *customer* | *customer-name* | *customer-street* | *customer-city* |
|---|---|---|---|
| | | | |

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | | | |

# QBE Skeleton Tables (Cont.)

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | | |

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | | | |

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | | |




# Queries on One Relation

- Find all loan numbers at the Perryridge branch.

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | P._*x* | Perryridge | |

- _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

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | P.ALL. | Perryridge | |

# Queries on One Relation (Cont.)

- Display full details of all loans
  - Method 1:

| loan | loan-number | branch-name | amount |
|---|---|---|---|
| | P._x | P._y | P._z |

  - Method 2: Shorthand notation

| loan | loan-number | branch-name | amount |
|---|---|---|---|
| P. | | | |




# Queries on One Relation (Cont.)

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

| loan | loan-number | branch-name | amount |
|---|---|---|---|
| | P. | | >700 |

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

| branch | branch-name | branch-city | assets |
|---|---|---|---|
| | P. | ¬ Brooklyn | |

# Queries on One Relation (Cont.)

- Find the loan numbers of all loans made jointly to Smith and Jones.

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | "Smith" | P._*x* |
| | "Jones" | _*x* |

- Find all customers who live in the same city as Jones

| *customer* | *customer-name* | *customer-street* | *customer-city* |
|---|---|---|---|
| | P._*x* | | _*y* |
| | Jones | | _*y* |



# Queries on Several Relations

- Find the names of all customers who have a loan from the Perryridge branch.

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | _*x* | Perryridge | |

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | P._*y* | _*x* |

# Queries on Several Relations (Cont.)

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

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P._*x* | |

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | _*x* | |



# 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.

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P._*x* | |

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| ¬ | _*x* | |

¬ means “there does not exist”

# Negation in QBE (Cont.)

- Find all customers who have at least two accounts.

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P._*x* | _*y* |
| | _*x* | ¬ _*y* |

¬ 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

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | _*n* | P._*x* |

| *conditions* |
|---|
| _*n* = Smith **or** _*n* = Jones |

# Condition Box (Cont.)

- QBE supports an interesting syntax for expressing alternative values

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| | P. | _*x* | |

| *conditions* |
|---|
| _*x* = (Brooklyn **or** Queens) |



# Condition Box (Cont.)

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

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | P. | | _*x* |

| *conditions* |
|---|
| _*x* ≥ 1300 |
| _*x* ≤ 1500 |

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

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | P. | | _*x* |

| *conditions* |
|---|
| _*x* = ( ≥ 1300 **and** ≤ 2000 **and** ¬ 1500) |

## Condition Box (Cont.)

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

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| | P._x | | _y |
| | | Brooklyn | _z |

| *conditions* |
|---|
| $\_y > \_z$ |

## The Result Relation

- Find the *customer-name*, *account-number,* and *balance* for alll 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:

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | _y | Perryridge | _z |

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | _x | _y |

| *result* | *customer-name* | *account-number* | *balance* |
|---|---|---|---|
| P. | _x | _y | _z |



## 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

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P.AO. | |

- 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.

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | P.AO(1). | Perryridge | P.DO(2). |

# 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.

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | | Perryridge | P.SUM.ALL. |




# 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.

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P.CNT.UNQ. | |

# Query Examples

- Find the average balance at each branch.

| account | account-number | branch-name | balance |
|---|---|---|---|
| | | P.G. | P.AVG.ALL._*x* |

- 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.

# Query Example (Cont.)

| depositor | customer-name | account-number |
|---|---|---|
| | P.G._x | _y |

| account | account-number | branch-name | balance |
|---|---|---|---|
| | _y | _z | |

| branch | branch-name | branch-city | assets |
|---|---|---|---|
| | _z | Brooklyn | |
| | _w | Brooklyn | |

| conditions |
|---|
| CNT.UNQ._z = CNT.UNQ._w |



# 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 | D. | |

# Deletion Query Examples

- Delete all loans with a loan amount between $1300 and $1500.
  - For consistency, we have to delete information from loan and borrower tables

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| D. | _y | | _x |

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| D. | | _y |

| *conditions* |
|---|
| _x = (≥ 1300 **and** ≤ 1500) |




# Deletion Query Examples (Cont.)

- Delete all accounts at branches located in Brooklyn.

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| D. | _y | _x | |

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| D. | | _y |

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| | _x | 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* |
|---|---|---|---|
| I. | A-9732 | Perryridge | 700 |



# 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.

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| I. | _*x* | Perryridge | 200 |

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| I. | _*y* | _*x* |

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | _*x* | Perryridge | |

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | _*y* | _*x* |

## 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._x * 1.05 |

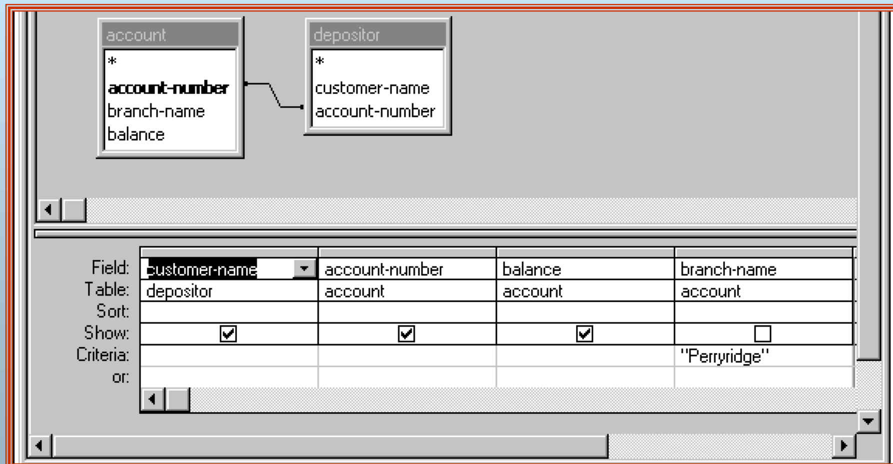



## Microsoft Access QBE

- 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

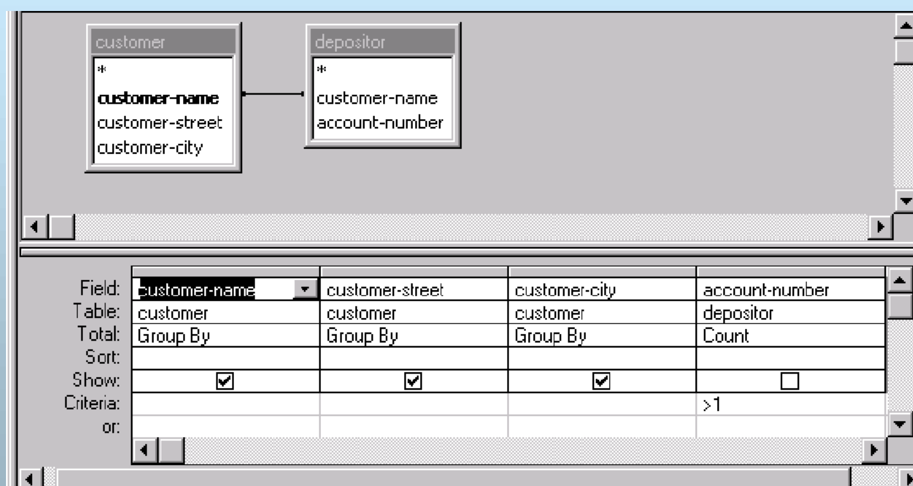
- Example query: Find the *customer-name*, *account-number* and *balance* for all accounts at the Perryridge branch






## 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) :– account(A, "Perryridge", B), B* > 700.
- Retrieve the balance of account number "A-217" in the view relation *v1.*

  *? v1("A-217", B).*
- 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) :– 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,* 5*)* :– *account(A, N, B), B* < 10000
  *interest-rate(A,* 6*)* :– *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.

## 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

  **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:

  $p(t_1, t_2, ..., t_n) :– L_1, L_2, ..., L_m.$

  - each of the $L_i$'s is a literal
  - head – the literal $p(t_1, t_2, ..., t_n)$
  - body – the rest of the literals
- A *fact* is a rule with an empty body, written in the form:

  $p(v_1, v_2, ..., v_n).$

  - indicates tuple $(v_1, v_2, ..., v_n)$ 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)* :– *account (A,*"Perryridge", *B), B* > 700.
  - An instantiation above rule:

    *v1(*"A-217", 750) :–*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 $q_i(v_{i,1}, ..., v_{i,ni})$ in the body of *R', I* contains the fact $q_i(v_{i,1}, ..., v_{i,ni})$.
  2. For each negative literal **not** $q_j(v_{j,1}, ..., v_{j,ni})$ in the body of *R', I* does not contain the fact $q_j(v_{i,1}, ..., v_{j,ni})$.

# Semantics of a Rule (Cont.)

- We define the set of facts that can be **inferred** from a given set of facts $I$ using rule $R$ as:

  $infer(R, I) = \{p(t_1, ..., t_n) \mid$ there is a ground instantiation $R'$ of $R$ where $p(t_1, ..., t_n)$ is the head of $R'$, and the body of $R'$ is satisfied in $I\}$

- *Given an set of rules* $\mathfrak{R} = \{R_1, R_2, ..., R_n\}$, we define

  $infer(\mathfrak{R}, I) = infer(R_1, I) \cup infer(R_2, I) \cup ... \cup infer(R_n, I)$

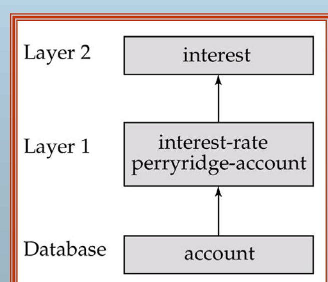


# Layering of Rules

- Define the interest on each account in Perryridge

  *interest(A, I)* :– *perryridge-account(A,B), interest-rate(A,R), I = B * R/*100.
  *perryridge-account(A,B)* :–*account(A,* "Perryridge", *B).*
  *interest-rate(A,*0) :–account(*N, A, B), B* < 2000.
  interest-rate(*A*,5) :–account(N, A, B), *B* > 2000.

- Layering of the view relations



# 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 $\Re_i$ denote the set of all rules defining view relations in layer *i.*

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

  *gt(X, Y)* :– *X* > *Y*
  *not-in-loan(B, L)* :– **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 *account-name* from account.

  *query(A)* :–*account(A, N, B).*

- Cartesian product of relations $r_1$ and $r_2$.

  $query(X_1, X_2, ..., X_{n1}\ Y_1, Y_1, Y_2, ..., Y_m)$ :–
  $r_1(X_1, X_2, ..., X_n), r_2(Y_1, Y_2, ..., Y_m)$.

- *Union of relations* $r_1$ and $r_2$.

  $query(X_1, X_2, ..., X_n)$ :–$r_1(X_1, X_2, ..., X_n)$,
  $query(X_1, X_2, ..., X_n)$ :–$r_2(X_1, X_2, ..., X_n)$,

- Set difference of $r_1$ and $r_2$.

  $query(X_1, X_2, ..., X_n)$ :–$r_1(X_1, X_2, ..., X_n)$,
  **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

    + account(A, "Johnstown", B) :- account (A, "Perryridge", B).

    – account(A, "Perryridge", B) :- account (A, "Perryridge", B)



# 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.

    *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 $\mathfrak{R}$ are defined to contain exactly the set of facts $I$ computed by the iterative procedure *Datalog-Fixpoint*

**procedure** Datalog-Fixpoint
  $I$ = set of facts in the database
  **repeat**
    $Old\_I = I$
    $I = I \cup infer(\mathfrak{R}, I)$
  **until** $I = Old\_I$

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



# Example of Datalog-FixPoint Iteration

| *employee-name* | *manager-name* |
|---|---|
| Alon | Barinsky |
| Barinsky | Estovar |
| Corbin | Duarte |
| Duarte | Jones |
| Estovar | Jones |
| Jones | Klinger |
| Rensal | Klinger |

| Iteration number | Tuples in *empl-jones* |
|---|---|
| 0 | |
| 1 | (Duarte), (Estovar) |
| 2 | (Duarte), (Estovar), (Barinsky), (Corbin) |
| 3 | (Duarte), (Estovar), (Barinsky), (Corbin), (Alon) |
| 4 | (Duarte), (Estovar), (Barinsky), (Corbin), (Alon) |

# 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.

  *? empl(X,* "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

```
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 $infer(R, I_1) \subseteq infer(R, I_2)$,
- 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* :- **not** *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



## 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, in the body of any rule with head, say, p

    p(..) :- …., q(..), …

    then the layer number of p is greater than or equal to the layer number of q
  - ★ Given any rule with a negative literal

    p(..) :- …, not q(..), …

    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.

# 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

| Region | Category | Sales | Subtotal |
|---|---|---|---|
| North | Computer Hardware | 1,000,000 | |
| | Computer Software | 500,000 | |
| | All categories | | 1,500,000 |
| South | Computer Hardware | 200,000 | |
| | Computer Software | 400,000 | |
| | All categories | | 600,000 |
| | | **Total Sales** | 2,100,000 |

# End of Chapter

# QBE Skeleton Tables for the Bank Example

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| | | | |

| *customer* | *customer-name* | *customer-street* | *customer-city* |
|---|---|---|---|
| | | | |

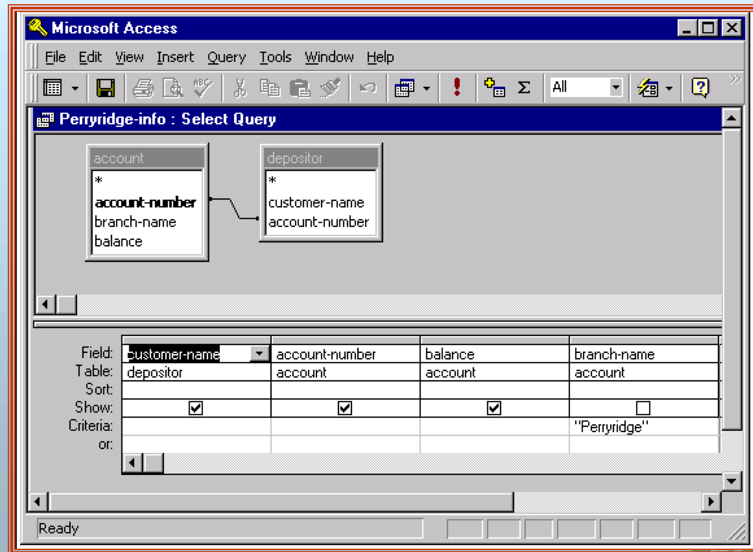
| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | | | |

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | | |

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | | | |

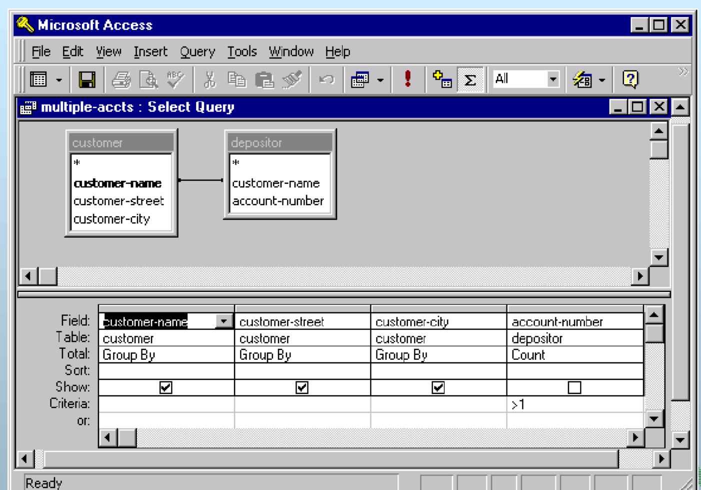
| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | | |

# An Example Query in Microsoft Access QBE






# An Aggregation Query in Microsoft Access QBE



## The *account* Relation

| *account-number* | *branch-name* | *balance* |
|---|---|---|
| A-101 | Downtown | 500 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-305 | Round Hill | 350 |
| A-201 | Perryridge | 900 |
| A-222 | Redwood | 700 |
| A-217 | Perryridge | 750 |




## The *v1* Relation

| *account-number* | *balance* |
|---|---|
| A-201 | 900 |
| A-217 | 750 |

# Result of *infer(R, I)*

| *account-number* | *balance* |
|---|---|
| A-201 | 900 |
| A-217 | 750 |



| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | P. | Perryridge | |

| *loan* | *loan-number* | *branch-name* | *amount* |
|---|---|---|---|
| | $\_x$ | Perryridge | |

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| I. | Capital | Queens | |

| *conditions* |
|---|
| $\_y \geq 2 * \_z$ |