Lesson 10: Relational Data Model & SQL
Contents

- Structure of Relational Databases
- Relational Algebra
- Basic Relational-Algebra Operations
- Additional Relational-Algebra Operations
- Extended Relational-Algebra Operations
- Null Values and Three-valued Logics
- Database Modification by Relational-Algebra Operations

- Brief Introduction to SQL
- SQL and Relations
- Fundamental SQL statements
- null values in SQL
- Database modifications in SQL
Why Relations?

- We have seen tables
- Why do we need another view of data?
- There is a number of reasons:
  - Need to create a rigorous mathematical model
  - This model enables for formalizing database operations
  - The exact model is needed to formalize declarative queries and optimize their processing
- The central idea is to describe a database as a collection of predicates over a finite set of predicate variables, defining constraints on the possible values and combinations of values.

<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-102</td>
<td>Bridges</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Brighton</td>
<td>900</td>
</tr>
<tr>
<td>A-215</td>
<td>Berkeley</td>
<td>700</td>
</tr>
<tr>
<td>A-217</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-305</td>
<td>Palo Alto</td>
<td>350</td>
</tr>
</tbody>
</table>
What is a Relation?

Mathematically, given sets $D_1, D_2, \ldots, D_n$ a relation $R$ is a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$

Thus, a relation is a set of $n$-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$

Example:

- $customer\_name = \{\text{Jones, Smith, Curry, Lindsay, \ldots}\}$ /* Set of all customer names */
- $customer\_street = \{\text{Main, North, Park, \ldots}\}$ /* Set of all street names*/
- $customer\_city = \{\text{Harrison, Rye, Pittsfield, \ldots}\}$ /* Set of all city names */

Then $r = \{(\text{Jones, Main, Harrison}), (\text{Smith, North, Rye}), (\text{Curry, North, Rye}), (\text{Lindsay, Park, Pittsfield})\}$

is a relation, i.e. subset of

$customer\_name \times customer\_street \times customer\_city$

As we are concerned with finite sets, such sets can be expressed by enumeration, i.e. tables
Relation is a Subset of a Cartesian Product

- No duplicates in sets
  - Very important for database applications
- Component set members can be in any order
  - Sorted or unsorted

Selected U.S. Presidents

First names

- Abraham
- Barac
- Bill
- Franklin
- George
- Jimmy
- John
- Theodore
- Thomas

Last names

- Bush
- Carter
- Clinton
- Jefferson
- Kenedy
- Lincoln
- Obama
- Roosevelt
- Washington
Attribute Types

- Each attribute of a relation has a name.
- The set of allowed values for each attribute is called the **domain** of the attribute.
- Attribute values are (normally) required to be **atomic**; that is, indivisible.
  - E.g. the value of an attribute can be an account number, but cannot be a set of account numbers.
- Domain is said to be atomic if all its members are atomic.
- The special value **null** is a member of every domain.
- The null value causes complications in the definition of many operations.
  - We shall ignore the effect of null values in our main presentation and consider their effect later.
Relation Schema & Relation Instance

- **Relation Schema**
  - $A_1, A_2, ..., A_n$ are attributes
  - $R = (A_1, A_2, ..., A_n)$ is a relation schema
    
    Example:
    
    $\text{Customer}_{-}\text{schema} = (\text{customer\_name}, \text{customer\_street}, \text{customer\_city})$
  
  - $r(R)$ denotes a relation $r$ on the relation schema $R$
    
    Example:
    
    $\text{customer} \ (\text{Customer\_schema})$

- **Relation Instance**
  - The current values (relation instance) of a relation are specified by a table
  - An element $t$ of $r$ is a tuple, represented by a row in a table

<table>
<thead>
<tr>
<th>customer_name</th>
<th>customer_street</th>
<th>customer_city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
</tbody>
</table>
A database consists of multiple relations

Information about an enterprise is broken up into parts, with each relation storing one part of the information

- `account` : stores information about accounts
- `depositor` : stores information about which customer owns which account
- `customer` : stores information about customers

Storing all information as a single relation such as `bank(account_number, balance, customer_name, ..)` results in
- repetition of information
  - e.g., if two customers own an account (What gets repeated?)
- the need for null values
  - e.g., to represent a customer without an account

Normalization theory deals with how to design relational schemas
Keys (revisited)

- Let $K \subseteq R$
- $K$ is a **superkey** of $R$ if values for $K$ are sufficient to identify a unique tuple of each possible relation $r(R)$
  - by “possible $r$ ” we mean a relation $r$ that could exist in the enterprise we are modeling.
  - Example: $\{\text{customer\_name, customer\_street}\}$ and $\{\text{customer\_name}\}$ are both superkeys of Customer, if no two customers can possibly have the same name
    - In real life, an attribute such as $\text{customer\_id}$ would be used instead of $\text{customer\_name}$ to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.

- $K$ is a **candidate key** if $K$ is minimal
  - Example: $\{\text{customer\_name}\}$ is a candidate key for Customer, since it is a superkey and no subset of it is a superkey.

- **Primary key**: a candidate key chosen as the principal means of identifying tuples within a relation
  - Should choose an attribute whose value never, or very rarely, changes.
    - E.g. email address is unique, but may change
Foreign Keys

- A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a **foreign key**.
  - E.g. `customer_name` and `account_number` attributes of `depositor` are foreign keys to `customer` and `account` respectively.
  - Only values occurring in the primary key attribute of the referenced relation may occur in the foreign key attribute of the referencing relation.

![Diagram showing relationships between `branch`, `account`, `depositor`, `customer`, `loan`, and `borrower` relations.](image-url)
Relational Algebra

- Procedural language
- Six basic operators
  - select: \( \sigma \)
  - project: \( \Pi \)
  - union: \( \cup \)
  - set difference: \(-\)
  - Cartesian product: \( \times \)
  - rename: \( \rho \)

The operators take one or two relations as inputs and produce a new relation as a result.
Select Operation

- Notation: $\sigma_p(r)$
- $p$ is called the selection predicate
- Defined as:
  $$\sigma_p(r) = \{ t \mid t \in r \text{ and } p(t) \}$$

Where $p$ is a formula in propositional calculus consisting of terms connected by: $\land$ (and), $\lor$ (or), $\lnot$ (not)
Each term is one of:
  - $<\text{attribute}> \ op \ <\text{attribute}>$ or $<\text{constant}>
  - where $op$ is one of: $=, \neq, >, \geq, <, \leq$

- Example of selection: $\sigma_{\text{branch\_name="Redwood"}}(\text{account})$

\[
\begin{array}{cccc}
A & B & C & D \\
\alpha & \alpha & 1 & 7 \\
\alpha & \beta & 5 & 7 \\
\beta & \beta & 12 & 3 \\
\beta & \beta & 23 & 10 \\
\end{array}
\]

\[
\begin{array}{cccc}
A & B & C & D \\
\alpha & \alpha & 1 & 7 \\
\beta & \beta & 23 & 10 \\
\end{array}
\]
Project Operation

Notation: $\Pi_{A_1, A_2, \ldots, A_k}(r)$

where $A_1, A_2$ are attribute names and $r$ is a relation name.

The result is defined as the relation of $k$ columns obtained by erasing the columns that are not listed.

- Duplicate rows are removed from result, since relations are sets.

Example: To eliminate the `branch_name` attribute of `account`

$$\Pi_{\text{account\_number, balance}}(\text{account})$$

\[
\begin{array}{ccc}
A & B & C \\
\alpha & 10 & 1 \\
\alpha & 20 & 1 \\
\beta & 30 & 1 \\
\beta & 40 & 2 \\
\end{array}
\]

\[
\begin{array}{cc}
A & C \\
\alpha & 1 \\
The \Pi_{A,C}(r) \\
\beta & 1 \\
\beta & 2 \\
\end{array}
\]

\[
\begin{array}{cc}
A & C \\
\alpha & 1 \\
\beta & 1 \\
\beta & 2 \\
\end{array}
\]
Union Operation

- Notation: \( r \cup s \)
- Defined as:
  \[ r \cup s = \{ t \mid t \in r \text{ or } t \in s \} \]
- For \( r \cup s \) to be valid.
  1. \( r, s \) must have the same **arity** (same number of attributes)
  2. The attribute domains must be **compatible**
     (example: 2\textsuperscript{nd} column of \( r \) deals with the same type of values as does the 2\textsuperscript{nd} column of \( s \))
- Example: to find all customers with either an account or a loan
  \[ \Pi_{\text{customer\_name}} (\text{depositor}) \cup \Pi_{\text{customer\_name}} (\text{borrower}) \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
</tr>
<tr>
<td>α</td>
<td>2</td>
</tr>
<tr>
<td>β</td>
<td>1</td>
</tr>
</tbody>
</table>

\( r \)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>α</td>
<td>2</td>
</tr>
<tr>
<td>β</td>
<td>3</td>
</tr>
</tbody>
</table>

\( s \)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
</tr>
<tr>
<td>β</td>
<td>1</td>
</tr>
<tr>
<td>β</td>
<td>3</td>
</tr>
</tbody>
</table>

\( r \cup s \)
Set Difference Operation

- Notation \( r – s \)
- Defined as:
  \[
  r – s = \{t \mid t \in r \text{ and } t \notin s\}
  \]

- Set differences must be taken between compatible relations.
  - \( r \) and \( s \) must have the same arity
  - attribute domains of \( r \) and \( s \) must be compatible

Relations \( r, s \):

\[
\begin{array}{c|c}
A & B \\
\hline
\alpha & 1 \\
\alpha & 2 \\
\beta & 1 \\
\end{array}
\quad
\begin{array}{c|c}
A & B \\
\hline
\alpha & 2 \\
\beta & 3 \\
\end{array}
\quad
\begin{array}{c|c}
A & B \\
\hline
\alpha & 1 \\
\beta & 1 \\
\end{array}
\]

\( r – s \):
Cartesian-Product Operation

- Notation \( r \times s \)
- Defined as:
  \[
  r \times s = \{ t, q \mid t \in r \text{ and } q \in s \}
  \]

- Assume that attributes of \( r(R) \) and \( s(S) \) are disjoint
  - That is, \( R \cap S = \emptyset \).
- Can build expressions using multiple operations
- If attributes of \( r(R) \) and \( s(S) \) are not disjoint, then renaming must be used.

\[
\begin{array}{cc}
A & B \\
\alpha & 1 \\
\beta & 2 \\
\end{array} \quad \begin{array}{ccc}
C & D & E \\
\alpha & 10 & a \\
\beta & 10 & a \\
\beta & 20 & b \\
\gamma & 10 & b \\
\end{array}
\]

\[
\begin{array}{cc}
A & B \\
\alpha & 1 \\
\alpha & 1 \\
\beta & 2 \\
\beta & 2 \\
\beta & 2 \\
\end{array} \quad \begin{array}{ccc}
C & D & E \\
\alpha & 10 & a \\
\alpha & 10 & b \\
\beta & 20 & b \\
\beta & 10 & a \\
\beta & 10 & b \\
\end{array}
\]

Relations \( r, s \):

- \( r \times s \):

**Caution**: May generate HUGE tables
### Composition of Operations

- Building operations by composing several together

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td>α</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>β</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>β</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>γ</td>
<td>10</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>α</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>β</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>β</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>γ</td>
<td>10</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

\[ r \times s: \]

\[ A=B \quad C=D \quad E=E \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td>α</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>β</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>β</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
Rename Operation

- Not a “true relational algebra” operation
- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

\[ \rho_X(E) \]

returns the expression \( E \) under the name \( X \)

- If a relational-algebra expression \( E \) has arity \( n \), then

\[ \rho_X(A_1, A_2, \ldots, A_n)(E) \]

returns the result of expression \( E \) under the name \( X \), and with the attributes renamed to \( A_1, A_2, \ldots, A_n \).
Banking Example

Relations
- \textit{branch}(\textit{branch\_name}, \textit{branch\_city}, \textit{assets})
- \textit{customer}(\textit{customer\_name}, \textit{customer\_street}, \textit{customer\_city})
- \textit{account}(\textit{account\_number}, \textit{branch\_name}, \textit{balance})
- \textit{loan}(\textit{loan\_number}, \textit{branch\_name}, \textit{amount})
- \textit{depositor}(\textit{customer\_name}, \textit{account\_number})
- \textit{borrower}(\textit{customer\_name}, \textit{loan\_number})

Example Queries
- Find all loans of over $1200
  \[ \sigma_{\textit{amount}>1200}(\textit{loan}) \]
- Find the loan number for each loan amounting over $1200
  \[ \Pi_{\textit{loan\_number}}(\sigma_{\textit{amount}>1200}(\textit{loan})) \]
- Find the names of all customers who have an account at the Redwood branch
  \[ \Pi_{\textit{customer\_name}}(\sigma_{\textit{branch\_name}=\text{Redwood}}(\sigma_{\textit{depositor}\cdot\textit{account\_number}=\textit{account}\cdot\textit{account\_number}}(\textit{depositor}\times\textit{loan}))) \]
Banking Example (cont.)

Example Queries (cont.)

- Find the names of all customers who have a loan at the Redwood branch but do not have an account at any branch of the bank

\[ \Pi_{\text{customer\_name}} \left( \sigma_{\text{branch\_name} = \text{Redwood}} \left( \sigma_{\text{borrower\_loan\_number} = \text{loan\_loan\_number}} \left( \text{borrower} \times \text{loan} \right) \right) \right) \]

\[ - \Pi_{\text{customer\_name}} \left( \text{depositor} \right) \]

- Find the names of all customers who have a loan at the Redwood branch
  
  Possibility No. 1
  \[ \Pi_{\text{customer\_name}} \left( \sigma_{\text{branch\_name} = \text{Redwood}} \left( \sigma_{\text{borrower\_loan\_number} = \text{loan\_loan\_number}} \left( \text{borrower} \times \text{loan} \right) \right) \right) \]

  Possibility No. 2
  \[ \Pi_{\text{customer\_name}} \left( \sigma_{\text{borrower\_loan\_number} = \text{loan\_loan\_number}} \left( \sigma_{\text{branch\_name} = \text{Redwood}} \left( \text{borrower} \times \text{loan} \right) \right) \right) \times \text{loan} \]
Banking Example (cont.)

**Example Queries (use of *rename*)**

- Find the largest account balance
- **Strategy:**
  - Find those balances that are *not* the largest
  - Rename *account* relation as *temp* so that we can compare each account balance with all others
  - Use set difference to find those account balances that were *not* found in the earlier step.
- The query is:

\[
\Pi_{\text{balance}}(\text{account}) - \Pi_{\text{account.balance}}
\left(\sigma_{\text{account.balance} < \text{temp.balance}}(\text{account} \times \rho_{\text{temp}}(\text{account}))\right)
\]
Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation

- Let $E_1$ and $E_2$ be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_P(E_1)$, $P$ is a predicate on attributes in $E_1$
  - $\Pi_S(E_1)$, $S$ is a list consisting of some of the attributes in $E_1$
  - $\rho_x(E_1)$, $x$ is the new name for the result of $E_1$
Additional Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

- Set intersection
- Natural join
- Division
- Assignment
Set-Intersection Operation

- **Notation:** \( r \cap s \)
- **Defined as:**
  \[
  r \cap s = \{ t \mid t \in r \text{ and } t \in s \}
  \]
- **Assume:**
  - \( r, s \) have the same **arity**
  - attributes of \( r \) and \( s \) are compatible
- **Note:** \( r \cap s = r - (r - s) \)

Relations \( r, s \):  
\[
\begin{array}{c|c|c|c|c}
A & B & A & B \\
\alpha & 1 & \alpha & 2 \\
\alpha & 2 & \beta & 3 \\
\beta & 1 & \beta & \\
\end{array}
\]

\( r \cap s \):  
\[
\begin{array}{c|c|c|c|c}
A & B \\
\alpha & 2 \\
\end{array}
\]
Natural-Join Operation

- Notation: \( r \bowtie s \)
- Let \( r \) and \( s \) be relations on schemas \( R \) and \( S \) respectively. Then, \( r \bowtie s \) is a relation on schema \( R \cup S \) obtained as follows:
  - Consider each pair of tuples \( t_r \) from \( r \) and \( t_s \) from \( s \).
  - If \( t_r \) and \( t_s \) have the same value on each of the attributes in \( R \cap S \), add a tuple \( t \) to the result, where
    - \( t \) has the same value as \( t_r \) on \( r \)
    - \( t \) has the same value as \( t_s \) on \( s \)
- The result of the natural join is the set of all combinations of tuples in \( R \) and \( S \) that are equal on their common attribute names
- Example:
  - \( R = (A, B, C, D) \)
  - \( S = (E, B, D) \)
  - Result schema = \( (A, B, C, D, E) \)
  - \( r \bowtie s \) is defined as:
    \[
    \Pi (r \times s) (\sigma (r.B = s.B \land r.D = s.D))
    \]
Natural Join Operation – Example

Relations r, s:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
<td>μ</td>
<td>a</td>
</tr>
<tr>
<td>β</td>
<td>2</td>
<td>γ</td>
<td>a</td>
</tr>
<tr>
<td>γ</td>
<td>4</td>
<td>β</td>
<td>b</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>γ</td>
<td>a</td>
</tr>
<tr>
<td>δ</td>
<td>2</td>
<td>β</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>α</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>β</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>γ</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>δ</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>ε</td>
</tr>
</tbody>
</table>

Practical example

**Employee**

<table>
<thead>
<tr>
<th>Name</th>
<th>EmpId</th>
<th>DeptName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry</td>
<td>1235</td>
<td>Finance</td>
</tr>
<tr>
<td>Sally</td>
<td>2241</td>
<td>Sales</td>
</tr>
<tr>
<td>Joe</td>
<td>3401</td>
<td>Finance</td>
</tr>
<tr>
<td>Harriet</td>
<td>2202</td>
<td>Production</td>
</tr>
</tbody>
</table>

**Dept**

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finance</td>
<td>George</td>
</tr>
<tr>
<td>Sales</td>
<td>Harald</td>
</tr>
<tr>
<td>Production</td>
<td>Charles</td>
</tr>
</tbody>
</table>

**Employee ⊙ Dept**

<table>
<thead>
<tr>
<th>Name</th>
<th>EmpId</th>
<th>DeptName</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry</td>
<td>1235</td>
<td>Finance</td>
<td>George</td>
</tr>
<tr>
<td>Sally</td>
<td>2241</td>
<td>Sales</td>
<td>Harald</td>
</tr>
<tr>
<td>Joe</td>
<td>3401</td>
<td>Finance</td>
<td>George</td>
</tr>
<tr>
<td>Harriet</td>
<td>2202</td>
<td>Production</td>
<td>Charles</td>
</tr>
</tbody>
</table>
Division Operation

- Notation: \( r \div s \)
- Suited to queries that include the phrase “for all”.
- Let \( r \) and \( s \) be relations on schemas \( R \) and \( S \) respectively where
  - \( R = (A_1, ..., A_m, B_1, ..., B_n) \) and \( S = (B_1, ..., B_n) \)
  - The result of \( r \div s \) is a relation on schema \( R - S = (A_1, ..., A_m) \)
    \[
    r \div s = \{ t \mid t \in \Pi_{R-S}(r) \land \forall u \in s (tu \in r) \}
    \]
  - where \( tu \) means the concatenation of tuples \( t \) and \( u \) to produce a single tuple

- Property
  - Let \( q = r \div s \)
    - Then \( q \) is the largest relation satisfying \( q \times s \subseteq r \)

- Definition in terms of the basic algebra operation
  - Let \( r(R) \) and \( s(S) \) be relations, and let \( S \subseteq R \)
    \[
    r \div s = \Pi_{R-S}(r) - \Pi_{R-S}\left( (\Pi_{R-S}(r) \times s) - \Pi_{R,S,S}(r) \right)
    \]
  - To see why
    - \( \Pi_{R,S,S}(r) \) simply reorders attributes of \( r \)
    - \( \Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R,S,S}(r) \) gives those tuples \( t \) in \( \Pi_{R,S}(r) \) such that for some tuple \( u \in s \), \( tu \notin r \)
Division Operation – Example

- Relations $r$, $s$:

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$\beta$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$\delta$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$\delta$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

- Practical example

**Reports_to**

<table>
<thead>
<tr>
<th>Name</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry</td>
<td>George</td>
</tr>
<tr>
<td>Sally</td>
<td>Harald</td>
</tr>
<tr>
<td>Joe</td>
<td>George</td>
</tr>
<tr>
<td>Harriet</td>
<td>Charles</td>
</tr>
</tbody>
</table>

**Boss**

<table>
<thead>
<tr>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
</tr>
<tr>
<td>Charles</td>
</tr>
</tbody>
</table>

**Reports_to $\div$ Boss**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry</td>
</tr>
<tr>
<td>Joe</td>
</tr>
<tr>
<td>Harriet</td>
</tr>
</tbody>
</table>
Assignment Operation

The assignment operation (\(\leftarrow\)) provides a convenient way to express complex queries.

- Write query as a sequential program consisting of
  - a series of assignments
  - followed by an expression whose value is displayed as a result of the query.

- Assignment must always be made to a temporary relation variable.

Example: Write \(r \div s\) as

\[
\begin{align*}
temp1 & \leftarrow \Pi_{R \times S} (r) \\
temp2 & \leftarrow \Pi_{R \times S} ((temp1 \times s) - \Pi_{R \times S, S} (r)) \\
result & = temp1 - temp2
\end{align*}
\]

- The result to the right of the \(\leftarrow\) is assigned to the relation variable on the left of the \(\leftarrow\).
- May use variable in subsequent expressions.
Bank Example Queries

- Find the names of all customers who simultaneously have a loan and an account at bank
  \[ \Pi_{\text{customer name}} (\text{borrower}) \cap \Pi_{\text{customer name}} (\text{depositor}) \]

- Find the name of all customers who have a loan at the bank and the loan amount
  \[ \Pi_{\text{customer name, loan number, amount}} (\text{borrower} \bowtie \text{loan}) \]

- Find all customers who have an account from at least the “Downtown” and the "Uptown" branches
  - Possibility 1
    \[ \Pi_{\text{customer name}} (\sigma_{\text{branch name} = \text{“Downtown”}} (\text{depositor} \bowtie \text{account})) \cap \Pi_{\text{customer name}} (\sigma_{\text{branch name} = \text{“Uptown”}} (\text{depositor} \bowtie \text{account})) \]
  - Possibility 2
    \[ \Pi_{\text{customer name, branch name}} (\text{depositor} \bowtie \text{account}) \div \rho_{\text{temp(branch name)}} (\{(\text{“Downtown”}), (\text{“Uptown”})\}) \]
    
    \[ \cdot \text{Note, that this version uses a "constant relation"} \]
Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join
Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list
  \[ \Pi_{F_1, F_2, \ldots, F_n}(E) \]

- \( E \) is any relational-algebra expression

- Each of \( F_1, F_2, \ldots, F_n \) are arithmetic expressions involving constants and attributes in the schema of \( E \).

- Is used to compute ‘derived’ (calculated) attributes

- Given relation
  \[ \text{credit\_info(customer\_name, limit, credit\_balance)}, \]
  find how much more each person can spend:
  \[ \Pi_{\text{customer\_name, limit - credit\_balance}}(\text{credit\_info}) \]
Aggregate Functions and Operations

- **Aggregate function** takes a collection of values and returns a single value as a result.
  - **avg**: average value
  - **min**: minimum value
  - **max**: maximum value
  - **sum**: sum of values
  - **count**: number of values

- **Aggregate operation** in relational algebra

\[
G_1, G_2, \ldots, G_n \quad \theta \quad F_1(A_1), F_2(A_2), \ldots, F_n(A_n)(E)
\]

\(E\) is any relational-algebra expression
  - \(G_1, G_2, \ldots, G_n\) is a list of attributes on which to group (can be empty)
  - Each \(F_i\) is an aggregate function
  - Each \(A_i\) is an attribute name
Aggregate Operation – Example

Relation \( r \):

\[
\begin{array}{ccc}
A & B & C \\
\alpha & \alpha & 7 \\
\alpha & \beta & 7 \\
\beta & \beta & 3 \\
\beta & \beta & 10 \\
\end{array}
\]

\( \forall_{\text{sum}(C)}(r) \):

\[
\begin{array}{c}
\text{sum}(c) \\
27 \\
\end{array}
\]

Relation \( \text{account} \) grouped by \( \text{branch\_name} \):

<table>
<thead>
<tr>
<th>\text{branch_name}</th>
<th>\text{account_number}</th>
<th>\text{balance}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>A-102</td>
<td>400</td>
</tr>
<tr>
<td>Perryridge</td>
<td>A-201</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>750</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-215</td>
<td>750</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-222</td>
<td>700</td>
</tr>
</tbody>
</table>

\( \forall_{\text{branch\_name}} \text{sum(balance)}(\text{account}) \):

<table>
<thead>
<tr>
<th>\text{branch_name}</th>
<th>\text{sum(balance)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Brighton</td>
<td>1500</td>
</tr>
<tr>
<td>Redwood</td>
<td>700</td>
</tr>
</tbody>
</table>
Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.
    - We shall study precise meaning of comparisons with nulls later
## Outer Join – Example

<table>
<thead>
<tr>
<th>loan</th>
<th>borrower</th>
</tr>
</thead>
<tbody>
<tr>
<td>loan_number</td>
<td>branch_name</td>
</tr>
<tr>
<td>L-170</td>
<td>Downtown</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
</tr>
</tbody>
</table>

**Natural join**

<table>
<thead>
<tr>
<th>loan</th>
<th>borrower</th>
</tr>
</thead>
<tbody>
<tr>
<td>loan_number</td>
<td>branch_name</td>
</tr>
<tr>
<td>L-170</td>
<td>Downtown</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
</tr>
</tbody>
</table>

**Left outer join**

<table>
<thead>
<tr>
<th>loan</th>
<th>borrower</th>
</tr>
</thead>
<tbody>
<tr>
<td>loan_number</td>
<td>branch_name</td>
</tr>
<tr>
<td>L-170</td>
<td>Downtown</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
</tr>
</tbody>
</table>

**Right outer join**

<table>
<thead>
<tr>
<th>loan</th>
<th>borrower</th>
</tr>
</thead>
<tbody>
<tr>
<td>loan_number</td>
<td>branch_name</td>
</tr>
<tr>
<td>L-170</td>
<td>Downtown</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
</tr>
</tbody>
</table>

**Full outer join**

<table>
<thead>
<tr>
<th>loan</th>
<th>borrower</th>
</tr>
</thead>
<tbody>
<tr>
<td>loan_number</td>
<td>branch_name</td>
</tr>
<tr>
<td>L-170</td>
<td>Downtown</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
</tr>
</tbody>
</table>
Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same
Null Values (cont.)

- Comparisons with null values return the special logical value: \textit{unknown}
  - If \textit{false} was used instead of \textit{unknown}, then \textit{not (A < 5)}
    would not be equivalent to \textit{A >= 5}

- Three-valued logic using the truth value \textit{unknown}:
  - OR: \( (unknown \text{ or } true) \) = \textit{true},
    \( (unknown \text{ or } false) \) = \textit{unknown}
    \( (unknown \text{ or } unknown) \) = \textit{unknown}
  - AND: \( (true \text{ and } unknown) \) = \textit{unknown},
    \( (false \text{ and } unknown) \) = \textit{false},
    \( (unknown \text{ and } unknown) \) = \textit{unknown}
  - NOT: \( (not \ unknown) \) = \textit{unknown}

- Result of select predicate is treated as \textit{false} if it evaluates to \textit{unknown}
Modification of the Database

The content of the database may be modified using the following operations:

- Deletion
- Insertion
- Updating

All these operations are expressed using the assignment operator.
Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes.
- A deletion is expressed in relational algebra by:
  \[ r \leftarrow r - E \]
  where \( r \) is a relation and \( E \) is a relational algebra query.

Examples
- Delete all account records in the Perryridge branch
  \[ account \leftarrow account - \sigma_{\text{branch\_name} = \text{"Perryridge"}}(account) \]
- Delete all loan records with amount in the range of 0 to 50
  \[ loan \leftarrow loan - \sigma_{\text{amount} \geq 0 \text{ and } \text{amount} \leq 50}(loan) \]
**Insertion**

- To insert data into a relation, we either:
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- In relational algebra, an insertion is expressed by:
  \[
  r \leftarrow r \cup E
  \]
where \( r \) is a relation and \( E \) is a relational algebra expression.
- The insertion of a single tuple is expressed by letting \( E \) be a constant relation containing one tuple
- Example:
  - Insert information in the database specifying that Smith has $1200 in account A-973 at the Perryridge branch
    \[
    \text{account} \leftarrow \text{account} \cup \{ ("A-973", "Perryridge", 1200) \}
    \]
  - Provide as a gift for all loan customers in the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.
    \[
    \begin{align*}
    r_1 & \leftarrow (\sigma_{\text{branch name} = "Perryridge"}(\text{borrower} \bowtie \text{loan})) \\
    \text{account} & \leftarrow \text{account} \cup \Pi_{\text{loan number, branch name}}.200 (r_1) \\
    \text{depositor} & \leftarrow \text{depositor} \cup \Pi_{\text{customer name, loan number}}(r_1)
    \end{align*}
    \]
Updating

- A mechanism to change a value in a tuple without charging all values in the tuple
- Use the generalized projection operator to do this task
  \[ r \leftarrow \Pi_{F_1, F_2, \ldots, F_l}(r) \]

- Each \( F_i \) is either
  - the \( i \)th attribute of \( r \), if the \( i \)th attribute is not updated, or,
  - if the attribute is to be updated \( F_i \) is an expression, involving only constants and the attributes of \( r \), which gives the new value for the attribute

- Examples
  - Make interest payments by increasing all balances by 5 %
    \[ \text{account} \leftarrow \Pi_{\text{account}\_\text{number}, \text{branch}\_\text{name}, \text{balance} \ast 1.05}(\text{account}) \]
  - Pay all accounts with balances over $10,000 6 % interest and pay all others 5 %
    \[ \text{account} \leftarrow \Pi_{\text{account}\_\text{number}, \text{branch}\_\text{name}, \text{balance} \ast 1.06}(\sigma_{\text{BAL} > 10000}(\text{account})) \]
    \[ \cup \Pi_{\text{account}\_\text{number}, \text{branch}\_\text{name}, \text{balance} \ast 1.05}(\sigma_{\text{BAL} \leq 10000}(\text{account})) \]
Lesson 9, part 2
Structured Query Language (SQL)
Create Table Construct

- An SQL relation is defined using the `create table` command:
  
  ```sql
  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint_1), ..., (integrity-constraint_k))
  ```

  - `r` is the name of the relation
  - each `A_i` is an attribute name in the schema of relation `r`
  - `D_i` is the data type of values in the domain of attribute `A_i`

- Integrity constraints in `create table`
  - `not null`
  - `primary key(A_1, ..., A_l)`

- Example:
  
  ```sql
  create table branch
  (  branch_name  char(15) not null,
  branch_city   char(30),
  assets        integer,
  primary key(branch_name)
  )
  ```

- Note:
  - SQL names are case insensitive (i.e., you may use upper- or lower-case letters); e.g.: Branch_Name ≡ BRANCH_NAME ≡ branch_name
Basic Query Structure

SQL is based on set and relational operations with certain modifications and enhancements.

A typical SQL query has the form:

\[
\text{select } A_1, A_2, \ldots, A_n \\
\text{from } R_1, R_2, \ldots, R_m \\
\text{where } P
\]

- \(A_i\) represents an attribute
- \(R_i\) represents a relation
- \(P\) is a predicate.

This query is equivalent to the relational algebra expression

\[
\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (\bigotimes_{i=1}^{m} R_i))
\]

The result of an SQL query is a relation.

Important remark:
- SQL is a declarative (query) language while relational algebra is procedural.
- Mapping SQL queries to relational expressions converts declarative queries to procedures.
- Query execution will use procedures implementing relation algebra operations.
The select clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra

Example:
- Find the names of all branches in the *loan* relation:
  
  ```sql
  select branch_name from loan
  ```
- In the relational algebra, the query would be:
  
  \[ \Pi_{\text{branch\_name}}(\text{loan}) \]

SQL allows duplicates in relations as well as in query results
- This violates relational model assumptions but may speed-up processing

To force the elimination of duplicates, insert the keyword **distinct** after **select**.
- Find the names of all branches in the *loan* relations, and remove duplicates
  
  ```sql
  select distinct branch_name from loan
  ```
- The keyword **all** specifies that duplicates not be removed
  
  ```sql
  select all branch_name from loan
  ```
The select clause (cont.)

- An asterisk in the select clause denotes “all attributes”
  ```sql
  select * from loan
  ```

- The `select` clause can contain arithmetic expressions involving the operation, +, −, ∗, and /, and operating on constants or attributes of tuples

- The query
  ```sql
  select loan_number, branch_name, amount * 100
  from loan
  ```
  would return a relation that is the same as the `loan` relation, except that the value of the attribute `amount` is multiplied by 100
  - This is, in fact, the generalized projection
    ```sql
    \( \Pi_{loan_number, branch_name, amount * 100}(loan) \)
The \textit{where} clause

- The \textit{where} clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.

- Example
  - Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than $1200.
    \[
    \text{select loan\_number} \\
    \text{from loan} \\
    \text{where branch\_name = 'Perryridge' and amount > 1200}
    \]

- Comparison
  - Results can be combined using the logical connectives \textit{and}, \textit{or}, and \textit{not}.
  - Comparisons may be applied to results of arithmetic expressions.
  - SQL includes a \textit{between} comparison operator
    - Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, $\geq 90,000$ and $\leq 100,000$)
      \[
      \text{select loan\_number from loan} \\
      \text{where amount between 90000 and 100000}
      \]
      which maps to
      \[
      \Pi_{\text{loan\_number}}(\sigma_{(\text{amount} \geq 90000) \land (\text{amount} \leq 100000)}(\text{loan}))
      \]
The from clause

The from clause lists the relations involved in the query

- Corresponds to the Cartesian product operation of the relational algebra
- Find the Cartesian product borrower x loan
  
  ```sql
  select * from borrower, loan
  ```
- Find the name, loan number and loan amount of all customers having a loan at the Brighton branch
  
  ```sql
  select customer_name, borrower.loan_number, amount 
  from borrower, loan 
  where borrower.loan_number = loan.loan_number and branch_name = 'Brighton'
  ```

  corresponds to

  $$
  \Pi_{\text{customer_name, borrower.loan_number, amount}} (\sigma_{\text{borrower.loan_number = loan.loan_number} \land \text{branch_name = 'Brighton'}} (\text{borrower x loan}))
  $$
The Rename Operation

- The SQL allows renaming relations and attributes using the `as` clause:

  \[
  \text{old-name as new-name}
  \]

- Find the name, loan number and loan amount of all customers; rename the column name `loan_number` as `loan_id`

  ```sql
  select customer_name, borrower.loan_number as loan_id, amount
  from borrower, loan
  where borrower.loan_number = loan.loan_number
  ```

**Home work:**

- Rewrite this query to relational expression
Tuple Variables

- Tuple variables are defined in the from clause via the use of the as clause.

**Example**

- Find the customer names and their loan numbers for all customers having a loan at some branch:
  ```sql
  select customer_name, T.loan_number, S.amount
  from borrower as T, loan as S
  where T.loan_number = S.loan_number
  ```

- Find the names of all branches that have greater assets than some branch located in Brooklyn:
  ```sql
  select distinct T.branch_name
  from branch as T, branch as S
  where T.assets > S.assets and S.branch_city = 'Brooklyn'
  ```
SQL allows duplicates

- **Multiset** versions of some of the relational algebra operators – given multiset relations \( r_1 \) and \( r_2 \):
  - \( \sigma_{\theta}(r_1) \): If there are \( c_1 \) copies of tuple \( t_1 \) in \( r_1 \), and \( t_1 \) satisfies selections \( \sigma_{\theta} \), then there are \( c_1 \) copies of \( t_1 \) in \( \sigma_{\theta}(r_1) \).
  - \( \Pi_A(r) \): For each copy of tuple \( t_1 \) in \( r_1 \), there is a copy of tuple \( \Pi_A(t_1) \) in \( \Pi_A(r_1) \) where \( \Pi_A(t_1) \) denotes the projection of the single tuple \( t_1 \).
  - \( r_1 \times r_2 \): If there are \( c_1 \) copies of tuple \( t_1 \) in \( r_1 \) and \( c_2 \) copies of tuple \( t_2 \) in \( r_2 \), there are \( c_1 \times c_2 \) copies of the tuple \( t_1 \ldots t_2 \) in \( r_1 \times r_2 \).

- **Example**:
  - Suppose multiset relations \( r_1 (A, B) \) and \( r_2 (C) \) are as follows:
    \( r_1 = \{(1, a) \ (2,a)\} \quad r_2 = \{(2), (3), (3)\} \)
  - Then \( \Pi_B(r_1) \) would be \{\( (a), (a) \)\}, while \( \Pi_B(r_1) \times r_2 \) would be \{\( (a,2), (a,2), (a,3), (a,3), (a,3), (a,3) \)\}

- **SQL duplicate semantics**:
  - `select A_1, A_2, ..., A_n from r_1, r_2, ..., r_m where P` is equivalent to the multiset version of the expression:
    \[ \prod_{A_1, A_2, ..., A_n} (\sigma_P (r_1 \land r_2 \land \ldots \land r_m)) \]
Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $\neg$

- Find all customers who have a loan, an account, or both:
  
  $$(\text{select } customer\text{\_name from depositor}) \cup (\text{select } customer\text{\_name from borrower})$$

- Find all customers who have both a loan and an account:
  
  $$(\text{select } customer\text{\_name from depositor}) \cap (\text{select } customer\text{\_name from borrower})$$

- Find all customers who have an account but no loan
  
  $$(\text{select } customer\text{\_name from depositor}) \neg \text{ (select } customer\text{\_name from borrower})$$
Aggregate Functions in SQL

- These functions operate on the multiset of values of a column of a relation, and return a value.
  - `avg` average value
  - `min` minimum value
  - `max` maximum value
  - `sum` sum of values
  - `count` number of values

- Find the average account balance at the Perryridge branch
  ```sql
  select avg(balance) 
  from account 
  where branch_name = 'Perryridge'
  ```

- Find the number of depositors in the bank
  ```sql
  select count(distinct customer_name) 
  from depositor
  ```
Null Values

- It is possible for tuples to have a null value, denoted by `null`, for some of their attributes.
- `null` signifies an unknown value or that a value does not exist.
- The predicate `is null` can be used to check for null values.
  - Example: Find all loan number which appear in the `loan` relation with null values for `amount`.
    
    ```sql
    select loan_number from loan
    where amount is null
    ```

- The result of any arithmetic expression involving `null` is `null`.
  - Example: `5 + null` returns `null`.
- However, aggregate functions simply ignore nulls.
- Any comparison with `null` returns `unknown`.
  - Example: `5 < null` or `null <> null` or `null = null`.
- Three-valued logic using the truth value `unknown` is the same as above for relations.
- “`P is unknown`” evaluates to true if predicate `P` evaluates to `unknown`. 
Nested Subqueries

- SQL provides a mechanism for the nesting of queries
- A **subquery** is a `select-from-where` expression that is nested within another query
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality
- Example:
  - Find all customers who have both an account and a loan at the bank

```sql
select distinct customer_name
from borrower
where customer_name in (select customer_name
                          from depositor)
```
Views

■ In some cases, it is not desirable for all users to see the entire logical model
  ○ that is, all the actual relations stored in the database

■ Consider a person who needs to know a customer’s name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

  (select customer_name, borrower.loan_number, branch_name
   from borrower, loan
   where borrower.loan_number = loan.loan_number )

■ A view provides a mechanism to hide certain data from the view of certain users.
  ○ Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a view.

■ A view is defined using the create view statement which has the form

  create view v as <query expression>

  The view name is represented by v.
  ○ Once a view is defined, the view name can be used to refer to the virtual relation that the view generates
Modification of the Database

- **Deletion**
  - Statement is `delete-from-where` with the arguments similar to the `select-from-where` construct
  - Delete all account tuples at the Brighton branch
    
    ```
    delete from account where branch_name = 'Brighton'
    ```

- **Insertion**
  - Statement is: `insert into` relation `values`<compatible_relation>
    - Add a new tuple to `account`
      ```
      insert into account (branch_name, balance, account_number)
      values ('Perryridge', 1200, 'A-9732')
      ```

- **Updates**
  - Statement is: `update` relation `set` attribute = expression `where` condition
    - Add 6% to accounts over $1000
      ```
      update account set balance = balance*1.06 where balance>1000
      ```
Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the `from` clause.
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- Completely based on relational-algebra joins. SQL syntax described in the SQL standards.
- **Example**
  - Find all customers who have either an account or a loan (but not both) at the bank
    ```sql
    select customer_name
    from (depositor full outer join borrower )
    where account_number is null or loan_number is null
    ```
End of Lesson 10

Questions?