Part I: Introduction to Databases

Kostis Sagonas

---

**Introduction to Database Concepts**

- Purpose of Database Systems
- View of Data
- Data Models
- Data Definition Language
- Data Manipulation Language

---

**Database Management System (DBMS)**

- Collection of interrelated data
- Set of programs to access the data
- DBMS contains information about a particular enterprise
- DBMS provides an environment that is both convenient and efficient to use.

**Database Applications:**
- Banking: all transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions

- Databases touch all aspects of our lives

---

**Purpose of Database System**

- In the early days, database applications were built on top of file systems
- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
  - Need to write a new program to carry out each new task
  - Data isolation — multiple files and formats
  - Integrity problems
  - Integrity constraints (e.g., account balance > 0) become part of program code
  - Hard to add new constraints or change existing ones

---

**Purpose of Database Systems (Cont.)**

- Drawbacks of using file systems (cont.)
  - Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - E.g., transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
  - Concurrent accessed needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - E.g., two people reading a balance and updating it at the same time
  - Security problems

- Database systems offer solutions to all the above problems

---

**Levels of Abstraction**

- **Physical level** describes how a record (e.g., customer) is stored.
- **Logical level** describes data stored in database, and the relationships among the data.

```plaintext
type customer = record
  name : string;
  street : string;
  city : integer;
end;
```

- **View level**: application programs hide details of data types.
  Views can also hide information (e.g., salary) for security purposes.
**In troduction to Databases**

**Instances and Schemas**

- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
  - e.g., the database consists of information about a set of customers and accounts and the relationship between them
  - Analogous to type information of a variable in a program
- **Physical schema** – database design at the physical level
- **Logical schema** – database design at the logical level
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

**Data Models**

- A collection of tools for describing
  - **data**
  - **data relationships**
  - **data semantics**
  - **data constraints**
- **Entity-Relationship model**
- **Relational model**
- **Other models:**
  - object-oriented model
  - semi-structured data models
  - Older models: network model and hierarchical model

**Entity-Relationship Model**

- **Example of schema in the entity-relationship model**

**Relational Model**

- **Example of tabular data in the relational model**

**A Sample Relational Database**

- **customer_id**
  - customer-name
  - customer-street
  - customer-city
  - account-number
- **account-number**
  - balance

(a) The customer table

(b) The account table

(c) The depositor table
**Data Definition Language (DDL)**

- Specification notation for defining the database schema.
  - E.g.
    ```sql
    create table account (
        account-number char(10),
        balance integer
    )
    ```
- DDL compiler generates a set of tables stored in a data dictionary.
- Data dictionary contains metadata (i.e., data about data).
  - Database schema
  - Data storage and definition language
    - Language in which the storage structure and access methods used by the database system are specified.
    - Usually an extension of the data definition language.

**Data Manipulation Language (DML)**

- Language for accessing and manipulating the data organized by the appropriate data model.
  - DML also known as query language.
- Two classes of languages.
  - Procedural – user specifies what data is required and how to get those data.
  - Nonprocedural – user specifies what data is required without specifying how to get those data.
- SQL is the most widely used query language.

**SQL**

- SQL: widely used non-procedural language.
  - E.g. find the name of the customer with customer-id 192-83-7465
    ```sql
    select customer.customer-name
    from customer
    where customer.customer-id = '192-83-7465'
    ```
  - E.g. find the balances of all accounts held by the customer with customer-id 192-83-7465
    ```sql
    select account.balance
    from depositor, account
    where depositor.customer-id = '192-83-7465' and depositor.account-number = account.account-number
    ```
- Application programs generally access databases through:
  - Language extensions that allow embedded SQL.
  - Application program interfaces (e.g. ODBC/JDBC) which allow SQL queries to be sent to a database.

**Part II: The Relational Model**

- Structure of Relational Databases
- Relational Algebra
- Tuple Relational Calculus
- Domain Relational Calculus
- Extended Relational-Algebra-Operations
- Modification of the Database
- Views

**Example of a Relation**

<table>
<thead>
<tr>
<th>account-number</th>
<th>branch-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Brighton</td>
<td>900</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</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>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>
**Basic Structure**

- Formally, given sets $D_1, D_2, ..., D_n$, a relation $r$ is a subset of $D_1 \times D_2 \times ... \times D_n$.
- Thus, a relation is a set of $n$-tuples $(a_1, a_2, ..., a_n)$ where $a_i \in D_i$.
- Example: if
customer-name = (Jones, Smith, Curry, Lindsay)
customer-street = (Main, North, Park)
customer-city = (Harrison, Rye, Pittsfield)
- Then $r = \{(Jones, Main, Harrison), (Smith, North, Rye), (Curry, North, Rye), (Lindsay, Park, Pittsfield)\}$
is a relation over customer-name x customer-street x customer-city.

**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. multivalued attribute values are not atomic.
  - E.g. composite attribute values are not atomic.

**Relation Schema**

- $A_1, A_2, ..., A_n$ are attributes.
- $R = (A_1, A_2, ..., A_n)$ is a relation schema.
- $r(R)$ is a relation on the relation schema $R$.
- E.g. customer (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.

**Database**

- A database consists of multiple relations.
- Information about an enterprise is broken up into parts, with each relation storing one part of the information.
  - E.g.: account: stores information about accounts depositor: stores information about which account 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. two customers own an account).
  - The need for null values (e.g. represent a customer without an account).
- Normalization theory deals with how to design relational schemas.
The customer Relation

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>Spring</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Brooks</td>
<td>Senator</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Glenn</td>
<td>Sand Hill</td>
<td>Woodside</td>
</tr>
<tr>
<td>Green</td>
<td>Walnut</td>
<td>Stamford</td>
</tr>
<tr>
<td>Hayes</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Johnson</td>
<td>Alma</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Turner</td>
<td>Putnam</td>
<td>Stamford</td>
</tr>
<tr>
<td>Williams</td>
<td>Nassau</td>
<td>Princeton</td>
</tr>
</tbody>
</table>

The depositor Relation

<table>
<thead>
<tr>
<th>name</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>A-102</td>
</tr>
<tr>
<td>Johnson</td>
<td>A-101</td>
</tr>
<tr>
<td>Johnson</td>
<td>A-201</td>
</tr>
<tr>
<td>Jones</td>
<td>A-217</td>
</tr>
<tr>
<td>Lindsay</td>
<td>A-222</td>
</tr>
<tr>
<td>Smith</td>
<td>A-215</td>
</tr>
<tr>
<td>Turner</td>
<td>A-305</td>
</tr>
</tbody>
</table>

E-R Diagram for the Banking Enterprise

Keys

- 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: (customer-name, customer-street) and (customer-name) are both superkeys of Customer, if no two customers can possibly have the same name.
- $K$ is a candidate key if $K$ is minimal
- Example: (customer-name) is a candidate key for Customer, since it is a superkey (assuming no two customers can possibly have the same name), and no subset of it is a superkey.

Determining Keys from E-R Sets

- Strong entity set. The primary key of the entity set becomes the primary key of the relation.
- Weak entity set. The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set.
- Relationship set. The union of the primary keys of the related entity sets becomes a super key of the relation.

Schema Diagram for the Banking Enterprise
Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - procedural
  - non-procedural
- "Pure" languages:
  - Relational Algebra
  - Tuple Relational Calculus
  - Domain Relational Calculus
- Pure languages form underlying basis of query languages that people use.

Relational Algebra

- Procedural language
- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename
- The operators take two or more relations as inputs and give a new relation as a result.

Select Operation – Example

Relation \( r \):

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

\[ \sigma_{A \land B > 5} (r) \]

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

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), \( \neg \) (not)
Each term is one of:
  - \langle attribute \rangle op \langle constant \rangle
  - \langle attribute \rangle op \langle attribute \rangle
e.g. To eliminate the branch-name attribute of account

Project Operation – Example

Relation \( r \):

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

\[ \Pi_{A,C} (r) \]

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

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 removed from result, since relations are sets
- E.g. To eliminate the branch-name attribute of account
Introduction to Databases

Union Operation – Example

- Relations r, s:
  
<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>α</td>
<td>α</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **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 (e.g., 2nd column of \( r \) deals with the same type of values as does the 2nd column of \( s \))

- E.g., to find all customers with either an account or a loan

\[ \Pi_{\text{customer-name}}(\text{depositor}) \cup \Pi_{\text{customer-name}}(\text{borrower}) \]

Set Difference Operation – Example

- Relations r, s:
  
<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>α</td>
<td>α</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **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

Cartesian-Product Operation – Example

- Relations r, s:
  
<table>
<thead>
<tr>
<th>r</th>
<th>s</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>2</td>
</tr>
</tbody>
</table>

- **Cartesian-Product Operation**

**Notation:** \( r \times s \)

**Defined as:**

\[ r \times s = \{ t \mid t \in r \text{ and } q \in s \} \]

- Assume that attributes of \( r(R) \) and \( s(S) \) are disjoint. (That is, \( R \cap S = \emptyset \)).

- If attributes of \( r(R) \) and \( s(S) \) are not disjoint, then renaming must be used.

\[ \Pi_{\text{customer-name}}(\text{depositor}) \cup \Pi_{\text{customer-name}}(\text{borrower}) \]
Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{C=A}(r \times s)$

<table>
<thead>
<tr>
<th>r x s</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>b</td>
<td>19</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>a</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>a</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
<td>20</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
<td>10</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

$\sigma_{B=A}(r \times s)$

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

Rename Operation

- Allows us to name, and therefore 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 any $n$, then

$\rho_{X(A_1, A_2, ..., A_n)}(E)$

returns the result of expression $E$ under the name $X$, and with the attributes renamed to $A_1, A_2, ..., A_n$.

Banking Example

- branch (branch-name, branch-city, assets)
- customer (customer-name, customer-street, customer-only)
- account (account-number, branch-name, balance)
- loan (loan-number, branch-name, amount)
- Depositor (customer-name, account-number)
- Borrower (customer-name, loan-number)

Example Queries

- Find all loans of over $1200$
  \[ \sigma_{\text{amount} > 1200} (\text{loan}) \]
- Find the loan number for each loan of an amount greater than $1200$
  \[ \prod_{\text{loan-number} \mid \sigma_{\text{amount} > 1200} (\text{loan})} \]

Example Queries

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

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  \[ \Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{Perryridge}} (\text{borrower.loan-number} = \text{loan.loan-number} (\text{borrower} \times \text{loan}))) \]
- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.
  \[ \Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{Perryridge}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number} (\text{borrower} \times \text{loan}))) \]
  \[ - \Pi_{\text{customer-name}} (\text{depositor}) \]
Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  - Query 1
    \[ \pi_{\text{customer-name}}(\sigma_{\text{branch-name} = \text{Perryridge}}(\text{borrower-loan-number} = \text{loan-loan-number}(\text{borrower} \times \text{loan}))) \]
  - Query 2
    \[ \pi_{\text{customer-name}}(\sigma_{\text{loan-loan-number}} = \text{borrower-loan-number}(\sigma_{\text{branch-name} = \text{Perryridge}}(\text{loan})) \times \text{borrower}) \]

Example Queries

- Find the largest account balance
  - Rename account relation as \( d \)
  - The query then is:
    \[ \pi_{\text{balance}}(\text{account}) \pi_{\text{account.balance}}(\sigma_{\text{account.balance} < d.\text{balance}}(\text{account} \times \rho_x(\text{account}))) \]

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:
  - \( E_1 \cup E_2 \)
  - \( E_1 \cap E_2 \)
  - \( E_1 \setminus E_2 \)
  - \( E_1 \times E_2 \)
  - \( \pi_{P}(E_1) \), \( P \) is a predicate on attributes in \( E_1 \)
  - \( \sigma_{P}(E_1) \), \( P \) is a predicate on 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:
  \[ r \cap s = \{ t | t \in r \text{ and } t \in s \} \]
- Assume:
  - \( r, s \) have the same arity
  - \( r \) and \( s \) are compatible
- Note: \( r \cap s = r \setminus (r \setminus s) \)

Set-Intersection Operation - Example

- Relation \( r, s \):
  - \( r \times s \)
    \[
    \begin{array}{|c|c|}
    \hline
    a & 1 \\
    a & 2 \\
    b & 1 \\
    \hline
    \end{array}
    \]
    \[
    \begin{array}{|c|c|}
    \hline
    a & 2 \\
    a & 2 \\
    b & 3 \\
    \hline
    \end{array}
    \]
  - \( r \cap s \)
    \[
    \begin{array}{|c|c|}
    \hline
    a & 2 \\
    \hline
    \end{array}
    \]
Natural Join Operation

- Notation: \( r \bowtie s \)
- Let \( r \) and \( s \) be relations on schemas \( R \) and \( S \) respectively. The result is a relation on schema \( R \cup S \) which is obtained by considering 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 \( R \cap S \), a tuple \( t \) is added to the result, where
  - \( t \) has the same value as \( t_r \) on \( r \)
  - \( t \) has the same value as \( t_s \) on \( s \)
- Example:
  \[ R = (A, B, C, D) \]
  \[ S = (E, B, D) \]
  \[ r \bowtie s \]
- Result schema = \( (A, B, C, D, E) \)
- \( r \bowtie s \) is defined as:
  \[ \Pi_{\text{A,B,C,D}}[r \times s] \]

Division Operation

- Suited to queries that include the phrase “for all”.
- Let \( r \) and \( s \) be relations on schemas \( R \) and \( S \) respectively where
  - \( R = (A_1, \ldots, A_m, B_1, \ldots, B_n) \)
  - \( S = (B_1, \ldots, B_n) \)
  - \( r \bowtie s \) is a relation on schema \( R - S = (A_1, \ldots, A_m) \)
  - \( r \bowtie s = \{ t \mid t \in \Pi_{R \setminus S}[r] \land \forall u \in s \{ t u r \} \} \)

Another Division Example

- Relations \( r, s \):
  \[
  \begin{array}{cccccc}
  A & B & C & D & E \\
  1 & 2 & 3 & 4 & 5 \\
  6 & 7 & 8 & 9 & 10
  \end{array}
  \]

  \[
  \begin{array}{cccc}
  D & E \\
  5 & 6 \\
  5 & 7
  \end{array}
  \]

Division Operation (Cont.)

- Property
  - Let \( q \bowtie r = s \)
  - Then \( q \bowtie r \) is the largest relation satisfying \( q \bowtie s \subseteq r \)
- Definition in terms of the basic algebra operation
  - Let \( R(t) \) and \( S(u) \) be relations, and let \( S \subseteq R \)
  - \( r \bowtie s = \Pi_{R \setminus S}( \Pi_{R \setminus S}(t) \bowtie s - \Pi_{R \setminus S}(t) ) \)

  To see why
  - \( \Pi_{R \setminus S}(t) \bowtie s \)
  - \( \Pi_{R \setminus S}(t) \bowtie s = \Pi_{R \setminus S}(t) \bowtie s \) gives those tuples \( t \) in \( \Pi_{R \setminus S}(t) \) such that for some tuple \( u \in s, tu \in r \).
Assignment Operation

- The assignment operation (←) provides a convenient way to express complex queries, writing a 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

  \[
  \text{temp1} \leftarrow \prod_{R-S}(r) \\
  \text{temp2} \leftarrow \prod_{R-S}((\text{temp1} \times s) - \prod_{R-S,S}(r)) \\
  \text{result} \leftarrow \text{temp1} \div \text{temp2}
  \]

  - The result to the right of the ← is assigned to the relation variable on the left of the ←.
  - May use variable in subsequent expressions.

Example Queries

- Find all customers who have an account from at least the "Downtown" and the Uptown" branches.
  - Query 1
    \[
    \prod_{\text{CName}}(\sigma_{\text{BName} = "Downtown"}(\text{Depositor}\rightarrow\text{Account})) \cap \\
    \prod_{\text{CName}}(\sigma_{\text{BName} = "Uptown"}(\text{Depositor}\rightarrow\text{Account}))
    \]
    where \( \text{CName} \) denotes customer-name and \( \text{BName} \) denotes branch-name.
  - Query 2
    \[
    \prod_{\text{CustomerName, BranchName}}(\text{Depositor}\rightarrow\text{Account}) \\
    - \text{Temp}(\text{BranchName} = ("Downtown", "Uptown"))
    \]

Example Queries

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

  \[
  \prod_{\text{CustomerName, BranchName}}(\text{Depositor}\rightarrow\text{Account}) \\
  \cap \prod_{\text{BranchName}}(\sigma_{\text{BranchCity} = "Brooklyn"}(\text{Branch}))
  \]

Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

  \[
  \prod_{F_1, F_2, \ldots, F_n}(E)
  \]
  
  - \( E \) 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 \( E \).
  - Given relation credit-info(customer-name, limit, credit-balance), find how much more each person can spend:

    \[
    \prod_{\text{CustomerName}, \text{limit} - \text{credit-balance}}(\text{credit-info})
    \]

Aggregate Functions and Operations

- Aggregation function takes a collection of values and returns a single value as a result.
  - \text{avg}: average value
  - \text{min}: minimum value
  - \text{max}: maximum value
  - \text{sum}: sum of values
  - \text{count}: number of values

  - Aggregate operation in relational algebra

    \[
    g_1, g_2, \ldots, g_n \prod_{\text{F_1, A_1}, \text{F_2, A_2}, \ldots, \text{F_n, A_n}}(E)
    \]

    - \( E \) 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 \( F_1 \) is an aggregate function
    - Each \( A_i \) is an attribute name
Introduction to Databases

Aggregate Operation – Example

Relation \( r \):

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

\( \sum_{C}(r) \)

\[
\begin{array}{ccc}
\text{sum-C} \\
27 \\
\end{array}
\]

Aggregate Operation – Example

Relation account grouped by branch-name:

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

branch-name \( \sum(\text{balance})(\text{account}) \)

<table>
<thead>
<tr>
<th>branch-name</th>
<th>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>

Aggregate Functions (Cont.)

- Result of aggregation does not have a name
- Can use rename operation to give it a name
- For convenience, we permit renaming as part of aggregate operation
  
  \[
  \text{branch-name} \sum(\text{balance}) \text{as sum-balance(} \text{account})
  \]

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 operation.

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.

Deletion Examples

- Delete all account records in the Perryridge branch.
  
  \[
  \text{account} \leftarrow \text{account} - \sigma_{\text{branch-name} = \text{‘Perryridge’}}(\text{account})
  \]

- Delete all loan records with amount in the range of 0 to 50
  
  \[
  \text{loan} \leftarrow \text{loan} - \sigma_{\text{amount} \geq 0 \text{ and amount} \leq 50}(\text{loan})
  \]

- Delete all accounts at branches located in Needham.
  
  \[
  r_{1} \leftarrow \sigma_{\text{branch-city} = \text{‘Needham’}}(\text{account})
  \]

  \[
  r_{2} \leftarrow \Pi_{\text{branch-name}, \text{account-number}, \text{balance}}(r_{1})
  \]

  \[
  r_{3} \leftarrow \Pi_{\text{customer-name}, \text{account-number}}(r_{2})
  \]

  \[
  \text{account} \leftarrow \text{account} - r_{3}
  \]

  \[
  \text{depositor} \leftarrow \text{depositor} - r_{3}
  \]
**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.

**Insertion Examples**

- Insert information in the database specifying that Smith has $1200 in account A-973 at the Perry ridge branch.
  
  \[
  \begin{align*}
  \text{account} & \leftarrow \text{account} \cup \{ ("Perry ridge", A-973, 1200) \} \\
  \text{depositor} & \leftarrow \text{depositor} \cup \{ ("Smith", A-973) \}
  \end{align*}
  \]

- Provide as a gift for all loan customers in the Perry ridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

  \[
  \begin{align*}
  r_1 & \leftarrow \sigma_{\text{branch-name} = "Perry ridge"} (\text{borrower}\times\text{loan}) \\
  \text{account} & \leftarrow \text{account} \cup \Pi_{\text{branch-name}, \text{account-number}, \text{balance}} (r_1) \\
  \text{depositor} & \leftarrow \text{depositor} \cup \Pi_{\text{customer-name}, \text{loan-number}} (r_1)
  \end{align*}
  \]

**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

  \[ r \leftarrow \Pi_{F_1, F_2, \ldots, F_I} (r) \]

  where 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.

**Update Examples**

- Make interest payments by increasing all balances by 5 percent.

  \[
  \begin{align*}
  \text{account} & \leftarrow \Pi_{\text{AN}, \text{BN}, \text{BAL}} * 1.05 (\text{account}) \\
  \text{where AN, BN and BAL stand for account-number, branch-name and balance, respectively.}
  \end{align*}
  \]

- Pay all accounts with balances over $10,000 6 percent interest and pay all others 5 percent.

  \[
  \begin{align*}
  \text{account} & \leftarrow \Pi_{\text{AN}, \text{BN}, \text{BAL}} \cdot 1.06 (\sigma_{\text{BAL} > 10000} (\text{account})) \\
  & \cup \Pi_{\text{AN}, \text{BN}, \text{BAL}} \cdot 1.05 (\sigma_{\text{BAL} \leq 10000} (\text{account}))
  \end{align*}
  \]

**Result of \( \sigma_{\text{branch-name} = "Perry ridge"} (\text{loan}) \)**

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
</tbody>
</table>

End of Part II
### Loan Number and the Amount of the Loan

<table>
<thead>
<tr>
<th>loan-number</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-11</td>
<td>900</td>
</tr>
<tr>
<td>L-14</td>
<td>1500</td>
</tr>
<tr>
<td>L-15</td>
<td>1500</td>
</tr>
<tr>
<td>L-16</td>
<td>1300</td>
</tr>
<tr>
<td>L-17</td>
<td>1000</td>
</tr>
<tr>
<td>L-23</td>
<td>2000</td>
</tr>
<tr>
<td>L-93</td>
<td>500</td>
</tr>
</tbody>
</table>

### Names of All Customers Who Have Either a Loan or an Account
- Adams
- Curry
- Hayes
- Jackson
- Jones
- Smith
- Williams
- Lindsay
- Johnson
- Turner

### Customers With An Account But No Loan
- Johnson
- Lindsay
- Turner

### Result of \( \sigma_{\text{branch-name} = "Perryridge"}(\text{borrower} \times \text{loan}) \)

<table>
<thead>
<tr>
<th>customer-name</th>
<th>loan-number</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-36</td>
<td>1500</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>1500</td>
</tr>
<tr>
<td>Curry</td>
<td>L-65</td>
<td>1500</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>1500</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-16</td>
<td>1500</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
<td>1500</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-16</td>
<td>1500</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
<td>1500</td>
</tr>
<tr>
<td>Jones</td>
<td>L-18</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-12</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-25</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-12</td>
<td>1500</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>1500</td>
</tr>
<tr>
<td>Williams</td>
<td>L-16</td>
<td>1300</td>
</tr>
</tbody>
</table>

### Result of \( \Pi_{\text{customer-name}} \)
- Adams
- Hayes
Largest Account Balance in the Bank

<table>
<thead>
<tr>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>900</td>
</tr>
</tbody>
</table>

Customers Who Live on the Same Street and In the Same City as Smith

<table>
<thead>
<tr>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curry</td>
</tr>
<tr>
<td>Smith</td>
</tr>
</tbody>
</table>

Customers With Both an Account and a Loan at the Bank

<table>
<thead>
<tr>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Smith</td>
</tr>
</tbody>
</table>

Result of $\Pi_{customer-name, loan-number, amount} (borrower \rightarrow loan)$

<table>
<thead>
<tr>
<th>customer-name</th>
<th>loan-number</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>1300</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>500</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>1500</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
<td>500</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
<td>1000</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>2000</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
<td>900</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>1000</td>
</tr>
</tbody>
</table>

Result of $\Pi_{branch-name} (\sigma_{customer-city = "Harrison"} (customer \rightarrow account \rightarrow depositor))$

<table>
<thead>
<tr>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
</tr>
<tr>
<td>Perryridge</td>
</tr>
</tbody>
</table>

Result of $\Pi_{branch-name} (\sigma_{branch-city = "Brooklyn"} (branch))$

<table>
<thead>
<tr>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
</tr>
<tr>
<td>Downtown</td>
</tr>
</tbody>
</table>
**Result of** $\Pi_{\text{customer-name, branch-name}} (\text{deposit}, \text{account})$

<table>
<thead>
<tr>
<th>customer-name</th>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>Perryridge</td>
</tr>
<tr>
<td>Johnson</td>
<td>Downtown</td>
</tr>
<tr>
<td>Johnson</td>
<td>Brighton</td>
</tr>
<tr>
<td>Jones</td>
<td>Brighton</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Redwood</td>
</tr>
<tr>
<td>Smith</td>
<td>Manus</td>
</tr>
<tr>
<td>Turner</td>
<td>Round Hill</td>
</tr>
</tbody>
</table>

**The credit-info Relation**

<table>
<thead>
<tr>
<th>customer-name</th>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>Perryridge</td>
</tr>
<tr>
<td>Johnson</td>
<td>Downtown</td>
</tr>
<tr>
<td>Johnson</td>
<td>Brighton</td>
</tr>
<tr>
<td>Jones</td>
<td>Brighton</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Redwood</td>
</tr>
<tr>
<td>Smith</td>
<td>Manus</td>
</tr>
<tr>
<td>Turner</td>
<td>Round Hill</td>
</tr>
</tbody>
</table>

**Result of** $\Pi_{\text{customer-name, (limit – credit-balance) as credit-available, (credit-info)}}$

<table>
<thead>
<tr>
<th>customer-name</th>
<th>credit-available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curry</td>
<td>250</td>
</tr>
<tr>
<td>Jones</td>
<td>5300</td>
</tr>
<tr>
<td>Smith</td>
<td>1600</td>
</tr>
<tr>
<td>Hayes</td>
<td>0</td>
</tr>
</tbody>
</table>

**The pt-workers Relation**

<table>
<thead>
<tr>
<th>employee-name</th>
<th>branch-name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Brown</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Gopal</td>
<td>Perryridge</td>
<td>5300</td>
</tr>
<tr>
<td>Johnson</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>Loreena</td>
<td>Downtown</td>
<td>1300</td>
</tr>
<tr>
<td>Peterson</td>
<td>Downtown</td>
<td>2500</td>
</tr>
<tr>
<td>Rao</td>
<td>Austin</td>
<td>1500</td>
</tr>
<tr>
<td>Sato</td>
<td>Austin</td>
<td>1600</td>
</tr>
</tbody>
</table>

**The pt-workers Relation After Grouping**

<table>
<thead>
<tr>
<th>employee-name</th>
<th>branch-name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rao</td>
<td>Austin</td>
<td>1500</td>
</tr>
<tr>
<td>Sato</td>
<td>Austin</td>
<td>1600</td>
</tr>
<tr>
<td>Johnson</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>Loreena</td>
<td>Downtown</td>
<td>1300</td>
</tr>
<tr>
<td>Peterson</td>
<td>Downtown</td>
<td>2500</td>
</tr>
<tr>
<td>Adams</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Brown</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Gopal</td>
<td>Perryridge</td>
<td>5300</td>
</tr>
</tbody>
</table>

**Result of** $\text{branch-name} \bowtie \text{sum(salary)} (\text{pt-workers})$

<table>
<thead>
<tr>
<th>branch-name</th>
<th>sum of salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Austin</td>
<td>3100</td>
</tr>
<tr>
<td>Downtown</td>
<td>5300</td>
</tr>
<tr>
<td>Perryridge</td>
<td>8100</td>
</tr>
</tbody>
</table>
In duction t o D a t a b ase s

Result of \( \text{branch-name} \notin \text{sum salary, max(salary)} \text{ as max-salary} \) (pt-works)

<table>
<thead>
<tr>
<th>branch-name</th>
<th>sum-salary</th>
<th>max-salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Austin</td>
<td>3100</td>
<td>1600</td>
</tr>
<tr>
<td>Downtown</td>
<td>5300</td>
<td>2500</td>
</tr>
<tr>
<td>Perryridge</td>
<td>8100</td>
<td>5300</td>
</tr>
</tbody>
</table>

Names of All Customers Who Have a Loan at the Perryridge Branch

<table>
<thead>
<tr>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
</tr>
<tr>
<td>Hayes</td>
</tr>
</tbody>
</table>

The branch Relation

<table>
<thead>
<tr>
<th>branch-name</th>
<th>branch-city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>Brooklyn</td>
<td>7100000</td>
</tr>
<tr>
<td>Downtown</td>
<td>Brooklyn</td>
<td>9000000</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>400000</td>
</tr>
<tr>
<td>North Town</td>
<td>Rye</td>
<td>3700000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>Horseneck</td>
<td>1700000</td>
</tr>
<tr>
<td>Pownal</td>
<td>Bennington</td>
<td>3000000</td>
</tr>
<tr>
<td>Redwood</td>
<td>Palo Alto</td>
<td>2100000</td>
</tr>
<tr>
<td>Round Hill</td>
<td>Horseneck</td>
<td>8000000</td>
</tr>
</tbody>
</table>

The loan Relation

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-11</td>
<td>Round Hill</td>
<td>900</td>
</tr>
<tr>
<td>L-14</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>L-17</td>
<td>Downtown</td>
<td>1000</td>
</tr>
<tr>
<td>L-23</td>
<td>Redwood</td>
<td>2000</td>
</tr>
<tr>
<td>L-93</td>
<td>Mianus</td>
<td>500</td>
</tr>
</tbody>
</table>

The borrower Relation

<table>
<thead>
<tr>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-16</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
</tr>
</tbody>
</table>

Part III: SQL
Introduction to Databases

SQL

- Basic Structure
- Set Operations
- Aggregate Functions
- Nested Subqueries
- Derived Relations
- Modification of the Database
- Data Definition Language
- Embedded SQL, ODBC and JDBC

Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:
  \[
  \text{select } A_1, A_2, \ldots, A_n \\
  \text{from } r_1, r_2, \ldots, r_m \\
  \text{where } P
  \]
  \(A_j\) represent attributes
  \(r_j\) represent relations
  \(P\) is a predicate.
- This query is equivalent to the relational algebra expression.
  \[\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))\]
  The result of an SQL query is a relation.

The select Clause

- The select clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- Find the names of all branches in the loan relation
  \[
  \text{select branch-name} \\
  \text{from loan}
  \]
  In the “pure” relational algebra syntax, the query would be:
  \[\Pi_{\text{branch-name}}(\text{loan})\]
- An asterisk in the select clause denotes “all attributes”
  \[
  \text{select } * \\
  \text{from loan}
  \]

NOTES:
- SQL does not permit the '*' character in names, so you would use, for example, branch_name instead of branch-name in a real implementation. We use '*' since it looks nice!
- SQL names are case insensitive.

The select Clause (Cont.)

- The select clause can contain arithmetic expressions involving the operation, +, -, *, and /, and operating on constants or attributes of tuples.
- The query:
  \[
  \text{select loan-number, branch-name, amount + 100} \\
  \text{from loan}
  \]
  would return a relation which is the same as the loan relations, except that the attribute amount is multiplied by 100.

The where Clause

- The where clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the from clause.
- The find all loan number for loans made at the Pennyridge branch with loan amounts greater than $1200,
  \[
  \text{select loan-number} \\
  \text{from loan} \\
  \text{where branch-name = Pennyridge and amount} > 1200
  \]
  Comparison results can be combined using the logical connectives and, or, and not.
  Comparisons can be applied to results of arithmetic expressions.
**The where Clause (Cont.)**

- SQL includes a `between` comparison operator in order to simplify `where` clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
- Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, $\geq 90,000$ and $\leq 100,000$)

```
select loan-number from loan
where amount between 90000 and 100000
```

**The from Clause**

- The `from` clause corresponds to the Cartesian product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- Find the Cartesian product borrower x loan

```
select *
from borrower, loan
```

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

```
select customer-name, borrower-loan-number, amount
from borrower, loan
where borrower-loan-number = loan-loan-number and branch-name = 'Perryridge'
```

**The Rename Operation**

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

```
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`.

```
select customer-name, borrower-loan-number as loan-id, amount
from borrower, loan
where borrower-loan-number = loan-loan-number
```

**Tuple Variables**

- Tuple variables are defined in the `from` clause via the use of the `as` clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

```
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.

```
select distinct T.branch-name
from branch as T, branch as S
where T.assets > S.amount and S.branch-city = 'Brooklyn'
```

**String Operations**

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
  * percent (%): The % character matches any substring.
  * underscore (_): The _ character matches any character.
- Find the names of all customers whose street includes the substring "Main".

```
select customer-name
from customer
where customer-street like '%Main%'
```

- Match the name "Main\%" like "Main\% escape \\

- SQL supports a variety of string operations such as:
  * concatenation using "||"
  * converting from upper to lower case (and vice versa)
  * finding string length, extracting substrings, etc.

**Ordering the Display of Tuples**

- List in alphabetic order the names of all customers having a loan in Perryridge branch

```
select distinct customer-name
from borrower, loan
where borrower-loan-number = loan-loan-number and branch-name = 'Perryridge'
order by customer-name
```

- We may specify `desc` for descending order or `asc` for ascending order, for each attribute; ascending order is the default.
  * E.g. `order by customer-name desc`
In relations with duplicates, SQL can define how many copies of tuples appear in the result.

Multiset versions of some of the relational algebra operators – given multiset relations \( r_1 \) and \( r_2 \):

1. If there are \( c_1 \) copies of tuple \( t_1 \) in \( r_1 \) and \( t_2 \) satisfies selection \( \sigma_{t_2} \) then there are \( c_1 \) copies of \( t_2 \) in \( \sigma_{t_2}(r_1) \).
2. For each copy of tuple \( t_1 \) in \( r_1 \), there is a copy of tuple \( \Pi_{t_1}(t_1) \) in \( \Pi_{t_1}(r_1) \) where \( \Pi_{t_1}(t_1) \) denotes the projection of the single tuple \( t_1 \).
3. If there are \( c_1 \times c_2 \) 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 \( t_1 \times t_2 \) in \( r_1 \times r_2 \).

Example: Suppose multiset relations \( r_1 \) (\( A, B \)) and \( r_2 \) (\( C \)) are as follows:

\( r_1 = ((1, a), (2, b)) \quad r_2 = ((2), (3), (3)) \)

Then \( \Pi_{A}(r_1) \) would be \(((a), (a))\), while \( \Pi_{A}(r_1) \times r_2 \) would be

\(((a,2), (a,2), (a,3), (a,3), (a,3))\)

SQL duplicate semantics:

\( \text{select } A_1, A_2, ..., A_n \text{ from } r_1, r_2, ..., r_m \text{ where } P \)

is equivalent to the multiset version of the expression:

\( \Pi_{A_1, A_2, ..., A_n}(\sigma_P(r_1 \times r_2 \times ... \times r_m)) \)

The set operations union, intersect, and except operate on relations and correspond to the relational algebra operations \( \cup, \cap, - \).

Each of the above operations automatically eliminates duplicates, to retain all duplicates use the corresponding multiset versions union all, intersect all and except all.

Suppose a tuple occurs \( m \) times in \( r \) and \( n \) times in \( s \), then, it occurs:

\( m+n \) times in \( r \cup s \) all \( s \)

\( \min(m,n) \) times in \( r \cap s \) all \( s \)

\( \max(0, m-n) \) times in \( r \setminus s \) all \( s \)

Set Operations

Find all customers who have a loan, an account, or both:

\( \text{select customer-name from depositor} \)

\( \text{union} \)

\( \text{select customer-name from borrower} \)

Find all customers who have both a loan and an account.

\( \text{select customer-name from depositor} \)

\( \text{intersect} \)

\( \text{select customer-name from borrower} \)

Find all customers who have an account but no loan.

\( \text{select customer-name from depositor} \)

\( \text{except} \)

\( \text{select customer-name from borrower} \)

Aggregate Functions

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

Aggregate Functions (Cont.)

Find the average account balance at the Perryridge branch.

\( \text{select avg(balance) from account where branch-name = 'Perryridge'} \)

Find the number of tuples in the customer relation.

\( \text{select count(*) from customer} \)

Find the number of depositors in the bank.

\( \text{select count(distinct customer-name) from depositor} \)
Aggregate Functions – Group By

- Find the number of depositors for each branch.
  
  ```sql
  select branch-name, count (distinct customer-name)
  from depositor, account
  where depositor.account-number = account.account-number
  group by branch-name
  ```
  
  Note: Attributes in `select` clause outside of aggregate functions must appear in `group by` list

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than $1.200.
  
  ```sql
  select branch-name, avg (balance)
  from account
  group by branch-name
  having avg (balance) > 1200
  ```
  
  Note: predicates in the `having` clause are applied after the formation of groups whereas predicates in the `where` clause are applied before forming groups

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- 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 Query

- Find all customers who have both an account and a loan at the bank.
  
  ```sql
  select distinct customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number and
  branch-name = 'Perryridge' and
  (branch-name, customer-name) in
  (select branch-name, customer-name
  from depositor, account
  where depositor.account-number =
  account.account-number)
  ```

- Find all customers who have a loan at the bank but do not have an account at the bank
  
  ```sql
  select distinct customer-name
  from borrower
  where customer-name not in (select customer-name
  from depositor)
  ```

- Same query using `some` clause
  
  ```sql
  select branch-name
  from branch
  where assets > some
  (select assets
  from branch
  where branch-city = 'Brooklyn')
  ```

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch
  
  ```sql
  select distinct customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number and
  branch-name = 'Perryridge' and
  (branch-name, customer-name) in
  (select branch-name, customer-name
  from depositor, account
  where depositor.account-number =
  account.account-number)
  ```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.
**Definition of Some Clause**

- \( F \text{ some } r \Rightarrow \exists t \in r \text{ s.t. } (F \text{ some } t) \)

  Where \( \text{comp} \) can be: \( <, <, >, =, ≠ \)

  - \( (5 < \text{some}) = \text{true} \) (read: 5 < some tuple in the relation)
  - \( (5 ≤ \text{some}) = \text{false} \)
  - \( (5 = \text{some}) = \text{true} \)
  - \( (5 ≠ \text{some}) = \text{true} \) (since \( 5 ≠ 5 \))

  However, \((≠ \text{some}) ≠ \text{not in}\)

---

**Definition of all Clause**

- \( F \text{ all } r \Rightarrow \forall t \in r \text{ s.t. } (F \text{ some } t) \)

  - \( (5 ≤ \text{all}) = \text{false} \)
  - \( (5 < \text{all}) = \text{true} \)
  - \( (5 = \text{all}) = \text{false} \)
  - \( (5 ≠ \text{all}) = \text{true} \) (since \( 5 ≠ 4 \) and \( 5 ≠ 6 \))

  However, \((≠ \text{all}) ≠ \text{not in}\)

---

**Example Query**

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

  ```
  select branch-name
  from branch
  where assets > all
  (select assets
  from branch
  where branch-city = 'Brooklyn')
  ```

---

**Test for Empty Relations**

- The \( \text{exists} \) construct returns the value \( \text{true} \) if the argument subquery is nonempty.

  ```
  \( \text{exists } r \Rightarrow r ≠ \emptyset \)
  ```

- \( \text{not exists } r \Rightarrow r = \emptyset \)

---

**Example Query**

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

  ```
  select distinct S.customer-name
  from depositor as S
  where not exists (
    (select T.branch-name
     from branch
     where branch-city = 'Brooklyn')
   except
    (select T.branch-name
     from depositor as T, account
     where T.account-number = R.account-number and
     S.customer-name = T.customer-name))
  ```

  **Note:**
  - \( X – Y = \emptyset \Rightarrow X ⊆ Y \)
  - **Note:** Cannot write this query using \( \text{all} \) and its variants

---

**Test for Absence of Duplicate Tuples**

- The \( \text{unique} \) construct tests whether a subquery has any duplicate tuples in its result.

  ```
  select R.customer-name
  from account, depositor as R
  where R.customer-name = R.customer-name and
  account.account-number and
  account.branch-name = 'Perryridge')
  ```

- Find all customers who have at most one account at the Perryridge branch.
Example Query

- Find all customers who have at least two accounts at the Perryridge branch.
  
  ```sql
  select distinct T.customer-name
  from depositor T
  where not unique (
    select R.customer-name
    from account, depositor as R
    where T.customer-name = R.customer-name and
    R.account-number = account.account-number and
    account.branch-name = 'Perryridge'
  )
  ```

Example Queries

- A view consisting of branches and their customers
  ```sql
  create view all-customer as
  (select branch-name, customer-name
  from depositor, account
  where depositor.account-number = account.account-number)
  union
  (select branch-name, customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number)
  ```

- Find all customers of the Perryridge branch
  ```sql
  select customer-name
  from all-customer
  where branch-name = 'Perryridge'
  ```

Derived Relations

- Find the average account balance of those branches where the average account balance is greater than $1200.
  ```sql
  select branch-name, avg(balance)
  from (select branch-name, avg(balance)
  from account
  group by branch-name)
  as result (branch-name, avg-balance)
  where avg-balance > 1200
  ```

  Note that we do not need to use the `having` clause, since we compute the temporary relation `result` in the `from` clause, and the attributes of `result` can be used directly in the `where` clause.

Modification of the Database – Deletion

- Delete all account records at the Perryridge branch
  ```sql
  delete from account
  where branch-name = 'Perryridge'
  ```

- Delete all accounts at every branch located in Needham city.
  ```sql
  delete from account
  where branch-name in (select branch-name
  from branch
  where branch-city = 'Needham')
  ```

  ```sql
  delete from depositor
  where account-number in (select account-number
  from branch, account
  where branch-city = 'Needham'
  and branch.branch-name = account.branch-name)
  ```

Example Query

- Delete the record of all accounts with balances below the average at the bank.
  ```sql
  delete from account
  where balance < (select avg(balance)
  from account)
  ```

  Problem: as we delete tuples from `deposit`, the average balance changes

  Solution used in SQL:

  1. First, compute `avg` balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing `avg` or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to account
  ```sql
  insert into account
  values ('A-9732', 'Perryridge',1200)
  ```

  or equivalently

  ```sql
  insert into account (branch-name, balance, account-number)
  values ('Perryridge', 1200, 'A-9732')
  ```

- Add a new tuple to account with balance set to null
  ```sql
  insert into account
  values ('A-777', 'Perryridge', null)
  ```
Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

- `insert into account`
  - `select loan-number, branch-name, 200`
  - `from loan`
  - `where branch-name = 'Perryridge'`

- `insert into depositor`
  - `select customer-name, loan-number`
  - `from loan, borrower`
  - `where branch-name = 'Perryridge'` and `loan.account-number = borrower.account-number`

The select from where statement is fully evaluated before any of its results are inserted into the relation (otherwise queries like `insert into (table1 select * from table1)` would cause problems.

Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

- Write two `update` statements:
  - `update account`
    - `set balance = balance * 1.06`
    - `where balance > 10000`
  - `update account`
    - `set balance = balance * 1.05`
    - `where balance <= 10000`

The order is important!

**Data Definition Language (DDL)**

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

**Domain Types in SQL**

- `char(n)`, Fixed length character string, with user-specified length n.
- `varchar(n)`, Variable length character strings, with user-specified maximum length n.
- `int`, Integer (a finite subset of the integers that is machine-dependent).
- `smallint`, Small integer (a machine-dependent subset of the integer domain type).
- `numeric(p,d)`, Fixed point number, with user-specified precision of p digits, with n digits to the right of decimal point.
- `real, double precision`, Floating point and double-precision floating point numbers, with machine-dependent precision.
- `float(n)`, Floating point number, with user-specified precision of at least n digits.

**Create Table Construct**

An SQL relation is defined using the `create table` command:

```sql
create table r (A1, D1, A2, D2, ..., An, Dn, (integrity-constraint1), ...
                (integrity-constraintn))
```

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

Example:

```sql
create table branch
    (branch-name char(15) not null,
     branch-city char(30),
     assets integer)
```

**Integrity Constraints in Create Table**

- `not null`
- `primary key {A1, ..., An}`
- `check (P)`, where `P` is a predicate

Example: Declare `branch-name` as the primary key for `branch` and ensure that the values of `assets` are non-negative.

```sql
create table branch
    (branch-name char(15),
     branch-city char(30),
     assets integer,
     primary key (branch-name),
     check (assets >= 0))
```

primary key declaration on an attribute automatically ensures `not null` in SQL-92 onwards, needs to be explicitly stated in SQL-89.
**Drop and Alter Table Constructs**

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation. All tuples in the relation are assigned null as the value for the new attribute. The form of the **alter table** command is:
  
  ```sql
  alter table r add A D
  ```

  where A is the name of the attribute to be added to relation r and D is the domain of A.
- The **alter table** command can also be used to drop attributes of a relation:
  
  ```sql
  alter table r drop A
  ```

  where A is the name of an attribute of relation r

  * Dropping of attributes not supported by many databases

---

**SQL Data Definition for Part of the Bank Database**

- **Create table**
  
  ```sql
  create table customer
  (customer-name varchar(25),
  customer-address varchar(50),
  customer-city varchar(25),
  customer-state varchar(2),
  customer-pin int,
  primary key (customer-name))
  ```

- **Create table**
  
  ```sql
  create table branch
  (branch-name varchar(25),
  branch-city varchar(25),
  assets int,
  primary key (branch-name),
  check (assets > 0))
  ```

- **Create table**
  
  ```sql
  create table account
  (account-number int,
  balance int,
  primary key (account-number),
  check (balance > 0))
  ```

- **Create table**
  
  ```sql
  create table deposit
  (depositor-name varchar(25),
  depositor-account-number int,
  primary key (depositor-name, account-number))
  ```

---

**Embedded SQL**

- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, and Cobol.
- A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- EXEC SQL statement is used to identify embedded SQL request to the preprocessor:

  ```sql
  EXEC SQL <embedded SQL statement> END-EXEC
  ```

  Note: this varies by language. E.g. the Java embedding uses:

  ```sql
  # SQL ( ... )
  ```

---

**Example Query**

From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

1. Specify the query in SQL and declare a cursor for it:

   ```sql
   EXEC SQL
   declare c cursor for
   select customer-name, customer-city
   from depositor, customer, account
   where depositor.customer-name = customer.customer-name
   and depositor-account-number = account.account-number
   and account.balance > :amount
   END-EXEC
   ```

---

**Embedded SQL (Cont.)**

- The **open** statement causes the query to be evaluated:

  ```sql
  EXEC SQL open c END-EXEC
  ```

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  ```sql
  EXEC SQL fetch c into :cn, :cc END-EXEC
  ```

  Repeated calls to **fetch** get successive tuples in the query result.

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available.

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  ```sql
  EXEC SQL close c END-EXEC
  ```

  Note: above details vary with language. E.g. the Java embedding defines Java iterators to step through result tuples.

---

**JDBC**

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- **JDBC** supports a variety of features for querying and updating data, and for retrieving query results.
- **JDBC** also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.

1. Model for communicating with the database:
   - Open a connection
   - Create a “statement” object
   - Execute queries using the Statement object to send queries and fetch results
   - Exception mechanism to handle errors
Introduction to Databases

JDBC Code

```java
public static void JDBCExample(String dbId, String userId, String password)
{
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@aurora.bell-lab.com:2000:bankdb", userId, password);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ... 
        stmt.close();
        conn.close();
    } catch (SQLException sqle) {
        System.out.println("SQLException: " + sqle);
    }
}
```

JDBC Code (Cont.)

```java
Update to database
try {
    stmt.executeUpdate("insert into account values ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple: " + sqle);
}
```

```java
Execute query and fetch and print results
ResultSet rs = stmt.executeQuery("select branch_name, avg(balance) from account
group by branch_name");
while (rs.next()) {
    System.out.println(rs.getString("branch_name") + " " + rs.getFloat(2));
}
```

JDBC Code Details

- **Getting result fields:**
  - rs.getString("branchname") and rs.getString(1) equivalent if branchname is the first argument of select result.
- **Dealing with Null values**
  - int a = rs.getInt("a");
  - if (rs.wasNull()) System.out.println("Got null value");