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.

  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.

Instances and Schemas

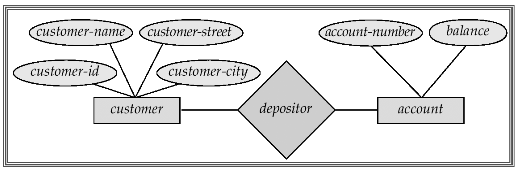
- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
  - ★ e.g., the database consists of information about a set of customers and accounts and the relationship between them)
  - ★ Analogous to type information of a variable in a program
  - ★ **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



Entity Relationship Model (Cont.)

- E-R model of real world
  - ★ Entities (objects)
    - ✓ E.g. customers, accounts, bank branch
  - ★ Relationships between entities
    - ✓ E.g. Account A-101 is held by customer Johnson
    - ✓ Relationship set *depositor* associates customers with accounts
- Widely used for database design
  - ★ Database design in E-R model usually converted to design in the relational model (coming up next) which is used for storage and processing

Relational Model

- Example of tabular data in the relational model

Attributes

customer-id	*customer-name*	*customer-street*	*customer-city*	*account-number*
192-83-7465	Johnson	Alma	Palo Alto	A-101
019-28-3746	Smith	North	Rye	A-215
192-83-7465	Johnson	Alma	Palo Alto	A-201
321-12-3123	Jones	Main	Harrison	A-217
019-28-3746	Smith	North	Rye	A-201

A Sample Relational Database

customer-id	*customer-name*	*customer-street*	*customer-city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account-number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer-id	*account-number*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Data Definition Language (DDL)

- Specification notation for defining the database schema
  - ★ E.g.
    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
    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
    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

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

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

Basic Structure

- Formally, given sets $D_1, D_2, \ldots D_n$ a *relation r* is a subset of $D_1 \times D_2 \times \ldots \times D_n$
  Thus a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where $a_i \in D_i$
- 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

- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*

  E.g. *Customer-schema* =
  (*customer-name, customer-street, customer-city*)
- *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

attributes

customer-name	*customer-street*	customer-city
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

tuples

customer

Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- E.g. *account* relation with unordered tuples

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

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 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. 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

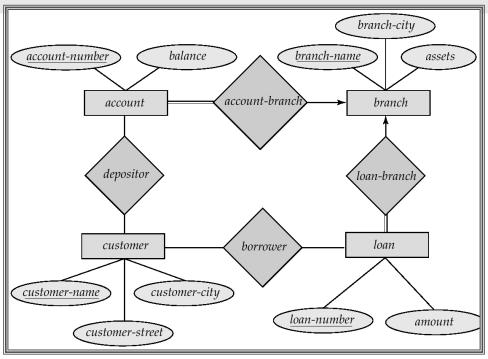
The *customer* Relation

customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

The *depositor* Relation

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

E-R Diagram for the Banking Enterprise



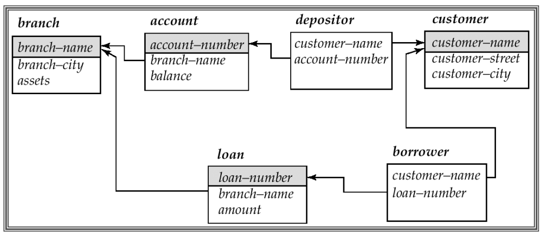
Keys

- Let K ⊆ 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.
  - ★ For binary many-to-one relationship sets, the primary key of the "many" entity set becomes the relation's primary key.
  - ★ For one-to-one relationship sets, the relation's primary key can be that of either entity set.
  - ★ For many-to-many relationship sets, the union of the primary keys becomes the relation's primary key

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*

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Select Operation

- Notation: $\sigma_p(r)$
- *p* is called the selection predicate
- Defined as:

  $\sigma_p(r) = \{t \mid t \in r \textbf{ and } p(t)\}$

  Where *p* is a formula in propositional calculus consisting of terms connected by : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
  Each term is one of:

  <attribute> *op* <attribute> or <constant>

  where *op* is one of: $=, \neq, >, \geq. <. \leq$
- Example of selection:
  $\sigma_{branch\text{-}name=\text{“Perryridge”}}(account)$

Project Operation – Example

- Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\Pi_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

- Notation:

  $\Pi_{A1,\, A2,\, \ldots,\, Ak}(r)$

  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*
  $\Pi_{account\text{-}number,\ balance}(account)$

Union Operation – Example

- Relations *r, s:*

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

r ∪ s:

A	B
α	1
α	2
β	1
β	3

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
  $\Pi_{customer\text{-}name}$ (*depositor*) $\cup$ $\Pi_{customer\text{-}name}$ (*borrower)*

Set Difference Operation – Example

- Relations *r, s:*

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

r – s:

A	B
α	1
β	1

Set Difference Operation

- Notation *r – s*
- Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

- 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*:

r

A	B
α	1
β	2

s

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

r x *s*:

A	B	C	D	E
α	1	α	10	a
α	1	β	19	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation

- Notation *r* x *s*
- Defined as:

$$r \times s = \{t\, q \mid t \in r \textbf{ and } q \in s\}$$

- 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.

Composition of Operations

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

A	B	C	D	E
α	1	α	10	a
α	1	β	19	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

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

A	B	C	D	E
α	1	α	10	a
β	2	β	20	a
β	2	β	20	b

Rename 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

$$\rho_{X\,(A1,\,A2,\,\ldots,\,An)}(E)$$

returns the result of expression *E* under the name *X*, and with the attributes renamed to *A1, A2, …., An*.

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

  $\sigma_{amount > 1200}(loan)$

- Find the loan number for each loan of an amount greater than $1200

  $\Pi_{loan\text{-}number}(\sigma_{amount > 1200}(loan))$

Example Queries

- Find the names of all customers who have a loan, an account, or both, from the bank

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

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

  $\Pi_{customer\text{-}name}(borrower) \cap \Pi_{customer\text{-}name}(depositor)$

Example Queries

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

  $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name=\text{“Perryridge”}}$
  $(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan)))$

- 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.

  $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{“Perryridge”}}$
  $(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\text{borrower x loan})))$
  $- \; \Pi_{customer\text{-}name}(\text{depositor})$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  - ★ Query 1

    $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan)))$

  - ★ Query 2

    $\Pi_{customer\text{-}name}(\sigma_{loan.loan\text{-}number = borrower.loan\text{-}number}((\sigma_{branch\text{-}name = \text{"Perryridge"}}(loan)) \times borrower))$

Example Queries

Find the largest account balance

- ★ Rename *account* relation as *d*
- ★ The query then is:

  $\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_d(account)))$

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:
  - ★ $E_1 \cup E_2$
  - ★ $E_1 - E_2$
  - ★ $E_1 \times E_2$
  - ★ $\sigma_p(E_1)$, *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:
- $r \cap s = \{ t \mid t \in r$ **and** $t \in s \}$
- Assume:
  - ★ *r*, *s* have the *same arity*
  - ★ attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

Set-Intersection Operation - Example

- Relation r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- r ∩ s

A	B
α	2

Natural-Join Operation

- Notation: r ⋈ 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*)
- Result schema = (*A, B, C, D, E*)
- *r* ⋈ *s* is defined as:

$$\Pi_{r.A,\ r.B,\ r.C,\ r.D,\ s.E}\ (\sigma_{r.B = s.B\ \ r.D = s.D}\ (r\ \text{x}\ s))$$

Natural Join Operation – Example

- Relations r, s:

A	*B*	*C*	*D*
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	*D*	*E*
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	∈

s

r ⋈ *s*

A	*B*	*C*	*D*	*E*
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Division Operation

$$r \div s$$

- Suited to queries that include the phrase "for all".
- Let *r* and *s* be relations on schemas R and S respectively where
  - ★ $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - ★ $S = (B_1, \ldots, B_n)$

  The result of r ÷ s is a relation on schema

  $R - S = (A_1, \ldots, A_m)$

$$r \div s = \{\ t\ \mid\ t \in \Pi_{R\text{-}S}(r) \wedge \forall\ u \in s\ (\ tu \in r\ )\ \}$$

Division Operation – Example

Relations *r, s*:

A	*B*
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
∈	6
∈	1
β	2

r

B
1
2

s

r ÷ *s*:

A
α
β

Another Division Example

Relations *r, s*:

A	*B*	*C*	*D*	*E*
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	*E*
a	1
b	1

s

r ÷ *s*:

A	*B*	*C*
α	a	γ
γ	a	γ

Division Operation (Cont.)

- 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 $S \subseteq R$

$$r \div s = \Pi_{R\text{-}S}\ (r) - \Pi_{R\text{-}S}\ (\ (\Pi_{R\text{-}S}\ (r) \times s) - \Pi_{R\text{-}S,S}(r))$$

To see why

- ★ $\Pi_{R\text{-}S,S}(r)$ simply reorders attributes of *r*
- ★ $\Pi_{R\text{-}S}(\Pi_{R\text{-}S}\ (r) \times s) - \Pi_{R\text{-}S,S}(r))$ gives those tuples t in

  $\Pi_{R\text{-}S}\ (r)$ such that for some tuple $u \in s$, $tu \notin r$.

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

$$temp1 \leftarrow \Pi_{R-S}(r)$$
$$temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$$
$$result = temp1 - temp2$$

  - ★ 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.

Example Queries

- Find all customers who have an account from at least the "Downtown" and the Uptown" branches.
  - ★ Query 1

$$\Pi_{CN}(\sigma_{BN=\text{"Downtown"}}(depositor \bowtie account)) \cap$$
$$\Pi_{CN}(\sigma_{BN=\text{"Uptown"}}(depositor \bowtie account))$$

  where *CN* denotes customer-name and *BN* denotes *branch-name*.
  - ★ Query 2

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account)$$
$$\div\ \rho_{temp(branch\text{-}name)}(\{(\text{"Downtown"}), (\text{"Uptown"})\})$$

Example Queries

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

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account)$$
$$\div\ \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$$

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.

$$\Pi_{F1,\ F2,\ \ldots,\ Fn}(E)$$

- E is any relational-algebra expression
- Each of $F_1, F_2, \ldots, F_n$ are 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:

$$\Pi_{customer\text{-}name,\ limit\ -\ credit\text{-}balance}(credit\text{-}info)$$

Aggregate Functions and Operations

- **Aggregation 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

$$_{G1,\ G2,\ \ldots,\ Gn}\ \mathcal{G}_{\ F1(A1),\ F2(A2),\ldots,\ Fn(An)}(E)$$

  - ★ 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*:

A	*B*	*C*
α	α	7
α	β	7
β	β	3
β	β	10

$\mathcal{g}_{\textbf{sum(c)}}$ (r)

sum-C
27

Aggregate Operation – Example

- Relation *account* grouped by *branch-name*:

branch-name	*account-number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$_{branch\text{-}name}\,\mathcal{g}_{\,sum(balance)}\,(account)$

branch-name	*balance*
Perryridge	1300
Brighton	1500
Redwood	700

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

$$_{branch\text{-}name}\,\mathcal{g}_{\,\textbf{sum}(balance)\ \textbf{as}\ sum\text{-}balance}\,(account)$$

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 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.

Deletion Examples

- Delete all account records in the Perryridge branch.

  $account \leftarrow account - \sigma_{\,branch\text{-}name\, =\, \text{“Perryridge”}}\,(account)$

- Delete all loan records with amount in the range of 0 to 50

  $loan \leftarrow loan - \sigma_{\,amount\, \geq\, 0\ and\ amount\, \leq\, 50}\,(loan)$

- Delete all accounts at branches located in Needham.

  $r_1 \leftarrow \sigma_{\,branch\text{-}city\, =\, \text{“Needham”}}\,(account \bowtie branch)$
  $r_2 \leftarrow \Pi_{branch\text{-}name,\, account\text{-}number,\, balance}\,(r_1)$
  $r_3 \leftarrow \Pi_{\,customer\text{-}name,\, account\text{-}number}\,(r_2 \bowtie depositor)$
  $account \leftarrow account - r_2$
  $depositor \leftarrow depositor - r_3$

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 Perryridge branch.

  $account \leftarrow account \cup \{(\text{"Perryridge"}, \text{A-973}, 1200)\}$

  $depositor \leftarrow depositor \cup \{(\text{"Smith"}, \text{A-973})\}$

- 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.

  $r_1 \leftarrow (\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \bowtie loan))$

  $account \leftarrow account \cup \Pi_{branch\text{-}name,\ account\text{-}number, 200}(r_1)$

  $depositor \leftarrow depositor \cup \Pi_{customer\text{-}name,\ loan\text{-}number}(r_1)$

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

  $$r \leftarrow \Pi_{F1, F2, \ldots, Fl,}(r)$$
- Each F_i is either the ith attribute of r, if the ith 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.

  $account \leftarrow \Pi_{AN,\ BN,\ BAL\ *\ 1.05}(account)$

  where *AN*, *BN* and *BAL* stand for *account-number*, *branch-name* and *balance*, respectively.

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

$$account \leftarrow \Pi_{AN,\ BN,\ BAL\ *\ 1.06}(\sigma_{BAL > 10000}(account)) \cup \Pi_{AN,\ BN,\ BAL\ *\ 1.05}(\sigma_{BAL \leq 10000}(account))$$

End of Part II

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

loan-number	*branch-name*	*amount*
L-15	Perryridge	1500
L-16	Perryridge	1300

Loan Number and the Amount of the Loan

loan-number	*amount*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Names of All Customers Who Have Either a Loan or an Account

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Customers With An Account But No Loan

customer-name
Johnson
Lindsay
Turner

Result of *borrower* × *loan*

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Result of $\sigma_{\textit{branch-name} = \text{“Perryridge”}}$ (*borrower* × *loan*)

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Result of $\Pi_{\textit{customer-name}}$

customer-name
Adams
Hayes

Largest Account Balance in the Bank

balance
900

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

customer-name
Curry
Smith

Customers With Both an Account and a Loan at the Bank

customer-name
Hayes
Jones
Smith

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

customer-name	*loan-number*	*amount*
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Result of $\Pi_{branch\text{-}name}(\sigma_{customer\text{-}city = \text{“Harrison”}}$*(customer ⋈ account ⋈ depositor))*

branch-name
Brighton
Perryridge

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

branch-name
Brighton
Downtown

Result of $\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account)$

customer-name	*branch-name*
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

The *credit-info* Relation

customer-name	*branch-name*
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

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

customer-name	*credit-available*
Curry	250
Jones	5300
Smith	1600
Hayes	0

The *pt-works* Relation

employee-name	*branch-name*	*salary*
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600

The *pt-works* Relation After Grouping

employee-name	*branch-name*	*salary*
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300

Result of $_{branch\text{-}name}\ \mathcal{G}\ _{\text{sum}(salary)}\ (pt\text{-}works)$

branch-name	*sum of salary*
Austin	3100
Downtown	5300
Perryridge	8100

Result of $_{branch\text{-}name}\,\mathcal{G}$ sum *salary*, max(*salary*) as *max-salary* (*pt-works*)

branch-name	*sum-salary*	*max-salary*
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300

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

customer-name
Adams
Hayes

The *branch* Relation

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

The *loan* Relation

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

The *borrower* Relation

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Part III: SQL

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:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - ★ A_is represent attributes
  - ★ r_is represent relations
  - ★ P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\Pi_{A1, A2, ..., An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$$

- 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

  select *branch-name*
  from *loan*
- In the "pure" relational algebra syntax, the query would be:

$$\Pi_{\text{branch-name}}(loan)$$

- An asterisk in the select clause denotes "all attributes"

  select *
  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 nicer!
- ★ SQL names are case insensitive.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- 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

  select distinct *branch-name*
  from *loan*
- The keyword **all** specifies that duplicates not be removed.

  select all *branch-name*
  from *loan*

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:

  select *loan-number, branch-name, amount* ∗ 100
  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. If consists of a predicate involving attributes of the relations that appear in the **from** clause.
- The find all loan number for loans made a the Perryridge branch with loan amounts greater than $1200.

  select *loan-number*
  from *loan*
  where *branch-name* = 'Perryridge' **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, ≥\$90,000 and ≤\$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.assets* **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**

Duplicates

- 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_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
  2. 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.
  3. 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 $t_1. t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

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

  $r_1 = \{(1, a)\ (2,a)\}$ $r_2 = \{(2), (3), (3)\}$
- 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:

  $\Pi_{A1,, A2, ..., An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$

Set Operations

- 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* **union all** *s*
  - ★ min(*m,n)* times in *r* **intersect all** *s*
  - ★ max(0, *m – n)* times in *r* **except all** *s*

Set Operations

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

  (select *customer-name* **from** *depositor*)
  union
  (select *customer-name* **from** *borrower)*
- Find all customers who have both a loan and an account.

  (**select** *customer-name* **from** *depositor*)
  intersect
  (select *customer-name* **from** *borrower)*
- Find all customers who have an account but no loan.

  (**select** *customer-name* **from** *depositor*)
  except
  (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.

  select avg *(balance)*
  from *account*
  where *branch-name* = 'Perryridge'
- Find the number of tuples in the *customer* relation.

  select count (*)
  from *customer*
- Find the number of depositors in the bank.

  select count (distinct *customer-name)*
  from *depositor*

Aggregate Functions – Group By

- Find the number of depositors for each branch.

  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.

  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.

  select distinct *customer-name*
  from *borrower*
  where *customer-name* **in (select** *customer-name*
  from depositor)

- Find all customers who have a loan at the bank but do not have an account at the bank

  select distinct *customer-name*
  from *borrower*
  where *customer-name* **not in (select** *customer-name*
  from *depositor)*

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch

  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.

Set Comparison

- Find 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.assets* **and**
  S.branch-city = 'Brooklyn'

- Same query using > **some** clause

  select *branch-name*
  from *branch*
  where *assets* > **some**
  (select *assets*
  from *branch*
  where *branch-city* = 'Brooklyn')

Definition of Some Clause

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ *s.t.* (F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5< **some** [0, 5, 6]) = true
(read: 5 < some tuple in the relation)

(5< **some** [0, 5]) = false

(5 = **some** [0, 5]) = true

(5 ≠ **some** [0, 5]) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Definition of all Clause

- F <comp> **all** $r \Leftrightarrow \forall\, t \in r$ (F <comp> t)

(5< **all** [0, 5, 6]) = false

(5< **all** [6, 10]) = true

(5 = **all** [4, 5]) = false

(5 ≠ **all** [4, 6]) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≢ **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 **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \varnothing$
- **not exists** $r \Leftrightarrow r = \varnothing$

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

- Note that $X - Y = \varnothing \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

select *T.customer-name*
from *depositor* **as** *T*
where 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 Query

- Find all customers who have at least two accounts at the Perryridge branch.

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

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

  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.

  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

  delete from *account*
  where *branch-name* = 'Perryridge'

- Delete all accounts at every branch located in Needham city.

delete from *account*
where *branch-name* **in** (**select** *branch-name*
    from *branch*
    where *branch-city* = 'Needham')
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.

  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*

  insert into *account*
    values ('A-9732', 'Perryridge',1200)

  or equivalently

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

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
    values ('A-777','Perryridge', *null*)

Modification of the Database – Insertion

- 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 *table*1 **select** * **from** *table*1
  would cause problems

Modification of the Database – Updates

- 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:

  create table *r* $(A_1\ D_1, A_2\ D_2, \ldots, A_n\ D_n,$
  (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
- Example:

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


Integrity Constraints in Create Table

- **not null**
- **primary key** $(A_1, \ldots, A_n)$
- **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.

```
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 **after 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

  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

  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 customer
   (customer-name    char(20),
    customer-street  char(30),
    customer-city    char(30),
    primary key (customer-name))

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

create table account
   (account-number   char(10),
    branch-name      char(15),
    balance          integer,
    primary key (account-number),
    check (balance >= 0))

create table depositor
   (customer-name    char(20),
    account-number   char(10),
    primary key (customer-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

  EXEC SQL <embedded SQL statement > END-EXEC

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

- Specify the query in SQL and declare a *cursor* for it

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

  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.

  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.

  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
- 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

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
          "jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
            … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```


JDBC Code (Cont.)

- 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);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery( "select branch_name, avg(balance)
                                     from account
                                     group by branch_name");
while (rset.next()) {
     System.out.println(
            rset.getString("branch_name") + "  " + rset.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()) Systems.out.println(“Got null value”);
```