Chapter 3: SQL

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
  - SQL:2006
  - SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

Data Definition Language

- The set of relations in a database must be specified to the system by means of a 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 d digits to the right of decimal point.
  - numeric(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly.
- **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.
- More are covered in Chapter 4.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

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

- 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, ..., A_n)

Example: Declare *branch_name* as the primary key for *branch.*

> **create table** *branch*
> (*branch_name* char(15)**,**
> *branch_city* char(30),
> *assets* integer,
> **primary key** (*branch_name*))

primary key declaration on an attribute automatically ensures **not null**

Drop and Alter Table Constructs

- **drop table** command: deletes all information about the dropped relation from the database.

  drop table r

- **alter table** command: add attributes to an existing relation:

  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.

  - All tuples in the relation are assigned *null* as the value for the new attribute.

- 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

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database

Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

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

  - A_i represents an attribute
  - r_i represents a relation
  - P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:

  select *branch_name*
  from *loan*

- In the relational algebra, the query would be:

$$\Pi_{branch\_name}(loan)$$

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.

SQL Example 1

CUST

customer-id	*customer-name*	*customer-street*	*customer-city*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

- SELECT *customer-id*
  FROM CUST

- The result on the right will be shown on screen.

customer-id
019-28-3746
182-73-6091
192-83-7465
244-66-8800
321-12-3123
335-57-7991
336-66-9999
677-89-9011
963-96-3963

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*

SQL example 2

customer-id	*customer-name*	*customer-street*	*customer-city*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

- SELECT *customer-city*
  FROM CUST

customer-city
Rye
Stamford
Palo Alto
Rye
Harrison
Pittsfield
Pittsfield
Harrison
Princeton

- SELECT DISTINCT *customer-city*
  FROM CUST

customer-city
Rye
Stamford
Palo Alto
~~Rye~~
Harrison
Pittsfield
~~Pittsfield~~
~~Harrison~~
Princeton

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *loan*

- 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 that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100.

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at 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
- Example: 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 lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower* $\times$ *loan*

  select $*$
  from *borrower, loan*

SQL example 3

CUST

cust-id	*name*
1	John
2	Smith
3	Joan

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- Write an SQL query to display, for each account, its id and the name of its owner.

- Obviously, we cannot answer this query using only one table.
- We need to do filtering and projection on the cartesian product.

- Answer:
- SELECT ACC.*acc-id*, CUST.*name*
  FROM CUST, ACC
  WHERE CUST.*cust-id* = ACC.*cust-id*

- Let us understand the query step-by-step.

SQL example 3 (cont.)

CUST

cust-id	*name*
1	John
2	Smith
3	Joan

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- SELECT ACC.*acc-id*, CUST.*name*
  FROM CUST, ACC
  WHERE
  CUST.*cust-id* = ACC.*cust-id*

- First, compute the cartesian product.

CUST. *cust-id*	CUST. *name*	ACC. *acc-id*	ACC. *cust-id*	ACC. *balance*
1	John	A1	1	20k
1	John	A2	1	5k
1	John	A3	2	35k
1	John	A4	3	100k
2	Smith	A1	1	20k
2	Smith	A2	1	5k
2	Smith	A3	2	35k
2	Smith	A4	3	100k
3	Joan	A1	1	20k
3	Joan	A2	1	5k
3	Joan	A3	2	35k
3	Joan	A4	3	100k

SQL example 3 (cont.)

CUST. *cust-id*	CUST. *name*	ACC. *acc-id*	ACC. *cust-id*	ACC. *balance*
1	John	A1	1	20k
1	John	A2	1	5k
1	John	A3	2	35k
1	John	A4	3	100k
2	Smith	A1	1	20k
2	Smith	A2	1	5k
2	Smith	A3	2	35k
2	Smith	A4	3	100k
3	Joan	A1	1	20k
3	Joan	A2	1	5k
3	Joan	A3	2	35k
3	Joan	A4	3	100k

- SELECT ACC.*acc-id*, CUST.*name*
  FROM CUST, ACC
  WHERE
  CUST.*cust-id* = ACC.*cust-id*

- Then, on the cartesian product, perform filtering.

CUST. *cust-id*	CUST. *name*	ACC. *acc-id*	ACC. *cust-id*	ACC. *balance*
1	John	A1	1	20k
1	John	A2	1	5k
2	Smith	A3	2	35k
3	Joan	A4	3	100k

SQL example 3 (cont.)

CUST. *cust-id*	CUST. *name*	ACC. *acc-id*	ACC. *cust-id*	ACC. *balance*
1	John	A1	1	20k
1	John	A2	1	5k
2	Smith	A3	2	35k
3	Joan	A4	3	100k

- SELECT ACC.*acc-id*, CUST.*name*
  FROM CUST, ACC
  WHERE CUST.*cust-id* = ACC.*cust-id*

ACC. *acc-id*	CUST. *name*
A1	John
A2	John
A3	Smith
A4	Joan

- Finally, apply projection.
- In general, if a query involves two (or more) relations, we call it a join.

SQL example 3 (cont.)

CUST. *cust-id*	CUST. *name*	ACC. *acc-id*	ACC. *cust-id*	ACC. *balance*
1	John	A1	1	20k
1	John	A2	1	5k
2	Smith	A3	2	35k
3	Joan	A4	3	100k

ACC. *acc-id*	CUST. *name*
A1	John
A2	John
A3	Smith
A4	Joan

- SELECT ACC.*acc-id*, CUST.*name*
  FROM CUST, ACC
  WHERE CUST.*cust-id* = ACC.*cust-id*

- The above query can be simplified as:
- SELECT *acc-id*, *name*
  FROM CUST, ACC
  WHERE CUST.*cust-id* = ACC.*cust-id*

- No ambiguity can arise because CUST doesn't have 'acc-id' and ACC doesn't have '*name*'.

SQL example 3 (cont.)

CUST. *cust-id*	CUST. *name*	ACC. *acc-id*	ACC. *cust-id*	ACC. *balance*
1	John	A1	1	20k
1	John	A2	1	5k
2	Smith	A3	2	35k
3	Joan	A4	3	100k

- SELECT *acc-id*, *name*
  FROM CUST, ACC
  WHERE CUST.*cust-id* = ACC.*cust-id*

ACC. *acc-id*	CUST. *name*
A1	John
A2	John
A3	Smith
A4	Joan

- The above query can be further written as:
  SELECT *acc-id*, *name*
  FROM CUST T1, ACC T2
  WHERE T1.*cust-id* = T2.*cust-id*

- T1 and T2 are used to rename the input tables.

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*

SQL example 4

customer-id	*customer-name*	*customer-street*	*customer-city*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

cid
019-28-3746
182-73-6091
192-83-7465
244-66-8800
321-12-3123
335-57-7991
336-66-9999
677-89-9011
963-96-3963

- SELECT *customer-id* AS cid
  FROM CUST

- Use **AS** in SELECT clause to rename output columns

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'

- Keyword **as** is optional and may be omitted

  borrower **as** *T* ≡ *borrower T*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that 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 all strings beginning with "ab%cd"

  like 'ab\%cd%' **escape** '\'
- SQL supports a variety of string operations such as
  - concatenation (using "ll")
  - 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.
  - Example: **order by** *customer_name* **desc**

SQL example 4

- The previous queries do not have any ordering requirements.
- We can request ordered results using 'ORDER BY'.
- SELECT *
  FROM ACC
  WHERE *balance* > 10000
  ORDER BY *balance*
- SELECT *
  FROM ACC
  WHERE *balance* > 10000
  ORDER BY *balance* DESC

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

acc-id	*cust-id*	*balance*
A1	1	20k
A3	2	35k
A4	3	100k

acc-id	*cust-id*	*balance*
A4	3	100k
A3	2	35k
A1	1	20k

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset:** set with repetitions
- **Multiset** versions of some of the relational algebra operators
  - Given multiset relations r_1 and r_2:

1. $\boldsymbol{\sigma_\theta(r_1)}$**:** 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. $\boldsymbol{\Pi_A(r_1)}$**:** 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. $\boldsymbol{r_1 \times r_2}$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1.\ t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

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

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

- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, while $\Pi_B(r_1) \times r_2$ would be

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

- SQL duplicate semantics:

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

is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database

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

SQL example 5 (intersection)

CUST

cust-id	*name*
1	John
2	Smith
3	Joan
4	Mike

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- (SELECT *cust-id*
  FROM CUST)
  INTERSECT
  (SELECT *cust-id*
  FROM ACC)

cust-id
1
2
3

- INTERSECT automatically removes duplicates.

SQL example 5 (cont.)

CUST

cust-id	*name*
1	John
2	Smith
3	Joan
4	Mike

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- (SELECT *cust-id*, *name*
  FROM CUST)
  INTERSECT
  (SELECT *cust-id*
  FROM ACC)

- The above query is wrong!
- Because the two SELECT clauses return different sets of columns. Hence, intersection cannot be performed.

SQL example 6 (union)

CUST

cust-id	*name*
1	John
2	Smith
3	Joan
4	Mike

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- (SELECT *cust-id*
  FROM CUST)
  UNION
  (SELECT *cust-id*
  FROM ACC)

cust-id
1
2
3
4

- UNION automatically removes duplicates.
- As with INTERSECT, the two SELECT queries must return the same columns, too.

SQL example 7 (except: set difference)

CUST

cust-id	*name*
1	John
2	Smith
3	Joan
4	Mike

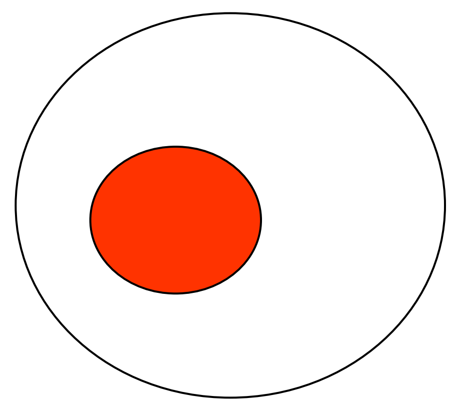
ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- (SELECT *cust-id*
  FROM CUST)
  EXCEPT
  (SELECT *cust-id*
  FROM ACC)

cust-id
4

- Returns the id that is in CUST but not in ACC.
- EXCEPT removes duplicates.
- The two SELECT must return the same columns.



SQL example 7 (except: set difference)

CUST

cust-id	*name*
1	John
2	Smith
3	Joan
4	Mike

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	35k
A4	3	100k

- Another example:
- (SELECT *cust-id*
  FROM ACC)
  EXCEPT
  (SELECT *cust-id*
  FROM CUST)
- Returns the id that is in ACC but not in CUST.
- There is no such id. So, the query result is empty.

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

Aggregate Functions

- These functions operate on a collection (a set or multiset) of values of a column of a relation, and return a single value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- Note: An aggregate function *cannot* be used directly in where clause

Aggregate Function Example

- Find the total amount of money ever deposited into account A1.

- SELECT SUM(*amount*)
  FROM DEPOSIT
  WHERE *acc-id* = 'A1'

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

- Find the total number of times that account A1 has been deposited into.

- SELECT COUNT(*)
  FROM DEPOSIT
  WHERE *acc-id* = 'A1'

Aggregate Function Example

- Find the number of distinct customers that ever deposited into account A1.

- SELECT COUNT(DISTINCT *cust-id*)
  FROM DEPOSIT
  WHERE *acc-id* = 'A1'

- Answer: 1

- Repeat the above query with respect to 'A3', the answer is 2.

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

Aggregate Functions – Group By

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

- SELECT *acc-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*

acc-id	
A1	55k
A2	5k
A3	135k

3 groups

- SELECT *acc-id*, *cust-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*, *cust-id*

acc-id	*cust-id*	
A1	1	55k
A2	1	5k
A3	2	35k
A3	3	100k

4 groups

Aggregate Functions – Group By

- In a group-by query, the SELECT clause can involve i) **attribute names** and ii) aggregate functions
  - **attribute name** must be some attribute appeared in the GROUP BY clause
  - aggregate function can take any attribute as argument
  - *have a single value per group*!

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

- For example
  SELECT *acc-id*, *cust-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*

  is wrong, due to the presence of *cust-id*.

Aggregate Functions – Having

- Find the total amount of money ever deposited into each account, provided that the account has been deposited at least twice.

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

- SELECT *acc-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*
  HAVING COUNT(*) >= 2

- HAVING applies only to groups, and hence, can be used with GROUP BY only.
  - The **having** clause is applied after GROUP BY
  - Predicates in the **where** clause are applied before GROUP BY
- HAVING usually contains only aggregate functions.

- Let us see how the above query is executed.

Aggregate Functions – Having

- SELECT *acc-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*
  HAVING COUNT(*) >= 2

- First, process GROUP BY.

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

DEPOSIT

	acc-id	*cust-id*	*amount*
GROUP 1	A1	1	20k
	A1	1	35k
GROUP 2	A2	1	5k
GROUP 3	A3	2	35k
	A3	3	100k

Aggregate Functions – Having

- SELECT *acc-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*
  HAVING COUNT(*) >= 2

- Then, process HAVING to eliminate the groups that do not qualify the HAVING condition.

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

DEPOSIT

	acc-id	*cust-id*	*amount*
GROUP 1	A1	1	20k
	A1	1	35k
GROUP 3	A3	2	35k
	A3	3	100k

Aggregate Functions – Having

- SELECT *acc-id*, SUM(*amount*)
  FROM DEPOSIT
  GROUP BY *acc-id*
  HAVING COUNT(*) >= 2

- Finally, process SELECT.

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

acc-id	
A1	55k
A3	135k

A common mistake

- Find the total amount of money ever deposited into each account, provided that the account has been deposited at least twice.

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	20k
A2	1	5k
A3	2	35k
A3	3	100k
A1	1	35k

- SELECT *acc-id*, SUM(*amount*)
  FROM DEPOSIT
  WHERE COUNT(*) >= 2
  GROUP BY *acc-id*

- The above query is wrong!
- There can be no aggregate function in WHERE.

- Remember: WHERE filters tuples, while an aggregate function applies to a group. So they are incompatible.

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

Null Values

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

    select *loan_number*
    from *loan*
    where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)
  from *loan*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

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

branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
account (account_number, branch_name, balance)
loan (loan_number, branch_name, amount)
depositor (customer_name, account_number)
borrower (customer_name, loan_number)

- 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*)

Set Comparison

branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
account (account_number, branch_name, balance)
loan (loan_number, branch_name, amount)
depositor (customer_name, account_number)
borrower (customer_name, loan_number)

- 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** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<, \ \leq, \ >, \geq =, \ \neq$

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = true

(read: 5 < some tuple in the relation)

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = false

(5 = **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true

(5 $\neq$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 $\neq$ 5)

(= **some**) $\equiv$ **in**
However, ($\neq$ **some**) $\not\equiv$ **not in**

Example Query

branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
account (account_number, branch_name, balance)
loan (loan_number, branch_name, amount)
depositor (customer_name, account_number)
borrower (customer_name, loan_number)

- 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')
```

Definition of all Clause

- F <comp> **all** *r* $\Leftrightarrow \forall\ t \in r$ (F <comp> *t*)
  - <comp> : $<,\ \leq,\ >,\ \geq,\ =,\ \neq$

$$(5 < \textbf{all} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \textbf{all} \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

($\neq$ **all**) $\equiv$ **not in**
However, (= **all**) $\not\equiv$ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the result of the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \varnothing$
- **not exists** $r \Leftrightarrow r = \varnothing$

Example (exists)

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	15k
A4	3	100k

- Find the ids of the accounts whose balances are not the largest.

- SELECT *acc-id*
  FROM ACC T1
  WHERE EXISTS (SELECT *
  FROM ACC T2
  WHERE T1.*balance* < T2.*balance*)

- Note that this nested query is different from the previous nested queries we have seen: It depends on the outside query.
  - T1 in the nested query references the table in the outside query.

- Lets see how it is executed.

Example (exists)

- SELECT *acc-id*
  FROM ACC T1
  WHERE EXISTS (SELECT *
  FROM ACC T2
  WHERE T1.*balance* < T2.*balance*)

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	15k
A4	3	100k

- Lets go over every tuple in T1 one by one. For each tuple, get its balance, and place it at the position of T1.*balance* to make the nested query complete.
- Specifically, when we are looking at the first tuple in T1, the nested query becomes:
  SELECT *
  FROM ACC T2
  WHERE 20k < T2.*balance*
- Execute it – does it return any tuples?
- Yes, so EXISTS evaluates to true, and the *acc-id* of the tuple in T1 we are looking at is displayed.

Example (exists)

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	15k
A4	3	100k

- SELECT *acc-id*
  FROM ACC T1
  WHERE EXISTS (SELECT *
  FROM ACC T2
  WHERE T1.*balance* < T2.*balance*)

- Repeat the above process for all tuples in T1.
- The *acc-ids* of all tuples are displayed, until we come to the last tuple, for which the nested query has the form:
  SELECT *
  FROM ACC T2
  WHERE 100k < T2.*balance*

- Execute it – does it return any tuples?
- No, so EXISTS evaluates to false, and the *acc-id* of the last tuple in T1 is not displayed.

Example (not exists)

- Find the id of the account whose balance is the largest.

ACC

acc-id	*cust-id*	*balance*
A1	1	20k
A2	1	5k
A3	2	15k
A4	3	100k

- SELECT *acc-id*
  FROM ACC T1
  WHERE NOT EXISTS (SELECT *
  FROM ACC T2
  WHERE T1.*balance* < T2.*balance*)

Example (not exists)

- Find the ids of the customers who have deposited into all accounts with balances larger than 15k.
- SELECT DISTINCT *cust-id*
  FROM DEPOSIT T1
  WHERE NOT EXISTS (
  (SELECT DISTINCT *acc-id*
  FROM ACC
  WHERE *balance* > 15000)
  EXCEPT
  (SELECT DISTINCT *acc-id*
  FROM DEPOSIT T2
  WHERE T1.*cust-id* = T2.*cust-id*))

ACC

acc-id	*balance*
A1	20k
A2	18k
A3	10k

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	2k
A1	1	1k
A2	1	1k
A2	2	3k
A3	3	2k
A3	2	5k

- Answer: 1
- What an inscrutable query! Lets understand it step-by-step.

Example (not exists)

- SELECT DISTINCT *cust-id* FROM DEPOSIT T1
  WHERE NOT EXISTS (
  (SELECT DISTINCT *acc-id*
  FROM ACC
  WHERE *balance* > 15000)
  EXCEPT
  (SELECT DISTINCT *acc-id*
  FROM DEPOSIT T2
  WHERE T1.*cust-id* = T2.*cust-id*))

ACC

acc-id	*balance*
A1	20k
A2	18k
A3	10k

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	2k
A1	1	1k
A2	1	1k
A2	2	3k
A3	3	2k
A3	2	5k

- The nested query depends on the outside query.
- So, we look at each tuple in T1, and use its *cust-id* to complete the nested query.
- Lets start with the first tuple in T1.

Example (not exists)

ACC

acc-id	*balance*
A1	20k
A2	18k
A3	10k

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	2k
A1	1	1k
A2	1	1k
A2	2	3k
A3	3	2k
A3	2	5k

- SELECT DISTINCT *cust-id* FROM DEPOSIT T1
  WHERE NOT EXISTS (
  (SELECT DISTINCT *acc-id*
  FROM ACC
  WHERE *balance* > 15000)
  EXCEPT
  (SELECT DISTINCT *acc-id*
  FROM DEPOSIT T2
  WHERE T1.*cust-id* = T2.*cust-id*))
- Lets start with the first tuple in T1. The nested query becomes:
- (SELECT DISTINCT *acc-id*
  FROM ACC
  WHERE *balance* > 15000)
  EXCEPT
  (SELECT DISTINCT *acc-id*
  FROM DEPOSIT T2
  WHERE 1 = T2.*cust-id*))

returns {A1, A2}

returns {A1, A2}

- The above query returns empty. So NOT EXIST evaluates to true, and *cust-id* 1 is displayed.

Example (not exists)

- SELECT DISTINCT *cust-id* FROM DEPOSIT T1
  WHERE NOT EXISTS (
  (SELECT DISTINCT *acc-id*
  FROM ACC
  WHERE *balance* > 15000)
  EXCEPT
  (SELECT DISTINCT *acc-id*
  FROM DEPOSIT T2
  WHERE T1.*cust-id* = T2.*cust-id*))
- Lets look at the 4th tuple in T1. The nested query becomes:
- (SELECT DISTINCT *acc-id*
  FROM ACC
  WHERE *balance* > 15000)
  EXCEPT
  (SELECT DISTINCT *acc-id*
  FROM DEPOSIT T2
  WHERE 2 = T2.*cust-id*))

returns {A1, A2}

returns {A2, A3}

- The above query returns {A1}. So NOT EXIST evaluates to false, and *cust-id* 2 is not displayed.

ACC

acc-id	*balance*
A1	20k
A2	18k
A3	10k

DEPOSIT

acc-id	*cust-id*	*amount*
A1	1	2k
A1	1	1k
A2	1	1k
A2	2	3k
A3	3	2k
A3	2	5k

Test for Absence of Duplicate Tuples

course (course_id, title, dept_name, credits)
section (course_id, sec_id, semester, year, building, room_number, time_slot_id)

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- **unique** predicate evaluates to true on the empty set
- Find all courses that were offered <u>at most once</u> in 2009.

  select T.course_id
  from course as T
  where **unique** (
      select R.course_id
      from section as R
      where T.course_id= R.course_id **and** R.year = 2009)

Test for Existence of Duplicate Tuples

course (course_id, title, dept_name, credits)
section (course_id, sec_id, semester, year, building, room_number, time_slot_id)

- Find all courses that were offered <u>at least twice</u> in 2009.

select T.course_id
from course as T
where **not unique** (
    select R.course_id
    from section as R
    where T.course_id= R.course_id **and** R.year = 2009)

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- 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 *branch_avg* (*branch_name, avg_balance*)
  where *avg_balance* > 1200

  Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *avg_balance* can be used directly in the **where** clause.

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all accounts with the maximum balance

  with *max_balance* (*value*) **as**
      select max (*balance*)
      from *account*
  select *account_number*
  from *account, max_balance*
  where *account.balance = max_balance.value*

Complex Queries using With Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

with *branch_total* (*branch_name*, *value*) **as**
  select *branch_name*, **sum** (*balance*)
  from *account*
  group by *branch_name*
with *branch_total_avg* (*value*) **as**
  select avg (*value*)
  from *branch_total*
select *branch_name*
from *branch_total*, *branch_total_avg*
where *branch_total.value >= branch_total_avg.value*

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

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

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

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database
  - The expression is substituted into queries using the view.

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'

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

  delete from *account*
  where *branch_name* = 'Perryridge'

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

  delete from *account*
  where *branch_name* **in** (**select** *branch_name*
  from *branch*
  where *branch_city* = 'Needham')

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

- Insert tuples on the basis of the result of a query.

  Instructor (ID, name, dept_name, salary)

- Make each student in the Music department who has earned more than 144 credit hours, an instructor in the Music department, with a salary of $18,000.

  insert into instructor
  select ID, name, dept_name, 18000
  from student
  where dept name = ’Music’ and tot_cred > 144;

  - SQL evaluates the select statement first, giving a set of tuples
  - Then inserted into the instructor relation.
  - Each tuple has an ID, a name, a dept name (Music), and an salary of $18,000.

- The **select from where** statement is evaluated fully 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* $\leq$ 10000
  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account
set balance = case
                  when balance <= 10000 then
                            balance *1.05
                  else   balance * 1.06
               end
```

Update of a View

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

  create view *loan_branch* **as**
  select *loan_number, branch_name*
  from *loan*

- Add a new tuple to *loan_branch*

  insert into *loan_branch*
  values ('L-37', 'Perryridge‘)

  This insertion must be represented by the insertion of the tuple

  ('L-37', 'Perryridge', *null*)

  into the *loan* relation

Updates Through Views (Cont.)

- Some updates through views are impossible to appear in the inserted view, for example
  - **create view** *v* **as**
    select *loan_number, branch_name, amount*
    from *loan*
    where *branch_name* = 'Perryridge'

  insert into *v* **values** **(** 'L-99', 'Downtown', '23'**)**

- Others cannot be translated uniquely
  - **insert into** *all_customer* **values (**'Perryridge', 'John')
    - *All_customer* is derived from relations *loan* and *acount*
    - Have to choose *loan* or *account*, and create a new *loan/account* number!
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause

Join operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  prereq information is missing for CS-315 and

  course information is missing for CS-437

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

Left Outer Join

- *course* **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

- *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Relations in SQL – Examples

- *course* **inner join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above and a natural join?

Joined Relations in SQL – Examples

- *course* **left outer join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Joined Relations – Examples

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- *course* **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

End of Chapter 3