Chapter 3: SQL

Contents

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, SQL:2003

History Cont.

- 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

- Allows the specification of 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).

Domain Types in SQL Cont.

- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *d* 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.
- More are covered in Chapter 4.

Create Table Construct

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

  create table *r* (*A1 D1*, *A2 D2*, ..., *An Dn*,
  (integrity-constraint1),
  ...,
  (integrity-constraintk))

  – *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

Create Table Construct

Cont.

- Example:

null values is not allowed in attribute brach_name

```
create table branch
      (branch_name  char(15) not null,
      branch_city    char(30),
      assets         integer)
```

Integrity Constraints

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

automatically ensures **not null** in SQL-92 onwards

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))
```

Drop Table Constructs

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

drop table r

Deletes not only all tuples of r, but also the schema for r

More drastic than

delete table r

Retains relation r, but deletes all tuples in r

Alter Table Constructs

- The **alter table** command is used to 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*, *D* is the domain of *A*.

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

Alter Table Constructs Cont.

- 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

Basic Query Structure

- SQL is based on relational algebra and relational calculus
- A typical SQL query has the form:

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

Attributes

Relations

Predicate

Basic Query Structure Cont.

- 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_nam*
from *loan*

corresponds to

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

The select Clause Cont.

- SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

Example

Branch_Name $\equiv$ *BRANCH_NAME* $\equiv$ *branch_name*

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

The select Clause Cont.

- To force the elimination of duplicates, insert the keyword **distinct** after select.

Example

- Find the names of all branches in the *loan* relations, and remove duplicates

select **distinct** *branch_name*
from *loan*

The select Clause**** Cont.

- The keyword **all** specifies that duplicates not be removed.

```
select all branch_name
from loan
```

- Default is **all**

The select Clause Cont.

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

```
select *
from loan
```

Equal to

```
select loan_number, branch_name, amount
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

  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

Example

- 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

The where Clause Cont.

- SQL use the logical connectives **and, or, not** in the where clause
- Operands of logical connectives can be expressions involving the comparison operation <, >, <=, >=, =, <>
- Comparisons can be applied to results of arithmetic expressions, strings, special types such as date types

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.

Example

- Find the Cartesian product *borrower* × *loan*

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

Example

- 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 attributes in the select clause using the **as** clause:

  old-name **as** *new-name*

Example

- 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.

Example

- Find the customer names and their loan numbers and loan amounts 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*

Example

- 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 that have any length (can be 0).
  - underscore (_). The _ character matches any character.

Example

- Find the names of all customers whose street includes the substring "Main".

select *customer_name*

from *customer*

where *customer_street* **like** '% Main%'

⌗ Match the string "80%"

like '80\%' **escape** '\'

String Operations Cont.

- 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

- The **order by** clause causes the tuples in the result of a query to appear in sorted order

Example

- 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*

Ordering the Display of Tuples

Cont.

- We may specify **desc** for descending order or **asc** for ascending order
- Ordering can be performed on multiple attributes

select *
from *loan*
order by *amount desc, loan_number asc*

⌗ For each attribute, ascending order is the default

Duplicates

- **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. $\boldsymbol{\Pi_A(r)}$**:** 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:

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

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

  is equivalent to the *multiset* version of the expression

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

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

Set Operations Cont.

- 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*

Example

- 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

Example

- 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
```

Count the number of tuples in a relation

- Find the number of depositors in the bank.

```
select count (distinct customer_name)
from depositor
```

eliminate duplicates

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*

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

Aggregate Functions – Having

- 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
```

- predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

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**

Null Values Cont.

- The result of any arithmetic expression involving *null* is *null*

  5 + *null* returns null

- Any comparison with *null* returns *unknown*

  5 < *null* or *null* <> *null* or *null* = *null*

null **is null** → true

null = *null* → unknown

Null Values Cont.

- 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*

Null Values Cont.

- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes

Example ■ Total all loan amounts

select sum (*amount*)
from *loan*

⌗ Above statement ignores null amounts

⌗ Result is *null* if there is no non-null amount

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

Set Membership

Set is produced by subquery

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

Test for set membership

```
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

Test for absence of set membership

```
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 )
```

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

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

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

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

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

(= **some**) ≡ **in**

However, (≠ **some**) ≢ **not in**

Example Query

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

```
select branch_name
from branch
where assets > all
        (select assets
         from branch
         where branch_city = 'Brooklyn')
```

Definition of all Clause

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

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

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

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

(5 $\neq$ **all** $\begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline\end{array}$) = true (since 5 $\neq$ 4 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 argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \varnothing$
- **not exists** $r \Leftrightarrow r = \varnothing$

Example Query

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

```
select distinct customer_name
from borrower
where exists (select *
              from depositor
              where depositor.customer_name =
                    borrower.customer_name )
```

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

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 ))
```

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* **as** *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')

- Variable from outer level is known as a **correlation variable**

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
```

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
```

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.

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

Views Cont.

- 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*

Views Cont.

- 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 Query

- 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)
```

Example Query

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

Cont.

Views Defined Using Other Views

- 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
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.

View Expansion Cont.

- View expansion of an expression repeats the following replacement step:

repeat

Find any view relation *vi* in *e1*

Replace the view relation *vi* by the expression defining *vi*

until no more view relations are present in *e1*

- As long as the view definitions are not recursive, this loop will terminate

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*)

Example Query

- 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
```

Modification of the Database Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

1.

```
update account
set balance = balance ∗ 1.06
where balance > 10000
```

2.

```
update account
set balance = balance ∗ 1.05
where balance ≤ 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 Through Views

- 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 *branch_loan*

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

⌗ represented by the insertion of the tuple ('L-37', 'Perryridge', *null*) into the *loan* relation

Update Through Views Cont.

- Some updates through views are impossible to translate into updates on the database relations

```
create view v as
      select loan_number, branch_name, amount
      from loan
      where branch_name = 'Perryridge'
```

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

Cont.

Update Through Views

- Others cannot be translated uniquely

```
insert into all_customer
values ('Perryridge', 'John')
```

  - 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

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

Joined Relations Cont.

- **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 $(A_1, A_1, \ldots, A_n)$

Joined Relations
Datasets for Examples

- Relation *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

Relation *borrower*

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- borrower information missing for L-260 and loan information missing for L-155

Joined Relations – Examples

loan **inner join** *borrower* **on**
loan.loan_number = borrower.loan_number

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

loan **left outer join** *borrower* **on**
loan.loan_number = borrower.loan_number

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Cont.

Joined Relations – Examples

loan **natural inner join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

loan **natural right outer join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

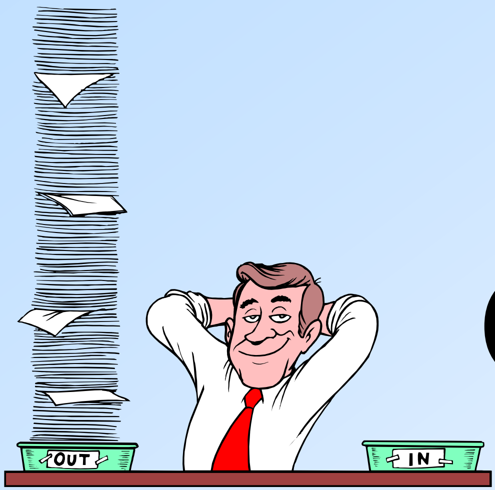
Joined Relations – Examples Cont.

loan **full outer join** *borrower* **using** (*loan_number*)

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Find all customers who have either an account or a loan (but not both) at the bank.

select *customer_name*
from (*depositor* **natural full outer join** *borrower*)
where *account_number* **is null or** *loan_number* **is null**



Conclusions

Questions?

End of Chapter 3