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
- Joined Relations**
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

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 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 relation.
- 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.
- More are covered in Chapter 4.
Create Table Construct

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

  ```sql
  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
                  (integrity-constraint_1),
                  ..., (integrity-constraint_k))
  ```

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

- Example:

  ```sql
  create table branch
  (branch_name char(15) not null,
   branch_city char(30),
   assets integer)
  ```
Integrity Constraints in Create Table

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

Example: Declare \textit{branch\_name} as the primary key for \textit{branch}.

```
create table branch
  (branch\_name char(15),
   branch\_city char(30),
   assets integer,
   primary key (branch\_name))
```

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

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

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

  \[
  \text{alter table } r \text{ add } A \ D
  \]
  
  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:

  \[
  \text{alter table } r \text{ drop } A
  \]
  
  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 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
  ```

  - $A_i$ represents an attribute.
  - $R_i$ represents a relation.
  - $P$ is a predicate.

- This query is equivalent to the relational algebra expression:

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

- The result of an SQL query is a relation.
The select Clause

The select clause 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:

```sql
select branch_name
from loan
```

In the relational algebra, the query would be:

```
Π_{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.
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
  
  ```sql
  select distinct branch_name from loan
  ```

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

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

\[
\text{select } * \\
\text{from loan}
\]

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

The query:

\[
\text{select loan_number, branch_name, amount } \ast 100 \\
\text{from loan}
\]

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.
  
  ```sql
  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, $\geq 90,000$ and $\leq 100,000$)

```sql
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 $borrower \times loan$
  
  
  ```sql
  select *
  from borrower, loan
  ```

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

  ```sql
  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:
  
  \[ \text{old-name as new-name} \]

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

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

```sql
select customer_name
from customer
where customer_street like '% Main%'
```

- Match the name “Main%”

```sql
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

  ```sql
  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`
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. \( \sigma_\theta(r_1) \): If there are \( c_1 \) copies of tuple \( t_1 \) in \( r_1 \), and \( t_1 \) satisfies selections \( \sigma_\theta \), then there are \( c_1 \) copies of \( t_1 \) in \( \sigma_\theta(r_1) \).

  2. \( \Pi_A(r) \): 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. \( r_1 \times r_2 \): 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 \( t_1 \cdot t_2 \) in \( r_1 \times r_2 \).
Duplicates (Cont.)

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

\[
\begin{align*}
  r_1 &= \{(1, a), (2, a)\} \\
r_2 &= \{(2), (3), (3)\}
\end{align*}
\]

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:

\[
\Pi_{A_1,A_2,\ldots,A_n} (\sigma_P(r_1 \times r_2 \times \ldots \times r_m))
\]
Set Operations

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

- 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:

  \[
  (\text{select } \text{customer\_name from depositor})
  \text{ union }
  (\text{select } \text{customer\_name from borrower})
  \]

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

  \[
  (\text{select } \text{customer\_name from depositor})
  \text{ intersect }
  (\text{select } \text{customer\_name from borrower})
  \]

- Find all customers who have an account but no loan.

  \[
  (\text{select } \text{customer\_name from depositor})
  \text{ except }
  (\text{select } \text{customer\_name from borrower})
  \]
Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value:
  - **avg**: average value
  - **min**: minimum value
  - **max**: maximum value
  - **sum**: sum of values
  - **count**: number of values
Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

\[
\text{select } \text{avg} \ (balance) \\
\text{from account} \\
\text{where branch\_name} = 'Perryridge'
\]

- Find the number of tuples in the \textit{customer} relation.

\[
\text{select count} \ (\ast) \\
\text{from customer}
\]

- Find the number of depositors in the bank.

\[
\text{select count} \ (\text{distinct} \ \text{customer\_name}) \\
\text{from depositor}
\]
Aggregate Functions – Group By

- Find the number of depositors for each branch.

```sql
select branch_name, count (distinct customer_name)
from depositor, account
where depositor.account_number = account.account_number
group by branch_name
```

Note: Attributes in `select` clause outside of aggregate functions must appear in `group by` list.
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.
Null Values

- It is possible for tuples to have a null value, denoted by `null`, for some of their attributes.
- `null` signifies an unknown value or that a value does not exist.
- The predicate `is null` can be used to check for null values.
  - Example: Find all loan number which appear in the `loan` relation with null values for `amount`.
    ```sql
    select loan_number
    from loan
    where amount is null
    ```
- The result of any arithmetic expression involving `null` is `null`.
  - Example: `5 + null` returns null.
- However, aggregate functions simply ignore nulls.
  - More on next slide.
Null Values and Three Valued Logic

- Any comparison with null returns unknown
  - Example: $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$

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

  ```sql
  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.
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.

```sql
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

```sql
select branch_name
from branch
where assets > some
    (select assets
     from branch
     where branch_city = 'Brooklyn')
```
Definition of Some Clause

F \text{<comp> some } r \iff \exists t \in r \text{ such that } (F \text{<comp>} t)

Where \text{<comp>} can be: <, \leq, >, =, \neq

\begin{array}{c|c|c}
\hline
\text{5} & \text{0} & \text{true} \\
\text{6} & \text{5} & \text{false} \\
\hline
\end{array}

(read: 5 < some tuple in the relation)

\begin{array}{c|c|c}
\hline
\text{0} & \text{5} & \text{true} \\
\hline
\end{array}

\begin{array}{c|c|c}
\hline
\text{0} & \text{5} & \text{true} \\
\hline
\end{array}

(5 \neq \text{some } 5) = \text{true (since } 0 \neq 5)

(= \text{some}) \equiv \text{in}

However, (\neq \text{some}) \not\equiv \text{not in}
Example Query

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

```
select branch_name
  from branch
  where assets > all
    (select assets
     from branch
     where branch_city = 'Brooklyn')
```
### Definition of all Clause

- **F <comp> all r ⇔ ∀ t ∈ r (F <comp> t)**

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 &lt; all ) = false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 &lt; all ) = true</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 = all ) = false</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 ≠ all ) = true (since 5 ≠ 4 and 5 ≠ 6)</td>
<td></td>
</tr>
</tbody>
</table>

(≠ all) ≡ not in  
However, (= all) ≠ in
Test for Empty Relations

- The `exists` construct returns the value `true` if the argument subquery is nonempty.
- `exists \ r \iff r \neq \emptyset`
- `not exists \ r \iff r = \emptyset`
Example Query

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

```sql
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 $X - Y = \emptyset \iff X \subseteq Y$
- *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.

```sql
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.

```sql
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 `branch_avg` 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

```sql
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.

```sql
with branch_total (branch_name, value) as
  select branch_name, sum(balance) as total
  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, by

\[
(\text{select} \ customer\_name, \ borrower.\text{loan}\_number, \ branch\_name \\
\text{from} \ borrower, \ loan \\
\text{where} \ borrower.\text{loan}\_number = \ loan.\text{loan}\_number )
\]

A view provides a mechanism to hide certain data from the view of certain users.

Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a view.
A view is defined using the **create view** statement which has the form

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

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

```sql
create view all_customer as
( select branch_name, customer_name
  from depositor, account
  where depositor.account_number = account.account_number )
union
( select branch_name, customer_name
  from borrower, loan
  where borrower.loan_number = loan.loan_number )
```

- Find all customers of the Perryridge branch

```sql
select customer_name
from all_customer
where branch_name = 'Perryridge'
```
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 $v_2$.
- A view relation $v$ is said to be recursive if it depends on itself.
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 of an expression repeats the following replacement step:

```
repeat
Find any view relation $v_i$ in $e_1$
Replace the view relation $v_i$ by the expression defining $v_i$
until no more view relations are present in $e_1$
```

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
  
  ```sql
  delete from account
  where branch_name = 'Perryridge'
  ```

- Delete all accounts at every branch located in the city ‘Needham’.
  
  ```sql
  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

- 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.

```sql
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 evaluated fully before any of its results are inserted into the relation (otherwise queries like

```sql
insert into table1 select * from table1
```
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
- 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
  
  \[
  \text{create view} \enspace \text{loan\_branch as}
  \]
  
  \[
  \text{select} \enspace \text{loan\_number, branch\_name}
  \]
  
  \[
  \text{from} \enspace \text{loan}
  \]

- Add a new tuple to branch\_loan
  
  \[
  \text{insert into} \enspace \text{branch\_loan}
  \]
  
  \[
  \text{values} \enspace (\text{'L-37', 'Perryridge'})
  \]

This insertion must be represented by the insertion of the tuple

\[
(\text{'L-37', 'Perryridge', null})
\]

into the loan relation
Updates 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 ( 'L­99', 'Downtown', '23')`

- 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.

- **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.

<table>
<thead>
<tr>
<th>Join types</th>
<th>Join Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner join</td>
<td><strong>natural</strong></td>
</tr>
<tr>
<td>left outer join</td>
<td>on <code>&lt;predicate&gt;</code> using (A_1, A_1, \ldots, A_n)</td>
</tr>
<tr>
<td>right outer join</td>
<td></td>
</tr>
<tr>
<td>full outer join</td>
<td></td>
</tr>
</tbody>
</table>
Joined Relations – Datasets for Examples

- Relation *loan*
- Relation *borrower*

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>Hayes</td>
<td>L-155</td>
</tr>
</tbody>
</table>

Note: 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

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
Joined Relations – Examples

- *loan* natural inner join *borrower*

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
</tbody>
</table>

- *loan* natural right outer join *borrower*

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>
Joined Relations – Examples

- \textit{loan full outer join} \textit{borower using} \textit{(loan\_number)}

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>

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

\begin{verbatim}
select customer\_name 
from (depositor natural full outer join borrower ) 
where account\_number is null or loan\_number is null
\end{verbatim}
End of Chapter 3
branch (\texttt{branch\_name, branch\_city, assets})
customer (\texttt{customer\_name, customer\_street, customer\_city})
loan (\texttt{loan\_number, branch\_name, amount})
borrower (\texttt{customer\_name, loan\_number})
account (\texttt{account\_number, branch\_name, balance})
depositor (\texttt{customer\_name, account\_number})
**Figure 3.3: Tuples inserted into loan and borrower**

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-11</td>
<td>Round Hill</td>
<td>900</td>
</tr>
<tr>
<td>L-14</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>L-17</td>
<td>Downtown</td>
<td>1000</td>
</tr>
<tr>
<td>L-23</td>
<td>Redwood</td>
<td>2000</td>
</tr>
<tr>
<td>L-93</td>
<td>Mianus</td>
<td>500</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>1900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-16</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
</tr>
<tr>
<td>Johnson</td>
<td>null</td>
</tr>
</tbody>
</table>
Figure 3.4: The *loan* and *borrower* relations

<table>
<thead>
<tr>
<th>loan_number</th>
<th>branch_name</th>
<th>amount</th>
<th>customer_name</th>
<th>loan_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>Hayes</td>
<td>L-155</td>
</tr>
</tbody>
</table>

*loan* | *borrower*