Lecture 3 SQL

Shuigeng Zhou

March 12, 2014
School of Computer Science
Fudan University

Outline

- A Short History of SQL
- Data Definition
- Basic SQL Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Derived Relations
- Views
- Modification of the Database
- Joined Relations

A Short History of SQL

- 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 (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.
- **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.
- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person-name* **char**(20) **not null**

Date/Time Types in SQL (Cont.)

- **date**. Dates, containing a (4 digit) year, month and date
  - E.g. **date** '2001-7-27'
- **time**. Time of day, in hours, minutes and seconds.
  - E.g. **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - E.g. **timestamp** '2001-7-27 09:00:30.75'
- **Interval**: period of time
  - E.g. Interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
- Can extract values of individual fields from date/time/timestamp
  - E.g. **extract** (**year from** r.starttime)
- Can cast string types to date/time/timestamp
  - E.g. **cast** <string-valued-expression> **as date**

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)
- **check** (*P*), where *P* is a predicate

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

* primary key declaration on an attribute automatically ensures not null in SQL-92 onwards, needs to be explicitly stated in SQL-89

Basic Insertion and Deletion of Tuples

- Newly created table is empty
- Add a new tuple to *account*

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

  - Insertion fails if any integrity constraint is violated
- Delete *all* tuples from *account*

  delete from *account*

  Note: Will see later how to delete selected tuples

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.

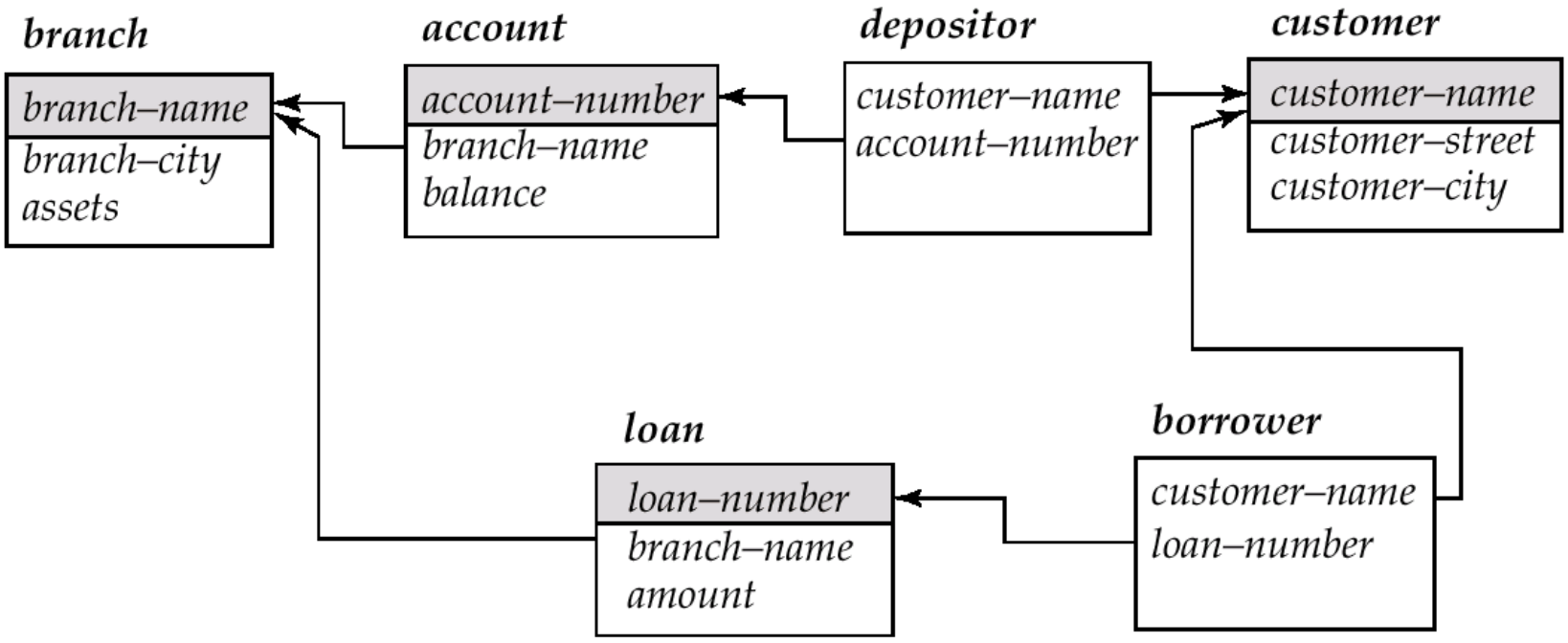
  alter table *r* **add** *A D*

  - 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*

  - Dropping of attributes not supported by many databases

Schema Used in Examples



Basic Structure

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

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

- This query is equivalent to the relational algebra (RA) expression:

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

- 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 RA
- E.g. find the names of all branches in the *loan* relation

  select *branch-name*
  from *loan*

- In the "pure" RA syntax, the query would be:

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

  - NOTE: SQL does not permit the '-' character in names,
  - NOTE: SQL names are case insensitive, i.e. you can use capital or small letters.

The select Clause (Cont.)

- SQL allows duplicates in relations
- To force the elimination of duplicates, insert the keyword **distinct** after **select**
- Find the names of all branches in the *loan* relations, and remove duplicates

```
select distinct branch-name
from loan
```

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

```
select all branch-name
from loan
```

The select Clause (Cont.)

- 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
  - Corresponding to generalized projection operation
- The query:

```
select loan-number, branch-name, amount * 100
from loan
```

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - corresponds to the selection predicate of the RA.
- 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
- E.g. 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 RA
- Find the Cartesian product *borrower x loan*

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

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

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

The Rename Operation

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

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

- Find the name, loan number and loan amount of all customers; rename the column name *loan-number* as *loan-id*:

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

Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause
- Find the customer names, 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
```

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

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

String Operations

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

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

- Match the name "Main%"

```
like 'Main\%' escape '\'
```

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

Ordering the Display of Tuples

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

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

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - E.g. **order by** *customer-name* **desc**

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result
- *Multiset* (or bag) 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 σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. $\Pi_A(r_1)$: 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. $r_1 \times r_2$: 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:

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

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; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**

  Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then, it occurs:
  - $m + n$ times in *r* **union all** *s*
  - $\min(m,n)$ times in *r* **intersect all** *s*
  - $\max(0, m - n)$ times in *r* **except all** *s*

Set Operations

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

  (select *customer-name* **from** *depositor*)
  union
  (select *customer-name* **from** *borrower*)

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

  (**select** *customer-name* **from** *depositor*)
  intersect
  (select *customer-name* **from** *borrower*)

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

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

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

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

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

```
select avg (balance)
from account
where branch-name = 'Perryridge'
```

- Find the number of tuples in the customer relation.

```
select count (*)
from customer
```

- Find the number of depositors in the bank.

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

Aggregate Functions – Group By

- Find the number of depositors for each branch.

  select *branch-name*, **count (distinct** *customer-name)*
  from *depositor, account*
  where *depositor.account-number = account.account-number*
  group by *branch-name*

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

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than $1,200.

```
select branch-name, avg (balance)
from account
group by branch-name
having avg (balance) > 1200
```

Note: predicates in the having clause are applied after the information 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*, signifies an unknown value or that a value does not exist
- The predicate **is null** can be used to check for null values

```
select loan-number
from loan
where amount is null
```

- The result of any arithmetic expression involving *null* is *null*
  - E.g. 5 + null returns null
- However, aggregate functions simply ignore nulls

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - *E.g. 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

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)
```

(Schema used in this example)

Set Comparison

- Find all branches that have greater assets than some branch located in Brooklyn

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

- Same query using > some clause

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

Definition of Some Clause

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

(5< **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

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

Definition of all Clause

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

(5< **all** {0, 5, 6}) = false

(5< **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≢ **in**

Example Query

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

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

Test for Empty Relations

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

Example Query

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

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

- (Schema used in this example)
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note*: Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

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

```
select T.customer-name
from depositor as T
where unique (
    select R.customer-name
   from account, depositor as R
   where T.customer-name = R.customer-name and
            R.account-number = account.account-number and
            account.branch-name = 'Perryridge')
```

- (Schema used in this example)

Example Query

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

```
select distinct T.customer-name
from depositor T
where not unique (
    select R.customer-name
    from account, depositor as R
    where T.customer-name = R.customer-name
    and
    R.account-number = account.account-number
    and
    account.branch-name = 'Perryridge')
```

(Schema used in this example)

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

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

- where:
  - <query expression> is any legal expression
  - The view name is represented by *v*

Example Queries

- A view consisting of branches and their customers

```
create view all-customer as
        (select branch-name, customer-name
        from depositor, account
        where depositor.account-number = account.account-number)
    union
        (select branch-name, customer-name
        from borrower, loan
        where borrower.loan-number = loan.loan-number)
```

- Find all customers of the Perryridge branch

```
select customer-name
from all-customer
where branch-name = 'Perryridge'
```

Derived Relations

- Find the average account balance of those branches where the average account balance is greater than $1200.

```
select branch-name, avg-balance
from (select branch-name, avg (balance)
        from account
        group by branch-name)
        as result (branch-name, avg-balance)
where avg-balance > 1200
```

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

With Clause

- With clause allows views to be defined locally to a query, rather than globally. Analogous to procedures in a programming language
- 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 Query using With Clause

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

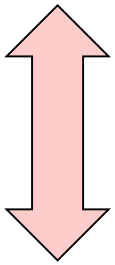
```
with branch-total (branch-name, value) as
    select branch-name, sum (balance)
    from account
    group by branch-name
with branch-total-avg(value) as
    select avg (value)
    from branch-total
select branch-name
from branch-total, branch-total-avg
where branch-total.value >= branch-total-avg.value
```

Modification of the Database – Deletion

- Delete all account records at the Perryridge branch

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

- Delete all accounts at every branch located in Needham city



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

- (Schema used in this example)

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 *account*, the average balance changes
  - Solution used in SQL:
    - 1. First, compute **avg** balance and find all tuples to delete
    - 2.Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *account*

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

  or equivalently

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

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

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

Modification of the Database – Insertion

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

```
insert into account
    select loan-number, branch-name, 200
    from loan
    where branch-name = 'Perryridge'
insert into depositor
    select customer-name, loan-number
    from loan, borrower
    where branch-name = 'Perryridge'
                and loan.account-number = borrower.account-number
```

- The select from where statement is fully evaluated before any of its results are inserted into the relation (otherwise queries like

  insert into *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 *loan* relation, hiding the *amount* attribute

  create view *branch-loan* **as**
  select *branch-name, loan-number*
  from *loan*

- Add a new tuple to *branch-loan*

  insert into *branch-loan*
  values ('Perryridge', 'L-307')

  This insertion must be represented by the insertion of the tuple

  ('L-307', 'Perryridge', *null*)

  into the *loan* relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - **commit work:** makes all updates of the transaction permanent in the database
    - **rollback work:** undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by **rollback work.**
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within

```
begin atomic
    ...
end
```

Joined Relations

- Join operations take two relations and return as a result another relation
- 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_2, …, 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

- Relation borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

- 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

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*

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

- *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*

Homework

- Exercises: 3.8, 3.9, 3.10, 3.21, 3.22, 3.23

End of Lecture 3