Chapter 3 Introduction to SQL

Rui Meng
United International College

March 28, 2019

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Value
- Nested Subquery
- View
- Database Modification
- Joined Relations

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
  - SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features
  - Not all examples here may work on your particular system

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation
- The domain of values associated with each attribute
- Integrity constraints
- And as we will see later, also other information such as
  - 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 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. (e.g., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32).
- **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.

Creating Table Construct

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

$$\textbf{create table } r(A_1 D_1, A_2 D_2, \ldots, A_n D_n,\\ (\text{integrity-constraint}_1),\\ \vdots\\ (\text{integrity-constraint}_k))$$

  - 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 varchar(15) **not null**,
branch_city varchar(30),
assests integer)

Integrity Constraints in Create Table

- **not null**
- **primary key** $(A_1, ..., A_n)$
- **foreign key** $(A_m, ..., A_n)$ **references** *r*

Example:

> **create table** *branch*(
> branch_name varchar(15) **not null**,
> branch_city varchar(30),
> assests integer,
> **primary key** (branch_name),
> **foreign key** (branch_city) **references** *city*)

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

Drop and Alter Table Constructs

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

  drop table r

- 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 and D is the domain of A.
  - All tuples in the relation are assigned **null** as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation:

  alter table r drop A

  - where A is the name of an attribute of relation r.
  - Dropping of attributes not supported by many databases (supported by MySQL).

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Value
- Nested Subquery
- View
- Database Modification
- Joined Relations

Basic Query Structure

- 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_j represents a relation
  - P is a predicate
- This query is equivalent to the following relational algebra expression:

$$\pi_{A_1, A_2, \cdots, A_n}(\sigma_p(r_1 \times r_2 \times \cdots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

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

  select *branch_name*
  from *loan*

- This query is equivalent to the following relational algebra expression:

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

- SQL names are case insensitive (i.e., you may use upper- or lower- case letters)
  - E.g., $Branch\_Name \equiv BRANCH\_NAME \equiv branch\_name$

The select Clause (Cont.)

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

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

  select all *branch_name*
  from *loan*
- Since duplicate retention is the default, we shall not use all in our examples.

The select Clause (Cont.)

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

  select *
  from *loan*

- An attribute can be a literal with no from clause, as a temporary constant table

  select '437'

  Results is a table with one column and a single row with value "437". Can give the column a name using:

  select '437' **as** *FOO*

- An attribute can be a literal with **from** clause

  select 'A'
  from *loan*

- Result is a table with one column and N rows (number of tuples in the instructors table), each row with value "A".

The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, $+, -, *$, and / and operating on .
- The query:

  select *loan_number, branch_name, amount* * 100
  from *loan*

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

The where Clause

- The where clause specifies conditions that the result must satisfy.
  - Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200:

  select *loan_number*
  from *loan*
  where *branch_name* = 'Perryridge' **and** *amount* $\geq$ 1200
- Comparison results can be combined using the logical connectives **and**, **or**, and **not**.
- Comparisons can be applied to results of arithmetic expressions.
- The operands of the logical connectives can be expressions involving the comparison operators $<$, $\leq$, $>$, $\geq$, $=$ and $\neq$.

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 $\leq 100,000$):

  select *loan_number*
  from *loan*
  where amount **between** 90000 **and** 100000

- Similarly, we can use the **not between** comparison operator.

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 *borrow* $\times$ *loan*:

  select $*$ from *borrow*, *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'

- Specify the matching condition in **where** clause.

The rename Operation

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

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

- The **as** clause can appear in both the **select** and **from** clauses.
  - Rename the attribute name in the query result;
  - Rename relations.
- 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 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*

  - The identifier *T* and *S*, that is used to rename a relation is referred to as a **table alias** or **tuple variable**.
- 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'*

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

  select *customer_name*
  from *customer*
  where *customer_street* **like** '%Main%'

  - To match the name "Main%", we use backslash (\) as the escape character:

    like 'Main\%' **escape** '\'

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*

- 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**
  - Example: **order by** *customer_name* **desc**, *amount* **asc**

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- Each duplicate is considered as a distinct tuple.
- **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 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_i.
  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:

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

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

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

$$\pi_{A_1, A_2, \cdots, A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Value
- Nested Subquery
- View
- Database Modification
- Joined Relations

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 (Cont.)

- 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*)

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Value
- Nested Subquery
- View
- Database Modification
- Joined Relations

Aggregate Functions

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

Aggregate Function	**Description**
avg	average value
min	minimum value
max	maximum value
sum	sum of values
count	number of values

Aggregate Functions (Cont.)

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

- Find the 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

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

- Find the 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

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

- Find the names of all branches 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.

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subquery
- View
- Database Modification
- Joined Relations

Null Values

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

      select(*loan_number*)
      from *loan*
      where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
    - Example: 5+*null* returns *null*.
- However, aggregate functions simply ignores nulls.
    - More on next slide.

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.
- What if collection has only null values?
  - **count** returns 0
  - All other aggregates return *null*.

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*.
    - Example: $5 < null$ or $null \neq null$ 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: (*unknown* **and** *true*) = *unknown*,
      (*unknown* **and** *false*) = *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

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- View
- Database Modification
- Joined Relations

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

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

  as follows:
  - A_i can be replaced be a subquery that generates a single value.
  - r_i can be replaced by any valid subquery.
  - P can be replaced with an expression of the form: B *<operation>(subquery)* , where B is an attribtue and <operation> to be defined later.

Subquery in where Clause

- A common use of subqueries in the *where* clause is to perform tests:
  - For set membership.
  - For set comparisons.
  - For set cardinality.

Set Memership

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

- Find all customers who have both an account and a loan at the bank:
  select distinct *customer_name*
  from *borrower*
  where *customer_name* **in** (**select** *customer_name*
  from *depositor*)
- Find all customers who have a loan at the bank but do not have an account at the bank:
  select distinct *customer_name*
  from *borrower*
  where *customer_name* **not in** (**select** *customer_name*
  from *depositor*)

Set Memership (Cont.)

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

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

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

- Find all branches that have greater assets than some branch located in Brooklyn:
  select distinct *T.branch_name*
  from *branch* as *T*, *branch* as *S*
  where *T.assets* > *S.assets* **and** *S.branch_city*='Brooklyn'

- Same query using >**some** clause:
  select *branch_name*
  from *branch*
  where *assets* >**some**
  (**select** *assets*
  from *branch*
  where *branch_city* = 'Brooklyn')

Definition of "some" Clause

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

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

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

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

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

- Note:
  - (=**some**)$\equiv$ **in**.
  - However, ($\neq$ **some**) is not equivalent to **not in**.

Set Comparison (Cont.)

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

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

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

Definition of "all" Clause

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

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

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

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

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

- Note:
  - ($\neq$**all**)$\equiv$ **not in**.
  - However, (=**all**) is not equivalent to **in**.

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** $r \Leftrightarrow r = \emptyset$.

Use of "exists" Clause

- Find all customers who have an account at all branches located in Brooklyn:
  select distinct *S.customer_name*
  from *depositor* **as** *S*
  where not exists(
  (**select** *branch_name*
  from *branch*
  where *branch_city*='Brooklyn')
  except
  (**select** *R.branch_name*
  from *depositor* **as** *T*, *account* **as** *R*
  where *T.account_number* = *R.account_number* **and**
  S.customer_name=*T.customer_name*))
- Note:
  - $X - Y = \emptyset \Leftrightarrow X \subseteq Y$.
  - Cannot write this query using =**all** and its variants.

Correlated subquerry

- A **correlated subquery** (**synchronized subquery**) is a subquery that uses values from the outer query.
- For **each row** processed by the outer query, the subquery is evaluated **once**. It can be inefficient.

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'*)
- Not that **unique** is not widely implemented, MySQL does not support such mechanism.

Test for Absence of Duplicate Tuples (Cont.)

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

Subqueries in from Clause

- SQL allows a subquery expression to be used in the **from** clause.
- Example: Find the average account balance of those branches where the average account balance is greater than $1200.

  select *branch_name, avg_balance*
  from (**select** *branch_name*, **avg** (*balance*)
          from *account*
          group by *branch_name*)
          as *branch_avg (branch_name, avg_balance)*
  where *avg_balance* $>$ 1200
- Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *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.
- Example: 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* $\geq$ *branch_total_avg.value*

Subqueries in the Select Clause

- Scalar subquery is one which is used where a single value is expected.
- List all branches along with the total amount deposits in each branch:

  select *branch_name*,
  (**select sum**(*balance*)
  from *account* **as** *T*
  where *T.branch_name* = *account.branch_name*)
  as *value*
  from *account*

- Runtime error if subquery returns more than one result tuple.

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Database Modification
- Joined Relations

Views

- In some cases, it is not desirable for all users to see the entire **logical model** (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customers name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by
  (**select** *customer_name, borrower.loan_number, branch_name*
  from *borrower, loan*
  where *borrower.loan_number* = *loan.loan_number*)
  item A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

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

Query with View

- A view consisting of branches and their customers:

  create view *all_customer* **as**

  (**select** *branch_name, customer_name*
  from *depositor, account*
  where *depositor.account_number =account.account_number*)
  union
  (**select** *branch_name, customer_name*
  from *borrower, loan*
  where *borrower.loan_number = loan.loan_number*)

- Find all customers of the Perryridge branch:

  select *customer_name*
  from *all_customer*
  where *branch_name* = 'Perryridge'

Views Defined Using Other Views

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

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Database Modification
- Joined Relations

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')

Deletion (Cont.)

- 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 adopted 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)

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*)

Insertion (Cont.)

- 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.loan_number*
      from *loan*, *borrower*
      where *branch_name* = 'Perryridge'
          and *loan.loan_number* = *borrower.loan_number*
- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like **insert into** *table1* **select** ∗ **from** *table1* would cause problems)

insert into *table1* **select * from** *table1* would cause problems)

Updates

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

    update *account*
    set *balance* $=$ *balance*$*$*1.06*
    where *balance* $>$ 10000

    update *account*
    set *balance* $=$ *balance*$*$*1.05*
    where *balance* $\leq$10000
  - The order is important.
  - Can be done better using the case statement (next slide).

Case Statement for Conditional Updates

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

  update *account*
  set *balance* = **case**
  when *balance* $\leq$10000 **then** *balance* $*$1.05
  else *balance* $*$1.06
  end

- The general form of the case statement is as follows:

  case
  when pred1 **then** result1
  when pred2 **then** result2
  ...
  when predn **then** resultn
  else result0
  end

Update of a View

- Create a view of all loan data in the loan relation, hiding the amount attribute:
  create view *branch_loan* **as**
  select *loan_number, branch_name*
  from *loan*
- Add a new tuple to *branch_loan*:
  insert into *branch_loan*
  values ('L−37', 'Perryridge')
  This insertion must be represented by the insertion of the tuple ('L−37', 'Perryridge', null) into the *loan* relation.

Update of a View (Cont.)

- Some updates through views are impossible to be translated into "real" updates on the view: **create view** *v* **as**
  select *loan_number, branch_name, amount*
  from *loan*
  where *branch_name* = 'Perryridge'
  insert into *v* **values** ('L-99', 'Downtown','23')
- Others cannot be translated uniquely:
  insert into *all_customer* **values** ('Perryridge', 'John')
  - Have to choose loan or account and create a new loan or account number!
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation.

create a new loan/account number!

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Database Modification
- Joined Relations

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.
  - **natural**: retain tuples with same value on common attributes (do not repeat attributes).
  - **on** <predicate>: allows a general predicate over the relations being joined (repeat attribtues).
  - **using** $(A_1, A_2, \cdots, A_n)$: a form of natural join only requires values to match on specific attribtues.

Joined Relations

- **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.
  - **inner join** or **join**, join operations that do not preserve nonmatched tuples.
  - **left outer join**: preserves nonmatched tuples in the first relation.
  - **right outer join**: preserves nonmatched tuples in the second relation.
  - **full outer join**: preserves nonmatched tuples in both relations.

Joined operations

- Relation *course*:

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

- Relation *prereq*:

course_id	pre_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that:
  - *prereq* information is missing for CS-315
  - *course* information is missing for CS-437

Outer Join

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

Natural Left Outer Join

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

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

Natural Right Outer Join

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

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

Natural Full Outer Join

- *course* **natural full outer join** *prereq*
  - Not supported in MySQL, use left outer join union right outer join instead.

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

Joined Conditions

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

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

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

Joined Conditions

- *course* **left outer join** *prereq* **on** $course.course\_id = prereq.course\_id$

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

Joined Conditions (Cont.)

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

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