Chapter 4: SQL

Topics to be covered

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

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003…….& so on.

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
- 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 .** An integer value.
- **Smallint.** A small integer.
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. Numeric(3,1)=44.7
- **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.
- **Date.** yyyy-mm-dd (date '2003-06-12')
- **Time.** hh:mm:ss (time '09:45:12')
- **Timestamp.** A combination of date and time (timestamp '2003-06-12 09:45:12'

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.

Data Definition Command

- Create
- Alter
- Drop
- Truncate

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)
- **Foreign key**

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

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

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

Drop and Alter Table Constructs

- 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*

  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

Data Manipulation Command

- Insert
- Delete
- Update
- Rename
  - Select
  - Where
  - From
  - String operations
  - Set operations
  - Aggregate functions

Modification of the Database – Insertion

- Add a new tuple to *account*

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

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('BOI', 1200, 'A-9732')

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

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

Modification of the Database – Deletion

- Delete all account tuples at the BOI branch

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

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

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
```

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*

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*

The select Clause (Cont.)

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

```
select *
from loan
```

- The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
- The query:

```
select loan_number, branch_name,
    amount ∗ 100
from loan
```

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

The where Clause

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

```
select loan_number
       from loan
       where branch_name = 'BOI' and amount > 1200
```

- Comparison results can be combined using the logical connectives **and, or,** and **not.** (operators <, <=, >, >=, =, and <>)
- Comparisons can be applied to results of arithmetic expressions.

The where Clause (Cont.)

- SQL includes a **between** comparison operator
- Example: Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)

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

OR

```
select loan_number
    from loan
    where amount >=90000 and amount<=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 X loan*

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

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

```
select customer_name, borrower.loan_number, amount
        from borrower, loan
        where   borrower.loan_number = loan.loan_number
                and  branch_name = 'BOI'
```

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*

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.

- 'Perry%' matches any string beginning with "Perry"
- '%idge%' matches any string containing "idge" as a substring
- '---' matches any string of exactly three characters
- '---%' matches any string of at least three characters
- like 'ab\%cd%' matches all strings beginning with "ab%cd"
- like 'ab\\cd%' matches all strings beginning with "ab\cd"

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

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

Ordering the Display of Tuples

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

```
select distinct customer_name
from    borrower, loan
where borrower.loan_number = loan.loan_number
                and
      branch_name = 'BOI'
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**

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**.

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 BOI branch.

```
select avg (balance)
        from account
        where branch_name = 'BOI'
```

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

Null Values

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

```
select loan_number
from loan
where amount is null
```

- Aggregate functions simply ignore nulls

Null Values and Aggregates

- 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.

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 BOI branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number =
                loan.loan_number and
              branch_name = 'BOI'  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.

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*

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')
```

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.

- **not exists (returns true if argument subquery is empty)**

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 BOI 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 = 'BOI')
```

Example Query

- Find all customers who have at least two accounts at the BOI 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 = 'BOI')
```

Views

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

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

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

View Definition

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

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

  where <query expression> is any legal SQL expression. The view name is represented by *v.*

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

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 BOI branch

```
select customer_name
        from all_customer
        where branch_name = 'BOI'
```

Data Control Command

GRANT COMMAND

- Grant < database_priv [database_priv…..] >to<user_name> identified by <password> [,<password…..];
- Grant <object_priv> | All on <object> to <user | public>[With Grant Option];

- [With Grant Option] – Allows the recipient user to give further grants on the objects.

Data Control Command

REVOKE COMMAND

- Revoke <database_priv> from <user [, user]>;
- Revoke <object_priv> on <object> from < user | public >;

- <database_priv> -- Specifies the system level privileges to be granted to the users or roles. This includes create / alter / delete any object of the system.
- <object_priv> -- Specifies the actions such as alter / delete / insert / references / execute / select / update for tables.
- <all> -- Indicates all the priveleges.
- The privileges can be granted to different users by specifying their names or to all users by using the “Public” option.

References:

- Korth, Slberchatz, Sudarshan,"Database System Concepts",5thEdition,McGraw–Hill