Comp 5311 Database Management Systems

3. Structured Query Language 1
Aspects of SQL

- Most common Query Language – used in all commercial systems
- Discussion is based on the SQL92 Standard.
  Commercial products have different features of SQL, but the basic structure is the same
- **Data Manipulation Language**
- Data Definition Language
- Constraint Specification
- Embedded SQL
- Transaction Management
- Security Management .....
Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:
  
  ```sql
  select A_1, A_2, ..., A_n
  from R_1, R_2, ..., R_m
  where P
  ```

  - A_i represent attributes
  - R_i represent relations
  - 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 (but may contain duplicates). SQL statements can be nested.
Projection

- The **select** clause corresponds to the **projection** operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- Find the names of all branches in the *loan* relation
  
  ```sql
  select branch-name
  from loan
  ```
  
  Equivalent to: $\Pi_{\text{branch-name}}(\text{loan})$

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

- **Note**: for our examples we use the tables:
  - Branch (**branch-name**, **branch-city**, assets)
  - Customer (**customer-name**, **customer-street**, **customer-city**)
  - Loan (**loan-number**, amount, **branch-name**)
  - Account (**account-number**, balance, **branch-name**)
  - Borrower (**customer-name**, **loan-number**)
  - Depositor (**customer-name**, **account-number**)
Duplicate Removal

- SQL allows duplicates in relations as well as in query results. Use `select distinct` to force the elimination of duplicates.
  Find the names of all branches in the loan relation, and remove duplicates

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

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

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

*force* the DBMS to remove duplicates

*force* the DBMS not to remove duplicates
Arithmetic Operations on Retrieved Results

- The `select` clause can contain arithmetic expressions involving the operators, $+,-,\div$ and $\times$, and operating on constants or attributes of tuples.

- The query:

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

would return a relation which is the same as the loan table, except that the attribute amount is multiplied by 100.
The where Clause

- The **where** clause specifies conditions that tuples in the relations in the **from** clause must satisfy.
- Find all loan numbers 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
  ```

- SQL allows logical connectives **and**, **or**, and **not**. Arithmetic expressions can be used in the comparison operators.
- Note: attributes used in a query (both **select** and **where** parts) must be defined in the relations in the **from** clause.
The where Clause (Cont.)

- SQL includes the *between* operator for convenience.
- 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 corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product borrower \( \times \) loan

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

It is rarely used without a where clause.
- Find the name and loan number of all customers having a loan at the Perryridge branch.

\[
\text{select distinct customer-name, borrower.loan-number} \\
\text{from borrower, loan} \\
\text{where borrower.loan-number = loan.loan-number and} \\
\text{branch-name = “Perryridge”}
\]
The Rename Operation

- Renaming relations and attributes using the as clause:
  old-name as new-name
- Find the name and loan number of all customers having a loan at the Perryridge branch; replace the column name loan-number with the name loan-id.

```sql
select distinct customer-name, borrower.loan-number as loan-id
from borrower, loan
where borrower.loan-number = loan.loan-number and
  branch-name = "Perryridge"
```
Tuple Variables/Alias

- 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.

```sql
select distinct customer-name, T.loan-number
from borrower as T, loan as S
where T.loan-number = S.loan-number
```

- Tuple variable/Alias can be used as short hand, but it is more than just a short hand (see next slide)
• Find the names of 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”
```

Does it returns branches within Brooklyn?
String Operations

- Character attributes can be compared to a pattern:
  - % matches any substring.
  - _ matches any single character.
- Find the name of all customers whose street includes the substring ‘Main’. (Eg Mainroad, Smallmain Road, AMainroad,...)
  ```sql
  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 at 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
```

- `order by` customer-name desc, amount asc
  `desc` for descending order; `asc` for ascending order (default)

- SQL must perform a sort to fulfill an `order by` request. Since sorting a large number of tuples may be costly, it is desirable to sort only when necessary.
Set Operations

- The set operation union, intersect, and except operate on relations and correspond to the relational algebra operations ∪, ∩ and −.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use 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:
  \[(\text{select customer-name from depositor})\]
  \[\text{union}\]
  \[(\text{select customer-name from borrower})\]

- Find all customers who have both a loan and an account.
  \[(\text{select customer-name from depositor})\]
  \[\text{intersect}\]
  \[(\text{select customer-name from borrower})\]

- Find all customers who have an account but no loan.
  \[(\text{select customer-name from depositor})\]
  \[\text{except}\]
  \[(\text{select customer-name from borrower})\]
Every SQL statement returns a relation/set in the result; remember a relation could be null or merely contain a single atomic value.

You can replace a value or set of values with a SQL statement (i.e., a subquery):

```sql
select * from loan where amount > 1200
```

```sql
select * from loan where amount > (select avg(amount) from loan)
```

Illegal if the subquery returns the wrong type for the comparison.
Example Query - IN

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

```sql
select distinct customer-name
from borrower
where customer-name in (select customer-name from depositor)
```

Check for each borrower if he/she is also a depositor

Return the set of depositors
Example Query – NOT IN

- Find all customers who have a loan at the bank but do not have an account at the bank.

```sql
select distinct customer-name
from borrower
where customer-name not in (select customer-name
from depositor)
```
The **Some** Clause

- Find all branches that have greater assets than some branch located in Brooklyn
  - Equivalent to “find all branches that have greater assets than the minimum assets of any branch located in Brooklyn”

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

Assets of all branches in Brooklyn
Some Semantics

(5 < some \( \begin{array}{c} 0 \\ 5 \\ 6 \end{array} \)) returns true (5 < 6)

(5 < some \( \begin{array}{c} 0 \\ 5 \\ 6 \end{array} \)) returns false

(5 = some \( \begin{array}{c} 0 \\ 5 \\ 6 \end{array} \)) = true

(5 \( \neq \) some \( \begin{array}{c} 0 \\ 5 \\ 6 \end{array} \)) = true (since 0 \( \neq \) 5)

Note:

(= some) is equivalent to in

However, (\( \neq \) some) is not equivalent to not in
The **All** Clause

- Find the names of all branches that have greater assets than *all* branches located in Brooklyn.
  - Equivalent to “find all branches that have greater assets than the maximum assets of any branch located in Brooklyn”

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

Assets of all branches in Brooklyn
All Semantics

\[
\begin{array}{c|c}
5 < \text{all} & \begin{array}{c}
0 \\
5 \\
6
\end{array} = \text{false} \\
5 < \text{all} & \begin{array}{c}
6 \\
10
\end{array} = \text{true} \\
5 = \text{all} & \begin{array}{c}
4 \\
5
\end{array} = \text{false} \\
5 \neq \text{all} & \begin{array}{c}
6 \\
10
\end{array} = \text{true}
\end{array}
\]

Note:

\( \neq \text{all} \) is equivalent to \textit{not in}

However, \( = \text{all} \) is \textit{not} equivalent to \textit{in}
Test for Empty Relations

- **exists** returns `true` if the argument subquery is nonempty.

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

  ```sql
  select customer-name from depositor as D where exists
  (select * from borrower as B where D.customer-name = B.customer-name)
  ```

- Find all customer names who have an account but no loan.

  ```sql
  select customer-name from depositor as D where not exists
  (select * from borrower as B where D.customer-name = B.customer-name)
  ```
Test for Absence of Duplicate Tuples

- `unique` tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have only 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")
```

For each depositor T, check ...

Find depositors with same name as T

Customers at Perryridge with same name as T
Example Query – NOT UNIQUE

- Find all customers with at least 2 accounts at the Perryridge branch.

```sql
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”)
```
Division in SQL

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

For each customer S, check ...

select distinct S.customer-name
from depositor as S
where not exist ( 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) )

Branches in Brooklyn
where customer S
doesn’t have an account

Branches where
customer S
has an account