**4.3 Basic Structure of SQL Queries**

* The basic structure of an SQL expression consists of three clauses:

**1. select** –It is used to list the attributes desired in the result of the query.

**2. from** – It lists the relations to be scanned in the evaluation of the expression.

**3. where** – It consists of a predicate involving attributes of the relations that appear in the **from** clause.

**4.3.1 The select Clause**

a) Find the names of all branches in the loan relation.

**select** branch\_name **from** loan

* Since duplicate elimination is time-consuming, SQL allows duplicates in relations as well as in the results of SQL expressions. But duplicates are not allowed in relational algebra because they are considered as set.
* To force the elimination of duplicates, a keyword **distinct** is inserted after **select**.

**select distinct** branch\_name **form** loan

* The asterisk “\*” symbol can be used to denote “all attributes”

**select** \* **from** loan

* The select clause may also contain arithmetic expressions involving the operators +, -, \* and / operating on constants or attributes of tuples.

**select** loan\_no, amount \* 100 **from** loan

**4.3.2 The where Clause**

a) Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than 300.

**select** \*

**from** loan

**where** branch\_name=’Perryridge’ **and** amount > 1300

* SQL uses the logical connectives **and**, **or** and **not.** The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, = and <>. SQL allows us to use comparison operators to compare strings and arithmetic expressions as well as special types such as dates.
* SQL includes a **between** comparison to simplify where clause that specify <= and >=. It is also possible to use **not between** operators.

b) Find the loan number of those loans with loan amounts between 400 and 800.

**select** \* **from** loan

**where** amount **between** 1300 **and** 1500

or

**select** \*

**from** loan

**where** amount >= 1300 **and** amount <= 1500

**4.3.3 The from Clause**

a) For all customers who have a loan from the bank, find their names, loan numbers and loan amount.

**select** customer\_name, loan.loan\_number, amount

**from** borrower, loan

**where** loan.loan\_no = borrower.loan\_no

**4.3.4 The Rename Operation**

* SQL provides a mechanism for renaming both relations and attributes. It uses **as** clause taking the form: old-name **as** new-name. The **as** clause can appear in both the **select** and **from** clause.
* The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.
* The rename of attributes is **needed** because:

1. Two relations in the **from** clause may have attributes with the same name, so an attribute name is duplicated in the result.

2. If we use an arithmetic expression in the **select** clause, the resultant attribute does not have a name.

3. If an attribute name can be derived from the base relation, we may want to change the attribute name in the result to make it more meaningful.

**select** customer\_name, loan.loan\_no **as** loan\_id, amount

**from** borrower, loan

**where** loan.loan\_no = borrower.loan\_no

**4.3.5 Tuple Variables**

a) Find all customers who have a loan from the bank, find their names, loan numbers and loan amount.

**select** customer\_name, L.loan\_no, amount

**from** borrower **as** B, loan **as** L

**where** L.loan\_no = B.loan\_no

**4.3.6 String Operation**

SQL specifies strings by enclosing them in single quotes, for example ‘Mirpur’. To represent ‘it’s ok’ we use ‘it’’s ok’

1. Pattern matching operation is done using operator **like** with two special characters. Patterns are case sensitive.

i) Percent (%) – matches any substring

ii) Underscore (\_) – matches any character

Examples:

a) ‘Mir%’ – ‘Mirpur’, ‘Mirzafar’

b) ‘%gla%’ – ‘Bangladesh’, ‘Bangla Motor’

c) ‘Book%’ – ‘Book Store’, ‘Book Fair’, ‘Booking’

d) ‘\_ \_ \_’ – any string of exactly three characters

e) ‘\_ \_ \_ %’ – any string of at least three characters

a) Find the information of all customers whose street address includes the substring ‘Main’.

**select** \*

**from** customer

**where** customer\_street **like** ‘%Main%’

SQL allows **not like** for searching mismatches.

2. SQL permits a variety of functions on character strings:

a) concatenating (using “||”) - [ ‘Computer’ || ‘ ‘|| ‘Science’] – “Computer Science”

b) extracting substrings - substr(‘Computer’, 4, 5) – “puter”

c) finding length of string - LENGTH(‘Computer’) – 8

d) conversion between uppercase and lowercase – UPPER(char), Lower(char)

**4.3.7 Ordering the Display of Tuples**

* SQL offers the user some control over the order on which tuples in a relation are displayed. The **order by** clause is used for sorting.

a) Find the list of all customers in alphabetic order who have a loan in the Perryridge branch.

**select distinct** customer\_name

**from** borrower **as** B, loan **as** L

**where** L.loan\_no = B.loan\_no **and** branch\_name = ‘Perryridge’

**order by** customer\_name

* By default, the **order by** clause lists items in ascending order. We can specifically assign **asc** and **desc** for ascending and descending order respectively. Also, ordering can be performed on multiple columns.

**select \*** **from** loan

**order by** amount **desc**, loan\_no **asc**

**4.4 Set Operations**

**Comparison:**

SQL union intersect except/minus

RA union (∪) set-intersection (∩) set-difference (-)

* Like relational algebra set operations, the relations participating in the set operations in SQL must be compatible; that is, they must have the same set of attributes.

**4.4.1 The Union Operation**

a) Find all customers having a loan, an account or both at the bank.

(**select** customer\_name **from** depositor)

**union**

(**select** customer\_name **from** borrower)

* Unlike the **select** clause, the **union** operation automatically eliminates duplicates. If we want to retain all duplicates:

(**select** customer\_name **from** depositor)

**union all**

(**select** customer\_name **from** borrower)

**4.4.2 The Intersect Operation (MySQL does not support)**

a) Find all customers who have both a loan and an account at the bank.

(**select** customer\_name **from** depositor)

**intersect**

(**select** customer\_name **from** borrower)

The **intersect** operation automatically eliminates duplicates.

**4.4.3 The Except Operation (Minus) (MySQL does not support)**

a) Find all customers who have an account but no loan at the bank.

(**select** customer\_name **from** depositor)

**minus**

(**select** customer\_name **from** borrower)

The **except** operation automatically eliminates duplicates.

**4.5 Aggregate Functions**

* Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

1. Average: **avg**

2. Minimum: **min**

3. Maximum: **max**

4. Total: **sum**

5. Count: **count**

* The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collection of non-numeric data types, such as string, as well.
* Aggregate functions on **a single set of tuples**:

a) Find the average account balance.

**select** **avg**(balance)

**from** account

b) Find the total account balance of Downtown branch.

**select** **sum**(balance)

**from** account

**where** branch\_name = ‘Downtown’

* Aggregate functions on **a group of sets of tuples**:

c) Find the average account balance, maximum account balance at each branch.

**select** branch\_name, **avg**(balance), **max**(balance)

**from** account

**group by** branch\_name

* **Retaining duplicates** is important in computing sum or average. There are some cases where we must eliminate duplicate before computing an aggregate function.

d) Find numbers of depositors for each branch.

**select** branch\_name, **count**(**distinct** customer\_name)

**from** depositor D, account A

**where** D.account\_number = A. account\_number

**group by** branch\_name

* At times it is useful to state a condition that applies to groups rather than to tuples.

e) Find branch name and average balance where average balance is greater than 1200.

**select** branch\_name, **avg**(balance)

**from** account

**group by** branch\_name

**having** **avg**(balance) > 1200

* At times, we wish to treat the entire relation as a single group. In such cases, **group by** is not used.

f) Find the minimum loan amount.

**select** **min**(amount) **from** loan

* The aggregate function **count** is frequently used to count the number of tuples in a relation.

g) Count the number of tuples in customer relation.

**select** **count**(\*) **from** customer

* SQL does not allow the use of **distinct** with **count**(\*). **distinct** can be used with **min** and **max**, but result does not change.

**4.6 Null Values**

a) Find all loan numbers which appear in the *loan* relation with null values for *amount.*

**select** loan\_no

**from** loan

**where** amount **is null**

* The use of a null value in arithmetic and comparison operations causes several complications. The result of any arithmetic expression involving null returns null. So 5 + null returns null. Any comparison with null returns unknown (other than **is null** and **is not null**). So, 5 < null or null <> null or null = null returns **unknown**.

**select sum**(amount)

**from** loan

ignores null amount. Result is null if there is no non-null amount.

* All aggregate functions except **count**(\*) ignore tuple with null values on the aggregated attributes.

**4.7 Nested Subqueries**

* A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to i) perform tests for set membership and ii) make set comparisons

**4.7.1 Set Membership**

* The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of the set membership.

Membership in a **one-attribute** relation:

a) Find all customers who have both a loan and an account at the bank.

**select** **distinct** customer\_name

**from** borrower

**where** customer\_name **in** (**select** customer\_name **from** depositor)

or

**select** **distinct** customer\_name

**from** depositor

**where** customer\_name **in** (**select** customer\_name **from** borrower)

b) Find all customers who have a loan but no account at the bank.

**select** **distinct** customer\_name

**from** borrower

**where** customer\_name **not** **in** (**select** customer\_name **from** depositor)

c) Find all customers who have an account but no loan at the bank.

**select** customer\_name

**from** depositor

**where** customer\_name **not** **in** (**select** customer\_name **from** borrower)

* The **in** and **not** in can also be used on enumerated sets.

d) Find all customers who have a loan at the bank and whose names are neither Rahim nor karim.

**select** **distinct** customer\_name

**from** borrower

**where** customer\_name **not** **in** (‘Rahim’, ‘Karim’)

**4.7.2 Set Comparison**

a) Find the names of all branches that have assets greater than at least one branch located in Brooklyn.

* The phrase **“greater than at least one”** is represented in SQL by > **some**

**select** branch\_name

**from** branch

**where** assets >= **some** (**select** assets

**from** branch

**where** branch\_city = ‘Horseneck’)

* SQL also allows < **some**, <= **some**, >= **some**, = **some** and <> **some** comparisons. Note that, = **some** is identical to **in**, whereas <> **some** is not the same as **not in**.
* The construct > all corresponds to the phrase **“greater than all”**.

b) Find the names of all branches that have an asset value greater than that of each branch located in Horseneck

**select** branch\_name

**from** branch

**where** assets > **all** (**select** assets

**from** branch

**where** branch\_city = ‘Horseneck’)

* SQL also allows < **all**, <= **all**, >= **all**, = **all** and <> **all** comparisons. Note that, <> **all** is identical to **not** **in.**

c) Find the branch that has the highest average balance.

**select** branch\_name

**from** account

**group by** branch\_name

**having avg**(balance) >= **all** (**select** **avg**(balance)

**from** account

**group by**  branch\_name)

or

**create** **view** avg\_bal **as**

**select** branch\_name, **avg**(balance) avg\_bal

**from** account

**group by**  branch\_name

**create** **view** max\_bal **as**

**select** **max**(avg\_bal) max\_bal

**from** avg\_bal

**select** branch\_name

**from** avg\_bal, max\_bal

**where** avg\_bal = max\_bal

**4.8 Views**

**Why?**

1. It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users. To see the customer, loan number and branch name but not the loan amount:

**select** branch\_name,customer\_name, b.loan\_no

**from** borrower b, loan l

**where** b.loan\_no = l.loan\_no )

2. It creates a personalized collection of relations that is better matched to a certain user’s intuition than the logical model

* Any relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views of the top of any given set of actual relations.

**4.8.1 View Definition**

We can define view in SQL using **create view** command. The form of the create view command is: **create view** v **as** <query expression>, where <query expression> is any legal query expression and the view name is represented by v.

a) Create a view consisting of branch names and the names of customers who have either an account or loan at the bank.

**create view** all\_customers **as**

(**select** branch\_name,customer\_name

**from** depositor d, account a

**where** d.account\_no = a.account\_no )

**union**

(**select** branch\_name,customer\_name

**from** borrower b, loan l

**where** b.loan\_no = l.loan\_no )

* The attribute names of a view can be specified explicitly as follows:

**create view** branch\_total\_loan(branch\_name, total\_loan) **as**

(**select** branch\_name, **sum**(amount)

**from** loan

**group by** branch\_name)

c) Using view *all\_customers*, find all customers of Perryridge branch.

**select** customer\_name

**from** all\_customers

**where** branch\_name = ‘Perryridge’

**4.9 Modification of the Database**

* Modification means how to add, remove or change information with SQL.

**4.9.1 Deletion**

* A delete request is expressed in much the same way as a query. Instead of displaying, the selected tuples are removed from the database. We can only delete whole tuples; we cannot delete values on only particular attributes.
* A deletion in SQL is of the form

**delete**

**from** r

**where** P

where P represents a predicate and r represents a relation. The delete statement first finds all tuples t in r for which P(t) is true and then deletes them form r. If the **where** clause is omitted, all tuples are deleted.

* A **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.

a) Delete all of Smith's account records

**delete from** depositor

**where** customer\_name=’Smith’

b) Delete all account tuples in the Perryridge branch

**delete from** account

**where** branch\_name=’Perryridge’

c) Delete all loans with loan amounts between 1300 and 1500.

**delete from** loan

**where** amount **between** 1300 **and** 1500

d) Delete all accounts at branches located in Brooklyn

**delete from** account

**where** branch\_name **in**

(**select** branch\_name

**from** branch

**where** branch\_city = ’Brooklyn’)

e) Delete the records of all accounts with balances below the average

**delete from** account

**where** balance < (**select avg**(balance)

**from** account)

**4.9.2 Insertion**

* To insert data into a relation, we either specify a tuple, or write a query whose result is the set of tuples to be inserted. Attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity.

a) Insert a tuple for a customer who has Tk 1200 in account A-9372 at the Perryridge branch

**insert into** account

**values** (‘A-9372’, ‘Perryridge’, 1200)

* The above insertion is identical with:

**insert into** account (account\_no, branch\_name, balance)

**values** (‘A-9372’, ‘Perryridge’, 1200)

* We may be interested to insert tuples on the basis of the result of the query

b) To provide each loan that the customer has in the Perryridge branch with a Tk 200 savings account.

**insert** **into** account

**select** branch\_name, loan\_no, 200

**from** loan

**where** branch\_name = ‘Perryridge’

* It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by null. We can prohibit the insertion of null values using the SQL DDL.

**insert into** account

**values** (‘A-9372’, null, 1200)

**4.9.3 Updates**

* Updating allows us to change some values in a tuple without necessarily changing all. Like insert and delete, we can choose the tuples to be updated using a query.

a) Increase all balances by 5 percent.

**update** account

**set** balance = balance \* 1.05

b) Give 6 percent interest for all accounts with balance over 700 and 5 percent for the rest.

**update** account

**set** balance = balance \* 1.06

**where** balance > 700

**update** account

**set** balance = balance \* 1.05

**where** balance <= 700

c) Pay 5% interest on account whose balance is greater than average

**update** account

**set** balance = balance \* 1.05

**where** balance > (**select** **avg** (balance)

**from** account)

**4.10 Joined Relation**

* SQL provides not only the basic Cartesian-product mechanism for joining tuples of relations, but, also provides various other mechanisms for joining relations.

**4.10.1 Join Types and Conditions**

* Join operations take two relations as input and return another relation as a result. Each of the variant of join operations in SQL consists of a **join type** and **join condition**.
* **Join types: inner join, left outer join, right outer join, full outer join**.
* **Join conditions: natural, on** <predicate>, **using** (A1,A2,…,An).

**4.10.2 Examples**

1. **select** \*

**from** loan **natural inner join** borrower

2. **select** \*

**from** loan **natural** **left outer join** borrower

3. **select** \*

**from** loan **natural** **right outer join** borrower

4. **select** \*

**from** loan **natural** **full outer join** borrower