**Relational Model**

* The relational model is today the primary data model for commercial data-processing applications. It has attained its primary position because of its simplicity, which eases the job of the programmer, as compared to earlier data models such as the network model or the hierarchical model.
* The relational algebra forms the **basis** of the widely used **SQL query language.** Overviews of the other two formal languages are provided, **the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic**. The domain relational calculus is the basis of the (Query By Example) QBE query language.

1. **Structure of Relational Databases**

* A relational database consists of a collection of tables, each of which is assigned a

unique name. Each table has a structure similar to E-R databases by tables.

* A row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name.
* In what follows, introduction of the concept of relation are as follows:

1. Basic Structure
2. Database Schema
3. Keys
4. Schema Diagram
5. Query Languages

* In a **procedural language,** the user instructs the system to perform a sequence of operations on the database to compute the desired result.
* In a **nonprocedural language**, the user describes the desired information without giving a specific procedure for obtaining that information.

**2) The Relational Algebra**

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename. In addition to the fundamental operations, there are several other operations — namely, set intersection, natural join, division, and assignment. We will define these operations in terms of the fundamental operations.

1. **The Select Operation**

σbranch-name = “Perryridge” (loan)

σcustomer-name = banker-name (loan-officer)

1. **The Project Operation**

Πloan-number, amount(loan)

1. **Composition of Relational Operations**

Πcustomer-name (σcustomer-city = “Harrison” (customer))

1. **The Union Operation**

Πcustomer-name (borrower)

Πcustomer-name (depositor)

Πcustomer-name (borrower) ∪ Πcustomer-name (depositor)

1. **The Set Difference Operation**

Πcustomer-name (depositor) − Πcustomer-name (borrower)

1. **The Cartesian-Product Operation**

(borrower.customer-name, borrower.loan-number, loan.loan-number,

loan.branch-name, loan.amount)

(customer-name, borrower.loan-number, loan.loan-number,

branch-name, amount)

σbranch-name = “Perryridge”(borrower × loan)

1. **The Rename Operation**

Rho

1. **The Set-Intersection Operation**

Πcustomer-name (borrower) ∩ Πcustomer-name (depositor)

1. **The Natural-Join Operation**

Πcustomer-name, loan-number, amount (borrower **join** loan)

1. **The Division Operation**
2. **The Assignment Operation**

**3) Extended Relational-Algebra Operations**

1. Generalized Projection
2. Aggregate Functions
3. Outer Join
4. Null Values∗∗

**4) Modification of the Database**

1. Deletion
2. Insertion
3. Updating
4. Views

**SQL**

* SQL uses a combination of relational-algebra and relational-calculus constructs.
* Developed by IBM.
* The SQL language has several parts:

• **Data-definition language (DDL)** -- The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

• **Interactive data-manipulation language (DML)** -- The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It also includes commands to insert tuples into, delete tuples from and modify tuples in the database.

• **View definition** -- The SQL DDL includes commands for defining views.

• **Transaction control** -- SQL includes commands for specifying the beginning and ending of transactions.

• **Embedded SQL and dynamic SQL** -- Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.

• **Integrity** -- The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.

• **Authorization** -- The SQL DDL includes commands for specifying access rights to relations and views.

The following is a **banking enterprise** with the following **relation schemas**:

**Branch-schema = (branch-name, branch-city, assets)**

**Customer-schema = (customer-name, customer-street, customer-city)**

**Loan-schema = (loan-number, branch-name, amount)**

**Borrower-schema = (customer-name, loan-number)**

**Account-schema = (account-number, branch-name, balance)**

**Depositor-schema = (customer-name, account-number)**

1. **BASIC STRUCTURE:**

The basic structure of an SQL expression consists of three clauses: select, from, and

where.

• 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.

• The **from clause** corresponds to the **Cartesian-product operation** of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.

• The **where clause** corresponds to the **selection predicate** of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the from clause.

**SYNTAX**: select A1, A2,...,An

from r1, r2,...,rm

where P

Each Ai represents an attribute, and each ri a relation. P is a predicate. The query is equivalent to the relational-algebra expression

ΠA1, A2,...,An (σP (r1 × r2 × ··· × rm))

1. **The select Clause Eg.**

“Find the names of all branches in the loan relation”:

**select** branch-name

**from** loan

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

**select all** branch-name

**from** loan

\*To remove duplicates we can use the keyword “**distinct**”.

The select clause may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query

**select** loan-number, branch-name, amount \* 100

**from** loan

1. **The where Clause Eg.**

“Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than Rs. 1200.”

**select** loan-number

**from** loan

**where** branch-name = ’Perryridge’ and amount > 1200

* SQL uses the logical connectives and, or, and not—rather than the mathematical symbols ∧, ∨, and ¬ —in the where clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
* SQL includes a **between** comparison operator to simplify where clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.

**select** loan-number

**from** loan

**where** amount **between** 90000 and 100000

1. **The from Clause Eg.**

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

**select** customer-name, borrower.loan-number, amount

**from** borrower, loan

**where** borrower.loan-number = loan.loan-number

We can extend the preceding query and consider a more complicated case in which we require also that the loan be from the Perryridge branch: “Find the customer names, loan numbers, and loan amounts for all loans 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’

1. **The Rename Operation Eg.**

* SQL provides a mechanism for renaming both relations and attributes. It uses the as clause, taking the form:

**old-name as new-name**

* Let us consider the following query again:

**select** customer-name, borrower.loan-number, amount

**from** borrower, loan

**where** borrower.loan-number = loan.loan-number

* For example, if we want the attribute name loan-number to be replaced with the name loan-id, we can rewrite the preceding query as

**select** customer-name, borrower.loan-number **as** loan-id, amount

**from** borrower, loan

**where** borrower.loan-number = loan.loan-number

1. **Tuple Variables Eg.**

* The as clause is particularly useful in defining the notion of tuple variables, as is done in the tuple relational calculus. A tuple variable in SQL must be associated with a particular relation. As given before

“For all customers who have a loan from the bank, find their names, loan numbers, and loan amount” as

**select** customer-name, T.loan-number, S.amount

**from** borrower as T, loan **as** S

**where** T.loan-number = S.loan-number

* Tuple variables are most useful for comparing two tuples in the same relation.
* “Find the names of all branches that have assets greater than at least one 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’

1. **String Operations Eg.**

* SQL expresses patterns by using the **like** comparison operator.

“Find the names of all customers whose street address includes the substring ‘Main’.”

**select** customer-name

**from** customer

**where** customer-street **like**’%Main%’

1. **Ordering the Display of Tuples Eg.**

**Eg.**

**select** distinct customer-name

**from** borrower, loan

**where** borrower.loan-number = loan.loan-number and

branch-name = ’Perryridge’

**order by** customer-name

**E.g.**

select \*

from loan

order by amount desc, loan-number asc

1. **Duplicates E.g.**

* We can define the duplicate semantics of an SQL query using **multiset** versions of the relational operators.
* For example, suppose that relations r1 with schema (A, B) and r2 with schema (C) are the following multisets:

r1 = {(1, a), (2, a)} r2 = {(2), (3), (3)}

* Then ΠB(r1) would be {(a),(a)}, whereas ΠB(r1) × r2 would be

{(a, 2),(a, 2),(a, 3),(a, 3),(a, 3),(a, 3)}

select A1, A2,...,An

from r1, r2,...,rm

where P

1. **Set Operations**
2. **The Union Operation Eg.**

* To find all customers having a loan, an account, or both at the bank, we write

(**select** customer-name

**from** depositor)

**union**

(**select** customer-name

**from** borrower)

* If we want to retain all duplicates, we must write **union all** in place of union:

(**select** customer-name

**from** depositor)

**union all**

**(select** customer-name

**from** borrower)

1. **The Intersect Operation Eg.**

* To find all customers who have both a loan and an account at the bank, we write

(**select distinct** customer-name

**from** depositor)

**intersect**

(**select** distinct customer-name

**from** borrower)

* If we want to retain all duplicates, we must write **intersect all** in place of intersect:

(**select** customer-name

**from** depositor)

**intersect all**

(**select** customer-name

**from** borrower)

1. **The Except Operation Eg**

* To find all customers who have an account but no loan at the bank, we write

(**select** distinct customer-name

**from** depositor)

**except**

(**select** customer-name

**from** borrower)

* If we want to retain all duplicates, we must write **except all** in place of except:

(**select** customer-name

**from** depositor)

**except all**

(**select** customer-name

**from** borrower)

1. **Aggregate Functions**
2. **AVERAGE**

“Find the average account balance at the Perryridge branch.”

**select avg** (balance)

**from** account

**where** branch-name = ’Perryridge’

1. **GROUP BY**

“Find the average account balance at each branch.”

**select** branch-name, **avg** (balance)

**from** account

**group by** branch-name

1. **DISTINCT**

If we do want to eliminate duplicates, we use the keyword distinct in the aggregate expression.

“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

1. **GROUP BY AND HAVING CLAUSE**

* For example, we might be interested in only those branches where the average account balance is more than Rs.1200.
* This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause.
* To express such a query, we use the **having clause** of SQL. SQL applies predicates in the having clause after groups have been formed, so aggregate functions may be used.

**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, we **do not use a group by clause**.
* “Find the average balance for all accounts.”

select avg (balance)

from account

* We use the aggregate function **count**, frequently to count the number of tuples in a relation. The notation for this function in SQL is **count (\*)**.
* “Find the number of tuples in the customer relation”

select count (\*)

from customer

* SQL does not allow the use of distinct with count(\*).
* To illustrate the use of **both a having clause** and a **where clause** in the same query,

we consider the query.

* “Find the average balance for each customer who lives in Harrison and has at least three accounts.”

**select** depositor.customer-name, **avg** (balance)

**from** depositor, account, customer

**where** depositor.account-number = account.account-number and

depositor.customer-name = customer.customer-name and

customer-city = ’Harrison’

**group by** depositor.customer-name

**having count** (distinct depositor.account-number) >= 3

1. **Nested Subqueries**

* A subquery is a select-from where expression that is nested within another query.

1. **Set Membership**

* Finding all account holders,

(select customer-name

from depositor)

* We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the subquery.

select distinct customer-name

from borrower

where customer-name in (select customer-name

from depositor)

* “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)

* “Find all customers who do 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)

* Another way to determine is:

select distinct customer-name

from borrower

where customer-name not in (’Smith’, ’Jones’)

1. **Set Comparison**

* “Find the names of all branches that have assets greater than those of at least one

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’

OR

select branch-name

from branch

where assets > all (select assets

from branch

where branch-city = ’Brooklyn’)

1. **Test for Empty Relations**

* The **exists** construct returns the value true if the argument subquery is nonempty.
* “Find all customers who have both an account and a loan at the bank”

select customer-name

from borrower

where exists (select \*

from depositor

where depositor.customer-name = borrower.customer-name)

* Using the except construct, we can write the query as follows:

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))

1. **Test for the Absence of Duplicate Tuples**

* The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples.
* “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’)

1. **Views**

The form of the create view command is

create view v as <query expression>

* Consider the view consisting of branch names and the names of customers who have either an account or a loan at that branch. Assume that we want this view to be called all-customer.

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)

* 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

groupby branch-name

**25th JANUARY 2021**

1. **COMPLEX QUERIES**
2. **Derived Relations**

* SQL allows a subquery expression to be used in the **from** clause.
* We must give the **result relation** a name, and we can **rename** the attributes.
* **For example**, consider the subquery

(select branch-name, **avg** (balance)

**from** account

**group by** branch-name)

**as** result (branch-name, avg-balance)

* This subquery generates a relation **consisting of the names of all branches and their corresponding avera*ge account balances.*** *The subquery result is named result, with the attributes* branch-name and *avg-balance.*
* “Find the average account balance of those branches where the average account balance is greater than Rs1200.”

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

* Example, suppose we wish to find the maximum across all branches of the total balance at each branch.

**select max**(tot-balance)

**from** (select branch-name, sum(balance)

**from** account

**group by** branch-name) **as** branch-total (branch-name, tot-balance)

1. **The 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.
* Consider the following query, which **selects accounts with the maximum balance**; if there are many accountswith the same maximum balance, all of them are selected.

**with** max-balance (value) as

**select max**(balance)

**from** account

**select** account-number

**from** account, max-balance

**where** account.balance = max-balance.value

* The **with** clause makes the query logic clearer; it also permits a view definition to be used in multiple places within a query.
* “For example, suppose we want to find all branches where the total account deposit is less 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

1. **Modification of the Database**
2. **Deletion**

* We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

delete from r

where P

* The delete statement first **finds all tuples t in r** for which P(t) is true, and **then deletes them from r**. The where clause can be omitted, in which case all tuples in r are deleted.
* A delete command operates on only one relation.
* The request

**delete from** loan

deletes all tuples from the *loan* relation.

* Delete all account tuples in the Perryridge branch.

**delete from** account

**where** branch-name = ’Perryridge’

* Delete all account tuples at every branch located in Needham.

**delete from** account

**where** branch-name **in** (select branch-name

**from** branch

**where** branch-city = ’Needham’)

This delete request first finds all branches in Needham, and then deletes all account tuples pertaining to those branches.

* The delete request can contain a nested select that references the relation from which tuples are to be deleted.
* For example: suppose that we want to delete the records of all accounts with balances below the average at the bank.

**delete from** account

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

**from** account)

The delete statement first tests each tuple in the relation account to check whether the account has a balance less than the average at the bank. Then, all tuples that fail the test—that is, represent an account with a lower-than-average balance—are deleted.

1. **Insertion**

* To insert data into a relation, **we either specify a tuple to be inserted** or **write a query whose result is a set of tuples to be inserted**.
* Suppose that we wish to insert the fact that there is an account A-9732 at the Perryridge branch and that is has a balance of $1200.

insert into account

values (’A-9732’, ’Perryridge’, 1200)

* Suppose that we want to present a new Rs.200 savings account as a gift to all loan customers of the Perryridge branch, for each loan they have.
* Let the loan number serve as the account number for the savings account.

insert into account

select loan-number, branch-name, 200

from loan

where branch-name = ’Perryridge’

* It is **important that we evaluate the select statement fully before** we carry out **any insertions**. If we carry out some insertions even as the select statement is being evaluated, a request such as

insert into account

select \*

from account

might insert an infinite number of tuples.