Unit II **Relational database model**

Contents:

- Logical view of data
- Keys
- Integrity rules
- Relational Database design: features of good relational database design
- Normalization (1NF, 2NF, 3NF, BCNF).
- Relational algebra: introduction
- Selection and projection
- Set operations, renaming
- Joins, Division, syntax, semantics.
- Operators, grouping and ungrouping
- Relational comparison.
- Tuple relational calculus
- Domain relational Calculus
- Calculus vs algebra
- Computational capabilities

- **Recommended Books:**
  1. A Silberschatz, H Korth, S Sudarshan, "Database System and Concepts", fifth Edition McGraw- Hill
  2. Database Systems ,Rob Coronel,Cengage Learning.
  3. Programming with PL/SQL for Beginners, H. Dand, R. Patil and T. Sambare,X-Team.
  4. Introduction to Database System,C.J.Date,Pearson

- **Prerequisites and Linking:**

Unit I	Pre-Requisites	Sem. II	Sem. III	Sem. IV	Sem. V	Sem. VI
Introduction To Databases & Transactions	--	WP, OOP	DBMS	CJ	EJ,AWP	Project

Notes:

What is the logical view?
Logical view: The logical view is concerned with the functionality that the system provides to end-users. UML diagrams used to represent the logical view include, class diagrams, and state diagrams. UML diagrams used to represent the physical view include the deployment diagram.

Database Keys
Keys are very important part of Relational database. They are used to establish and identify relation between tables. They also ensure that each record within a table can be uniquely identified by combination of one or more fields within a table.

- **Super Key**

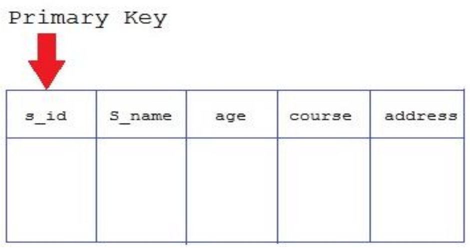
Super Key is defined as a set of attributes within a table that uniquely identifies each record within a table. Super Key is a superset of Candidate key.

- **Candidate Key**

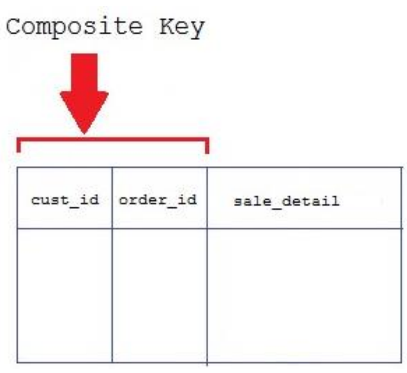
Candidate keys are defined as the set of fields from which primary key can be selected. It is an attribute or set of attribute that can act as a primary key for a table to uniquely identify each record in that table.

- **Primary Key**

Primary key is a candidate key that is most appropriate to become main key of the table. It is a key that uniquely identify each record in a table.



- **Composite Key**
  Key that consist of two or more attributes that uniquely identify an entity occurance is called Composite key. But any attribute that makes up the Composite key is not a simple key in its own.



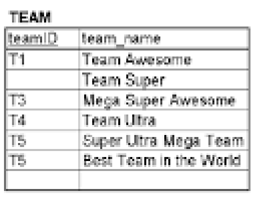
Integrity Rules

- Integrity rules may sound very technical but they are simple and straightforward rules that each table must follow.
- These are very important in database design, when tables break any of the integrity rules our database will contain errors when retrieving information.
- Hence the name "integrity" which describes reliability and consistency of values.
- there are two types of integrity rules that we will look at:
  - Entity Integrity Rule
  - Referential Integrity Rule

Entity Integrity Rule:

- The entity integrity rule refers to rules the primary key must follow.
  - The primary key value cannot be null.
  - The primary key value must be unique.
- If a table does not meet these two requirements, we say the table is violating the entity integrity rule.

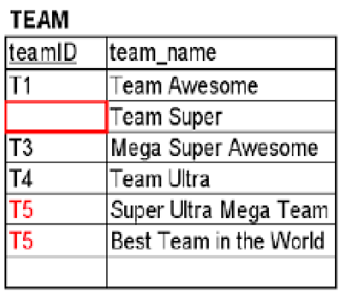
- For example, does this table violate entity integrity rule? Where?



TEAM

teamID	team_name
T1	Team Awesome
	Team Super
T3	Mega Super Awesome
T4	Team Ultra
T5	Super Ultra Mega Team
T5	Best Team in the World

- The team table violates the entity integrity rules at two places.
  - Team Super- missing primary key
- Super Ultra Mega Team and Best Team in the World -has the same primary key.

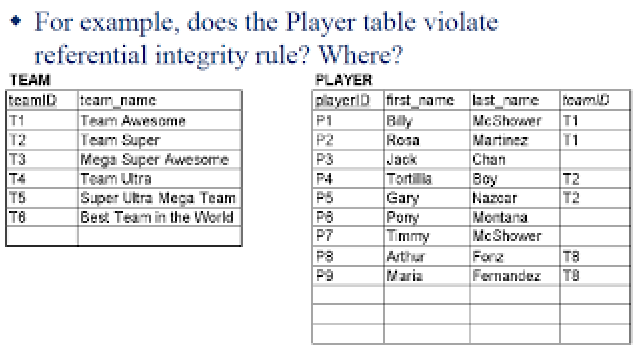


TEAM

teamID	team_name
T1	Team Awesome
	Team Super
T3	Mega Super Awesome
T4	Team Ultra
T5	Super Ultra Mega Team
T5	Best Team in the World

Referential Integrity Rule:

- The referential integrity rule refers to the foreign key.
- The foreign key may be null and may have the same value but:
  - The foreign key value must match a record in the table it is referring to.
- Tables that do not follow this are violating the referential integrity rule.



- For example, does the Player table violate referential integrity rule? Where?

TEAM

teamID	team_name
T1	Team Awesome
T2	Team Super
T3	Mega Super Awesome
T4	Team Ultra
T5	Super Ultra Mega Team
T6	Best Team in the World

PLAYER

playerID	first_name	last_name	teamID
P1	Billy	McShower	T1
P2	Rosa	Martinez	T1
P3	Jack	Chan	
P4	Tortillia	Boy	T2
P5	Gary	Nazcar	T2
P6	Pony	Montana	
P7	Timmy	McShower	
P8	Arthur	Fonz	T9
P9	Maria	Fernandez	T9

Relational Database design

- ➢ The relational database model was conceived by E.F. Codd in 1969,then a researcher at IBM.
- ➢ The basic idea behind the relational model is that a database consists of a sequence of relations (or tables) that can be manipulated using non-procedural operations that return tables.
- ➢ A relational DBMS must use its relational facilities exclusively to manage and interact with the database.
- ➢ When designing a database ,we need to decide which tables to create ,what columns they will contain ,as well as the relationships between the tables.

Goals

- ➢ Design should ensure that all database operations will be efficiently performed and DBMS should not perform expensive consistency checks.
- ➢ No data redundancy should be their.

Features of good relational database design

- The primary feature of a relational database is its primary key, which is a unique identifier assigned to every record in a table. An example of a good primary key is a registration number. It makes every record unique, facilitating the storage of data in multiple tables, and every table in a relational database must have a primary key field.
- Another key feature of relational databases is their ability to hold data over multiple tables. This feature overcomes the limitations of simple flat file databases that can only have one table. The database records stored in a table are linked to records in other tables by the primary key. The primary key can join the table in a one-to-one relationship, one-to-many relationship or many-to-many relationship.
- Relational databases enable users to delete, update, read and create data entries in the database tables. This is accomplished though structured query language, or SQL, which is based on relational algebraic principles. SQL also enable users to manipulate and query data in a relational database.
- Relational tables follow various integrity rules that ensure the data stored in them is always accessible and accurate. The rules coupled with SQL enable users to easily enforce transaction and concurrency controls, thus guaranteeing data integrity. The relational database concept was established by Edgar F. Codd in 1970.

Normalization (1NF, 2NF, 3NF, BCNF).

Database Normalisation is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anamolies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Normalization is used for mainly two purpose,

- Eliminating reduntant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

https://www.youtube.com/watch?v=IwswNIZ_PSg

Problem Without Normalization

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized. To understand these anomalies let us take an example of **Student** table.

S_id	S_Name	S_Address	Subject_opted
401	Adam	Noida	Bio
402	Alex	Panipat	Maths
403	Stuart	Jammu	Maths
404	Adam	Noida	Physics

- Updation Anamoly : To update address of a student who occurs twice or more than twice in a table, we will have to update S_Address column in all the rows, else data will become inconsistent.
- Insertion Anamoly : Suppose for a new admission, we have a Student id(S_id), name and address of a student but if student has not opted for any subjects yet then we have to insert NULL there, leading to Insertion Anamoly.
- Deletion Anamoly : If (S_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

Normalization Rule

Normalization rule are divided into following normal form.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF

First Normal Form (1NF)

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

Student Table :

Student	Age	Subject
Adam	15	Biology, Maths
Alex	14	Maths
Stuart	17	Maths

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

Student Table following 1NF will be :

Student	Age	Subject
Adam	15	Biology
Adam	15	Maths
Alex	14	Maths
Stuart	17	Maths

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails Second normal form.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {Student, Subject}, Age of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

New Student Table following 2NF will be :

Student	Age
Adam	15
Alex	14
Stuart	17

In Student Table the candidate key will be Student column, because all other column i.e Age is dependent on it.

New Subject Table introduced for 2NF will be :

Student	Subject
Adam	Biology
Adam	Maths
Alex	Maths
Stuart	Maths

In Subject Table the candidate key will be {Student, Subject} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

Third Normal Form (3NF)

Third Normal form applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this transitive functional dependency should be removed from the table and also the table must be in Second Normal form. For example, consider a table with following fields.

Student_Detail Table :

Student_id	Student_name	DOB	Street	city	State	Zip

In this table Student_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called transitive dependency. Hence to apply 3NF, we need to move the street, city and state to new table, with Zip as primary key.

New Student_Detail Table :

Student_id	Student_name	DOB	Zip

Address Table :

Zip	Street	city	state

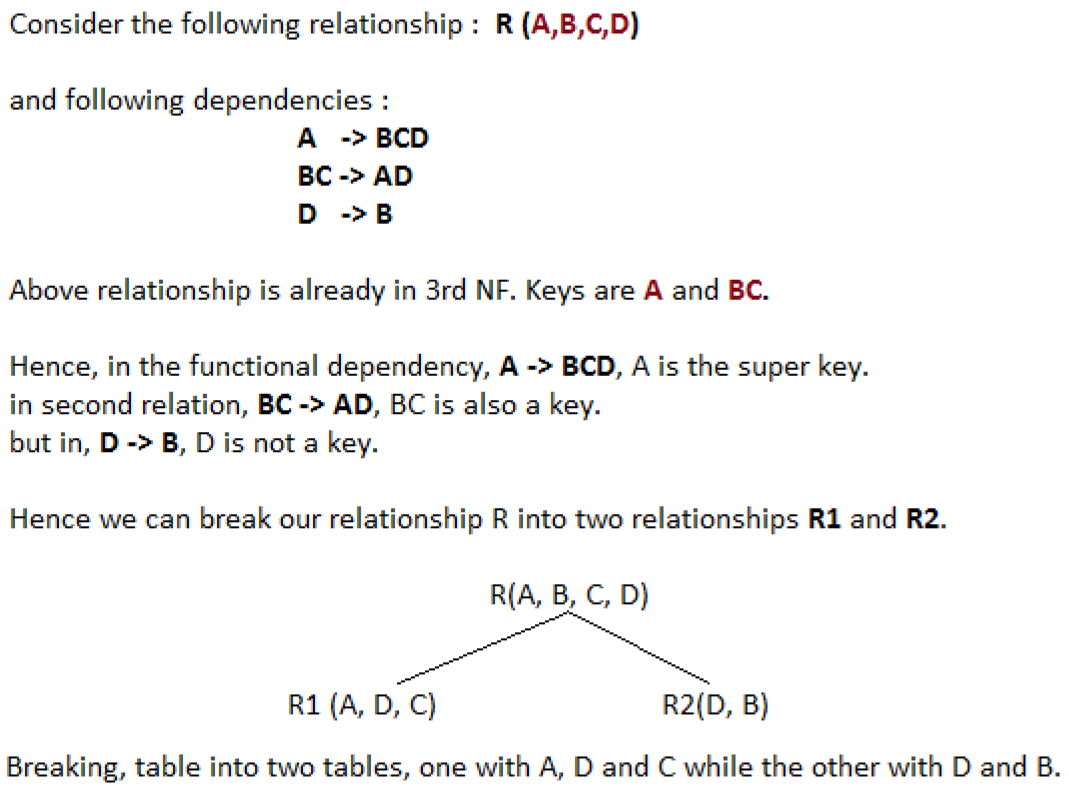
The advantage of removing transtive dependency is,

- Amount of data duplication is reduced.
- Data integrity achieved.

Boyce and Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anamoly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

- R must be in 3rd Normal Form
- and, for each functional dependency (X -> Y), X should be a super Key.


Consider the following relationship : **R (A,B,C,D)**

and following dependencies :

A -> BCD
BC -> AD
D -> B

Above relationship is already in 3rd NF. Keys are **A** and **BC.**

Hence, in the functional dependency, **A -> BCD**, A is the super key.
in second relation, **BC -> AD**, BC is also a key.
but in, **D -> B**, D is not a key.

Hence we can break our relationship R into two relationships **R1** and **R2**.

$$\begin{tikzcd}
 & R(A, B, C, D) \arrow[dl, dash] \arrow[dr, dash] & \\
R1\ (A, D, C) & & R2(D, B)
\end{tikzcd}$$

Breaking, table into two tables, one with A, D and C while the other with D and B.

THE RELATIONAL ALGEBRA

INTRODUCTION

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.

https://www.youtube.com/watch?v=rcVXazo7TD

The Tuple Relational Calculus

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, by contrast, is a monprocedural query language. It describes the desired information without giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as

$$\{ t \mid P(t) \}$$

That is, it is the set of all tuples t such that predicate P is true or t. following our earlier notation, we use t[A] to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational–algebra.

SELECTION AND PROJECTION

The Select Operation

The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the loan relation where the branch is "Perryridge", we write

$$\sigma_{branch\text{`}name="Perryridge"}(loan)$$

The relation that results from the preceding query is as shown in figure 1.

We can find all tuples in which the amount lent is more than $1200 by writing

$$\sigma_{amount>1200}(loan)$$

In general, we allow comparisons using =, ≠, <, ≤, >, ≥ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives and (∧), or (∨), and not (–). Thus, to find those tuples pertaining to loans of more than $1200 made by the Perry ridge branch, we write

$$\sigma_{branch\text{-}name="Perryridge" \wedge amount > 1200}(loan)$$

Loan–number	**Branch–name**	**Amount**
L–15	Perry–ridge	1500
L–16	Perry–ridge	1300

Fig. 1 : Result of $\sigma_{branch\text{–}name = \text{“Perryridge”}}$ (loan).

The selection predicate may include comparisons between two attributes. To illustrate, consider the relation loan-officer that consists of three attributes: customer-name, banker-name, and loan-number, which specifies that a particular banker is the loan officer for a loan that belongs to some customer. To find all customers who have the same name as their loan officer, we can write

$$\sigma_{customer-bane=banker-name}(loan-officer)$$

https://www.youtube.com/watch?v=zmapn-7kyFw

The Project Operation

Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name. The project operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter Pi ($\prod$). We list those attributes that we wish to appear in the result as a subscript to $\prod$. The argument relation follows in parentheses. Thus, we write the query to list all loan numbers and the amount of the loan as

$$\prod_{loan - number,\ amount} (loan)$$

Figure 2 shows the relation that results from this query.

Loan–number	**Amount**

L–11	900
L–14	1500
L–15	1500
L–16	1300
L–17	1000
L–23	2000
L–93	500

Fig. 2 : Loan number and the amount of the loan..

https://www.youtube.com/watch?v=wIrNqYG1HjI

SET OPERATION AND RENAME

Set Operations

The SQL operations union, intersect, and except operate on relations and correspond to the relational–algebra operations $\cup$, $\cap$, and $-$. Like union, intersection, and set difference in relational algebra, the relations participating in the operations must be compatible; that is, they must have the same set of attributes.

Let us demonstrate how several of the example queries that we considered in Chapter 3 can be written in SQL. We shall now construct queries involving the union, intersect, and except operations of two sets : the set of all customers who have an account at the bank, which can be derived by

```
select customer–name
from depositor
```

and the set of customers who have a loan at the bank, which can be derived by

```
select customer–name
from borrower
```

We shall refer to the relations obtained as the result of the preceding queries as d and b, respectively.

1) The Union Operation

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)
```

The union operation automatically eliminates duplicates, unlike the select clause. Thus, in the preceding query, if a customer—say, Jones—has several accounts or loans (or both) at the bank, then Jones will appear only once in the result.

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)
```

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be five tuples with the name Jones in the result.

2) The intersect Operation

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)
```

The intersect operation automatically eliminates duplicates. Thus, in the preceding query, if a customer-say, Jones-has several accounts and loans at the bank, then Jones will appear only once in the result.

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)
```

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be two tuples with the name Jones in the result.

3) The Except Operation

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)
```

The except operation automatically eliminates duplicates. Thus, in the preceding query, a tuple with customer name Jones will appear (exactly once) in the result only if Jones has an account at the bank, but has no loan at the bank.

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)
```

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies of the tuple in d minus the number of duplicate copies of the tuple in b, provided that the difference is positive. Thus, if Jones has three accounts and one loan at the bank, then there will be two tuples with the name

Jones in the result. If, instead, this customer has two accounts and three loans at the bank, there will be no tuple with the name Jones in the result.

The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the rename operator, denoted by the lowercase Greek letter rho (p), lets us do this, Given a relational–algebra expression E the expression

$$p_x (E)$$

returns the result of expression E under the name x.

customer–name
Adams
Hayes

Fig. 3 : Result of $\prod_{\text{customer–name}}$ $(\sigma_{\text{borrower.lonad–number=long.loan–number}}(\sigma_{\text{branch–name}}$ = "Perryridge" (borrower×loan))).

A relation r by itself is considered a (trivial) relational–algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name.

A second form of the rename operation is as follows. Assume that a relational-algebra expression E has arity n. Then, the expression

$$p_x(A_1, A_2, \ldots..A_n) (E)$$

returns the result of expression E under the name x, and with the attributes renamed to A_1, A_2, ….A_n.

To illustrate renaming a relation, we consider the query "Find the largest account balance in the bank." Our strategy is to (1) compute first a temporary relation consisting of those balances that are not the largest and (2) take the set difference between the relation $\prod_{\text{balance}}$ (account) and the temporary relation just computed, to obtain the result.

Step 1 : To compute the temporary relation, we need to compare the values of all account balances. We do this comparison by computing the Cartesian product account × account and forming a selection to compare the value of any two balances appearing in one tuple. First, we need to devise a mechanism to distinguish between the two balance attributes. We shall use the rename operation to rename one reference to the account relation; thus we can reference the relation twice without ambiguity.

Balance
500
400
700
750
350

Fig. 4 : Result of the sub-expression
$\Pi_{account.balance} (\sigma_{account.balance < d. balance} (account \times \rho d (account)))$.

Balance
900

Fig. 5 : Largest account balance in the bank.

We can now write the temporary relation that consists of the balances that are not the largest :

$$\Pi_{account.balance} (\sigma_{account.balance < d. balance} (account \times \rho d(account)))$$

This expression gives those balances in the account relation for which a larger balance appears somewhere in the account relation (renamed as d). The result contains all balances except the larges one. Figure 4 shows this relation.

Step 2 : The query to find the largest account balance in the bank can be written as :

$$\Pi_{balance} (account) - \Pi_{account.balance} (\sigma_{account.balance < d.balance} (account \times \rho d (account)))$$

Figure 5 shows the result of this query.

As one more example of the rename operation, consider the query “Find the names of all customers who live on the same street and in the same city as Smith.” We can obtain Smith’s street and city by writing

$$\Pi_{customer-street, customer-city} (\sigma_{customer-name = \text{“Smith”}} (customer))$$

However, in order to find other customers with this street and city, we must reference the customer relation a second time. In the following query, we use the rename operation on the preceding expression to give its result the name smith-addr, and to rename its attributes to street and city, instead of customer, street and customer-city:

$$\Pi_{customer.customer-name}$$

$(\sigma_{customer.customer\text{-}street\ =\ Smith\text{-}addr.street\ \wedge\ customer.\ customer\text{-}city\ =\ smith\text{-}addr.city}$ (customer $\times$ $\rho_{smith\text{-}addr\ (street,city)}$ $(\prod_{customer-street,\ customer-city}$ $(\sigma_{customer\text{-}name\ =\ \text{“Smith”}}$(customer))))

The result of this query, when we apply it to the customer relation, appears in figure 6.

The rename operation is not strictly required, since it is possible to use a positional notation for attributes. We can name attributes of a relation implicitly by using a positional notation, where \$1, \$2, … refer to the first attribute, the second attribute, and so on. The positional notation also applies to results of relational-algebra operations.

Customer-name
Curry
Smith

Fig. 6 : Customers who live on the same street and in the same city as smith.

The following relational-algebra expression illustrates the use of positional notation with the unary operator σ :

$$\sigma_{\$2\ =\ \$3}(R \times R)$$

If a binary operation needs to distinguish between its two operand relations, a similar positional notation can be used for relation names as well. for example, \$R1 could refer to the first operand, and \$R2 could refer to the second operand. However, the positional notation is inconvenient for humans, since the position of the attribute is a number, rather than an easy-to-remember attribute name. Hence, we do not use the positional notation in this textbook.

The Division Operation

The division operation, denoted by $\div$, is suited to queries that include the phrase “for all”. Suppose that we wish to find all customers who have an account at all the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression

$$r1 = \prod \text{branch-name } (\sigma \text{branch-city} = \text{“Brooklyn” (branch)})$$

The result relation for this expression appears in figure 18.

branch–name
Brighton
Downtown

Fig. 18 : Result of ∏branch-name (σbranch-city = “Brooklyn” (branch)).

We can find all (customer-name, branch–name) pairs for which the customer has an account at a branch by writing

r2 = ∏customer-name, branch-name (depositor ⋈ account)

Figure 19 shows the result relation for this expression.

Now, we need to find customers who appear in r2 with every branch name in r1. The operation that provides exactly those customers is the divide operation. We formulate the query by writing

∏customer-name, branch-name (depositor ⋈ account)
÷ ∏branch-name (σbranch-city = “Brooklyn” (branch))

The result of this expression is a relation that has the schema (customer–name) and that contains the tuple (Johnson).

Formally, let r(R) and s(S) be relations, and let S ⊆ R; that is, every attribute of schema S is also in schema R. The relation r ÷ s is a relation on schema R − S (that is, on the schema containing all attributes of schema R that are not is schema S). A tuple t is in r ÷ s if and only if both of two conditions hold :

1. t is in ∏R–S(r)
2. For every tuple ts in s, there is a tuple tr in r satisfying both of the following :

(a) ts [S] = ts[S] (b) tr [R–S] = t

It may surprise you to discover that, given a division operation and the schemas of the relations, we can, in fact, define the division operation in terms of the fundamental operations. Let r(R) and s(S) be given, with S ⊆ R :

r ÷ s = ∏R–S(r) – ∏R–S ((∏R–S (r) × s) –∏R–S, S(r))

customer-name	branch–name
Heyes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Fig. 19 : Result of ∏customer–name, branch–name (depositor ⋈ account).

To see that this expression is true, we observe that ∏R–S (r) gives us all tuples t that satisfy the first condition of the definition of the definition of division. The expression on the right side of the set difference operator

$$\Pi_{R-S}\ ((\Pi_{R-S}\ (r) \times s) - \Pi_{R-S,\ S}(r))$$

serves to eliminate those tuples that fail to satisfy the second condition of the definition of division. Let us see how it does so. Consider ∏R–S (r) × s. This relation is on schema R, and pairs every tuple in ∏R–S, (r) with every tuple in s. The expression ∏R–S, S(r) merely reorders the attributes of r.

Thus, (∏R–S (r) × s) – ∏R–S, S(r) gives us those pairs of tuples from ∏R–S (r) and s that do not appear in r. If tuple tj is in

$$\Pi_{R-S}\ ((\Pi_{R-S}\ (r) \times s) - \Pi_{R-S,\ S}(r))$$

Then there is some tuple ts in s that does not combine with tuple tj to form a tuple in r. Thus, tj holds a value for attributes R–S that does not appear in r ÷ s. It is these values that we eliminate from ∏R–S (r).

CALCULUS

THE TUPLE RELATIONAL CALCULUS

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, by contrast, is a monprocedural query language. It describes the desired information without giving a specific procedure for obtaining that information.
A query in the tuple relational calculus is expressed as

$$\{\ t \mid P(t)\ \}$$

That is, it is the set of all tuples t such that predicate P is true or t. following our earlier notation, we use t[A] to denote the value of tuple t on attribute A, and we use t ∈ r to denote that tuple t is in relation r.
Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational–algebra.
Example Queries
Say that we want to find the branch-name, loan-number, and amount for loans of over $1200:

$$\{t \mid t \in loan \wedge t[amount] > 1200\}$$

Suppose that we want only the loan-number attribute, rather than all attributes of the loon relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (loan-number). We need those tuples on (loan-number) such that there is a tuple in loan with the amount attribute > 1200. To express this request, we need the construct "there exists" from mathematical logic. The notation

$\exists\ t \in r\ (Q(t))$

means "there exists a tuple t in relation r such that predicate Q{t) is true."

Using this notation, we can write the query "Find the loan number for each loan of an amount greater than $1200" as

$\{t \mid \exists\ s \in \text{loan}\ (t[\text{loan–number}] = s[\text{loan–number}] \wedge s[\text{amount}] > 1200)\}$

In English, we read the preceding expression as "The set of all tuples t such that there exists a tuple s in relation loan for which the values of t and s for the loan-number attribute are equal, and the value of s for the amount attribute is greater than $1200."

Tuple variable t is defined on only the loan-number attribute, since that is the only attribute having a condition specified for t. Thus, the result is a relation on (loan–number).

Consider the query "Find the names of all customers who have a loan from the Perryridge branch". This query is slightly more complex than the previous queries, since it involves two relations: borrower and loan. As we shall see, however, all it requires is that we have two "there exists" clauses in our tuple-relational-calculus expression, connected by and (∧). We write the query as follows:

$$\begin{aligned}\{t \mid \exists\ s \in \text{borrower}\ (&\ t[\text{customer-name}] = s[\text{customer–name}]\\ &\wedge \exists u \in \text{loan}\ (u[\text{loan–number}] = s[\text{loan–number}]\\ &\wedge \exists\ u \in \text{loan}\ (u[\text{branch–name}] = \text{“Perryridge”}))\}\end{aligned}$$

In English, this expression is "The set of all (customer-name) tuples for which the customer has a loan that is at the Perryridge branch". Tuple variable if ensures that the customer is a borrower at the Perryridge branch. Tuple variable s is restricted to pertain to the same loan number as s. Figure 20 shows the result of this query.

To find all customers who have a loan, an account, or both at the bank, we used the union operation in the relational algebra. In the tuple relational calculus, we shall need two "there exists" clauses, connected by or (V):

$\{t \mid \exists\ s \in \text{borrower} (t[\text{customer-name}] = s[\text{customer–name}])$
$\wedge \exists u \in \text{depositor} (t [\text{customer–name}] = u[\text{customer–name}])\}$

This expression gives us the set of all customer-name tuples for which at least one of the following holds:

- The customer name appears in some tuple of the borrower relation as a borrower from the bank.
- The customer-name appears in some tuple of the depositor relation as a depositor of the bank.

If some customer has both a loan and an account at the bank, that customer appears only once in the result, because the mathematical definition of a set does not allow duplicate members.

If we now want only those customers who have both an account and a loan at the bank, all we need to do is to change the or ($\vee$) to and ($\wedge$) in the preceding expression.

$\{t \mid \exists\ s \in \text{borrower} (t[\text{customer-name}] = s[\text{customer–name}])$
$\wedge \exists u \in \text{depositor} (t [\text{customer–name}] = u[\text{customer–name}])\}$

Now consider the query "Find all customers who have an account at the bank but do not have a loan from the bank." The tuple-relational–calculus expression for this query is similar to the expressions that we have just seen, except for the use of the not ($\neg$) symbol:

$\{t \mid \exists\ u \in \text{depositor} (t [\text{customer-name}] = u[\text{customer–name}])$
$\wedge \neg \exists\ s \in \text{borrower} (t [\text{customer–name}] = u[\text{customer–name}])\}$

customer–name
Adams
Hayes

Fig.: Names of all customer who have a loan at the Perryridge branch.

This tuple-relational-calculus expression uses the $\exists\ u \in$ depositor (...) clause to require that the customer have an account at the bank, and it uses the $\neg \exists\ s \in$ borrow r (...) clause to eliminate those customers who appear in some tuple of the borrower relation as having a loan from the bank.

The query that we shall consider next uses implication, denoted, by $\Rightarrow$. The formula $P \Rightarrow Q$ means "P implies Q"; that is, "if P is true, then must be true." Note that $P \Rightarrow Q$ is logically equivalent to $\neg P \vee Q$. The use of implication rather than not and or often suggests, a more intuitive interpretation of a query in English.

"Find all customers who have an account at all branches located in Brooklyn." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by $\forall$. The notation

$$\forall\ t \in r\ (Q(t))$$

means "Q is true for all tuples t in relation r."

We write the expression for our query as follows:

$$\{t \mid \exists\ r \in \text{customer}\ (r\ [\text{customer–name}] = t\ [\text{customer–name}]\ ) \wedge$$
$$(\ \forall\ u \in \text{branch}\ (u\ [\text{branch–city}] = \text{"Brooklyn"} \Rightarrow$$
$$\exists\ s \in \text{depositor}\ (t[\text{customer–name}] = s[\text{customer–name}]$$
$$\wedge\ \exists\ w \in \text{account}\ (w[\text{account–number}] = s[\text{account–number}]$$
$$\wedge\ w\ [\text{branch–name}] = u[\text{branch–name}])))))\}$$

In English, we interpret this expression as "The set of all customers (that is, (customer-name) tuples t) such that, for all tuples u in the branch relation, if the value of u on attribute branch-City is Brooklyn, then the customer has an account at the branch whose name appears in the branch–name attribute of u." Note that there is a subtlety in the above query: If there is no branch in Brooklyn, all customer names satisfy the condition. The first line of the query expression is critical in this case–without the condition.

$$\exists\ r \in \text{customer}\ (r[\text{customer–name}] = t\ [\text{customer–name}])$$

if there is no branch in Brooklyn, any value of t (including values that are not customer names in the depositor relation) would qualify.

Formal Definition

We are now ready for a formal definition. A tuple–relational–calculus expression is of the form

$$\{t \mid P\ (t)\}$$

Where P is a formula. Several tuple variables may appear in a formula. A tuple variable is said to be a free variable unless it is quantified by a $\exists$ or $\forall$. thus, in

$$t \in \text{loan} \wedge \exists\ s \in \text{customer}\ (t[\text{branch–name}] = s\ [\text{branch–name}])$$

t is a free variable. tuple variable s is said to be a bound variable.

A tuple-relational-calculus formula is built up out of atoms. An atom has one of the following forms:

- $s \in r$, where s is a tuple variable and r is a relation (we do not allow use of the $\notin$ operator)
- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison

operator ($<, \leq, =, \neq, >, \geq$); we require that attributes x and y have domains whose members can be compared by Θ.

- s[x] Θ c, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P1 is a formula, then so are ¬P1 and (P1).
- If P1 and P2 are formulae, then so are P1 ∨ P2, P1 ∧ P2, and P1 ⇒ P2.
- If P1(s) is a formula containing a free tuple variable s, and r is a relation, then

  $$\exists\ s \in r\ (P1\ (s)) \text{ and } \forall\ s \in r\ (P1\ (s))$$

  are also formulae.

As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include the following three rules:

i) P1 ∧ P2 is equivalent to ¬(¬(P1)∨ ¬(P2)).

ii) ∀t∈r(P1(t)) is equivalent to ¬ ∃ ∈ r (¬P1(t)).

iii) P1 ⇒ P2 is equivalent to ¬ (P1) ∨ P2.

DOMAIN RELATIONAL CALCULUS

A second form of relational calculus, called domain relational calculus, uses domain variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.

Formal Definition

An expression in the domain relational calculus is of the form

{<x1 x2,……., xn > | P (x1, x2, ….., xn)}

where x1, x2, …., xn represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- < x1 x2,……., xn > ∈ r, where r is a relation on n attributes and x1 x2,…, xn are domain variables or domain constants.

- $x \Theta y$, where x and y are domain variables and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$). We require that attributes x and y have domains that can be compared by Θ.
- $x \Theta c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules :

- An atom is a formula.
- If P1 is a formulae, then so are ¬P1 and (P1).
- If P1 and P2 are formulae, then so are $P1 \vee P2$, $P1 \wedge P2$, and $P1 \Rightarrow P2$.
- If P1 (x) is a formula in x, where x is a domain variable, then $\exists x (P1 (x))$ and $\forall x (P1(x))$ are also formulae.

As a notational shorthand, we write

$\exists a, b, c (P(a, b, c))$

for $\exists a (\exists b (\exists c (P (a, b, c))))$

https://www.youtube.com/watch?v=LL_eHNQA6wk&index=2&list=PLx5CT0AzDJClCw2DlI7FYUcBn5MfvpZKY

Example Queries

We now give domain-relational-calculus queries for the examples that we considered earlier Note the similarity of these expressions and the corresponding tuple-relational-calculus expressions.

- Find the loan number, branch name, and amount for loans of over $1200:

  $\{<l, b, a> \mid <l, b, a> \in loan \wedge a > 1200\}$

- Find all loan numbers for loans with an amount greater than $1200:

  $\{< l > \mid \exists\, b, a\ (< l, b, a > \in loan \wedge a > 1200)\}$

Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference. In the tuple calculus, when we write ∃ s for some tuple variable s, we bind it immediately to a relation by writing ∃ s ∈ r. However, when we write ∃ b in the domain calculus, b refers not to a tuple, but rather to a domain value. Thus, the domain of variable b is unconstrained until the subformula <l, b, a> ∈ loan constrains b to branch names that appear in the loan relation. For example,

- Find the names of all customers who have a loan from the Perryridge branch and find the loan amount:

$\{<c, a> \mid \exists\, l\ (< c, l > \in borrower$

$\wedge \exists\, b\ (<l, b, a>\in loan \wedge b = \text{“Perryridge”}))\}$

- Find the names of all customers who have a loan, an account, or both at the Perryridge branch:

$\{<c> \mid \exists\, l\ (<c, l> \in borrower$

$\wedge \exists\, b, a\ (<l, b, a> \in loan \wedge b = \text{“Perryridge”}))$

$\vee \exists\, a\ (<c, a> \in depositor$

$\wedge \exists\, b, n\ (<a, b, n> \in account \wedge b = \text{“Perryridge”}))\}$

- Find the names of all customers who have an account at all the branches located in Brooklyn:

$\{<c> \mid \exists\, n\ (<c, n> \in customer) \wedge$

$\forall\, x, y, z\ (<x, y, z> \in branch \wedge y = \text{“Brooklyn”} \Rightarrow$

$\exists\, a, b\ (< a, x, b> \in account \wedge <c, a> \in depositor\ ))\}$

In English, we interpret this expression as "The set of all {customer-name) tuples c such that, for all (branch-name, branch-city, assets) tuples, x, y, z, if the branch city is Brooklyn, then the following is true":

- There exists a tuple in the relation account with account number a and branch name x.

- There exists a tuple in the relation depositor with customer c and account numbers. a"

Expressive Power of Languages

When the domain relational calculus is restricted to safe expressions, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. Since we noted earlier that the restricted tuple relational calculus is equivalent to the relational algebra, all three of the following are equivalent:

- The basic relational algebra (without the extended relational algebra operations)
- The tuple relational calculus restricted to safe expressions
- The domain relational calculus restricted to safe expressions

We note that the domain relational calculus also does not have any equivalent of the aggregate operation, but it can be extended to support aggregation, and extending it to handle arithmetic expressions is straightforward.

Questions

1) Define different database keys
2) Write a short note on Integrity Rules.
3) Mention the features of good relational database design.
4) List the different steps of designing relational database process.
5) Explain Normalization with its form.
6) Explain the relation between Primary and Foreign key.
7) Explain the concept of Relational Algebra.
8) Define Joins and its types.
9) Explain set operators with example.
10) Write short note on Relational division operator.

Multiple Choice Questions

1) Relational Algebra is a __________ query language that takes two relation as input and produces another relation as output of the query.
   a) Relational
   b) Structural
   c) Procedural
   d) Fundamental

2. Which of the following is a fundamental operation in relational algebra ?
   a) Set intersection
   b) Natural join
   c) Assignment
   d) None of the mentioned

3. Which is a join condition contains an equality operator:
   a) Equijoins
   b) Cartesian
   c) Natural
   d) Left

4. Which are the primitive operators of relation algebra as proposed by codd:
   a. Selection
   b. Projection
   c. Cartesian product
   d. Set union
   e. Set difference
   f. Rename
   g. All of these
   h. None of these

5. Which of the following relational algebra operations do not require the participating tables to be union-compatible?

A. Union
   B. Intersection
   C. Difference
   D. Join

6. Cartesian product in relational algebra is
   A. a Unary operator
   B. a Binary operator
   C. a Ternary operator
   D. not defined

7. In precedence of set operators the expression is evaluated from
   a) Left to left

b) Left to right
c) Right to left
d) From user specification

8. Which is a unary operation:
a) Selection operation
b) Primitive operation
c) Projection operation
d) Generalized selection

9. The ___________ operation, denoted by −, allows us to find tuples that are in one relation but are not in another.
a) Union
b) Set-difference
c) Difference
d) Intersection

10. For select operation the ________ appear in the subscript and the ___________ argument appears in the paranthesis after the sigma.
a) Predicates, relation
b) Relation, Predicates
c) Operation, Predicates
d) Relation, Operation