Unit-2 Relational Data Model

Prepared by:

Nilesh Parmar

Outline

- Basic structure: Relation, attribute, domain
- Database schema, Relational Schema, schema diagram
- Relational algebra operations
- Additional Relational algebra operations
- Extended Relational algebra operations
- Null values
- Modification of database: Deletion, Insertion, Update
- Integrity constraints (Completed in unit-1)

Basic structure: Relation, attribute, domain

- A relational database is a collection of tables, each of which is assigned a unique name.
- A row in a table represents a relationship among a set of values..
- The concept of table is closely related with concept of relation in mathematics, from which the relational model has come up. Following to that, concept of relation has been introduced.
- Consider the table named as account:

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

- This table has three column headers: account-number, branch-name, and balance.
- Following the terminology of the relational model, we refer to these headers as attributes.

Cntd…

- For each attribute, there is a set of permitted values, called the domain of that attribute. For example, domain of attribute branch-name is considered as, the set of all branch names.
- Let D1 denote the set of all account numbers, D2 the set of all branch names, and D3 the set of all balances. Any row of account must consist of a triplet (v1, v2, v3), where v1 is an account number, v2 is a branch name, and v3 is a balance. In general, account will contain only a subset of the set of all possible rows. Therefore, account is a subset of: D1 x D2 x D3
- In general, a table of 'n' attributes must be a subset of: $D1 \times D2 \times \cdots \times Dn-1 \times Dn$
- **Relation:** It is a subset of a Cartesian product of a list of domains.
- Because tables are essentially relations, we shall use the terms relation and tuple in place of the terms table and row.
- A tuple variable is a variable that stands for a tuple; in other words, a tuple variable is a variable whose domain is the set of all tuples.

Cntd…

- For tuple variable ‘t’ and relation ‘r’, we can say that $t \in r$ to denote that tuple ‘t’ in relation ‘r’. The order in which tuples appear in a relation is irrelevant, since a relation is a set of tuples.
- Types of domain:

1) Atomic domain: If elements of the domain are indivisible units, it is called as atomic domain.
2) Non-atomic domain: If elements of the domain are further divisible, it is called as non-atomic domain.

- For example, the set of integers is an atomic domain, but the set of all sets of integers is a non-atomic domain.
- We require that, for all relations ‘r’, the domains of all attributes of ‘r’ be atomic.
- NULL value can be member of any domain. NULL means unknown or does not exist. NULL values cause a number of difficulties when we access or update the database, So, it should be eliminated if possible.

Database Schema and Relation Schema

- Database schema is the logical design of the database. It contains details of all relations of database.
- Database instance is a snapshot of the data in the database at a given instant of time.
- Relation schema consists of a list of attributes and their corresponding domains + integrity constraints.
- The concept of a relation schema corresponds to the programming-language notion of type definition. The concept of a relation corresponds to the programming-language notion of a variable.

➢ Convention for relation & relational schema:

- We will use lowercase names for relations, and names beginning with an uppercase letter for relation schemas.

Cntd…

- Example: relation ‘account’ is given with attribures account-number, branch-name and balance. Schema for this relation is given as:

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

- account is a relation on Account-schema, it is depicted by: account(Account-schema)
- Relation instance is a snapshot of data of relation at any moment of time. Concept of relation instance is similar to value of a variable in programming language. Value of variable may change with time; similarly contents of relation instance may change with time as the relation is updated.

Keys

- Conceptually individual tuple (row) are distinct from a database perspective, however the difference among them must be specified in terms of their attributes. No two tuples are allowed to have exactly same value for all attributes.
- Key uniquely identifies every tuple of a relation.
- Super key: It is a set of one or more attributes that, taken collectively, identifies each tuple in a relation uniquely.
- Example: In our account table (relation having attributes- account-number, branch-name, balance) account-number attribute is sufficient to distinguish all the tuples from one another. So, account-number is a super key. Similarly, {account-number + branch-name} combinedly work as super key. branch-name alone can not work as super key because several branch might have same name. Here, we can see that any attribute which is taken collectively with account-number becomes super key.

Cntd...

- Candidate Key: Minimal super keys are called as candidate keys.
- Example: Consider a relation- customer with attributes- customer-id, customer-name, customer-city. In this relation several distinct set of attributes work as a candidate key. Suppose {customer-name+ customer-city} combinedly sufficient to distinguish every tuple of customer relation. So, both {customer-id} and {customer-name+ customer-city} are candidate keys. {customer-id + customer-name} is also able to uniquely identify tuples, but it is not candidate key, as it is not minimal because customer-id alone can identify each tuple uniquely. {customer-id + customer-name} can be said as super key.
- Primary key: Out of all candidate key, one key is chosen by database designer as principal means of identifying tuples within relation.
- Primary key should be chosen such that its attributes are changed never or very rarely. Means address field of customer can never be a primary key as it is changed very likely.

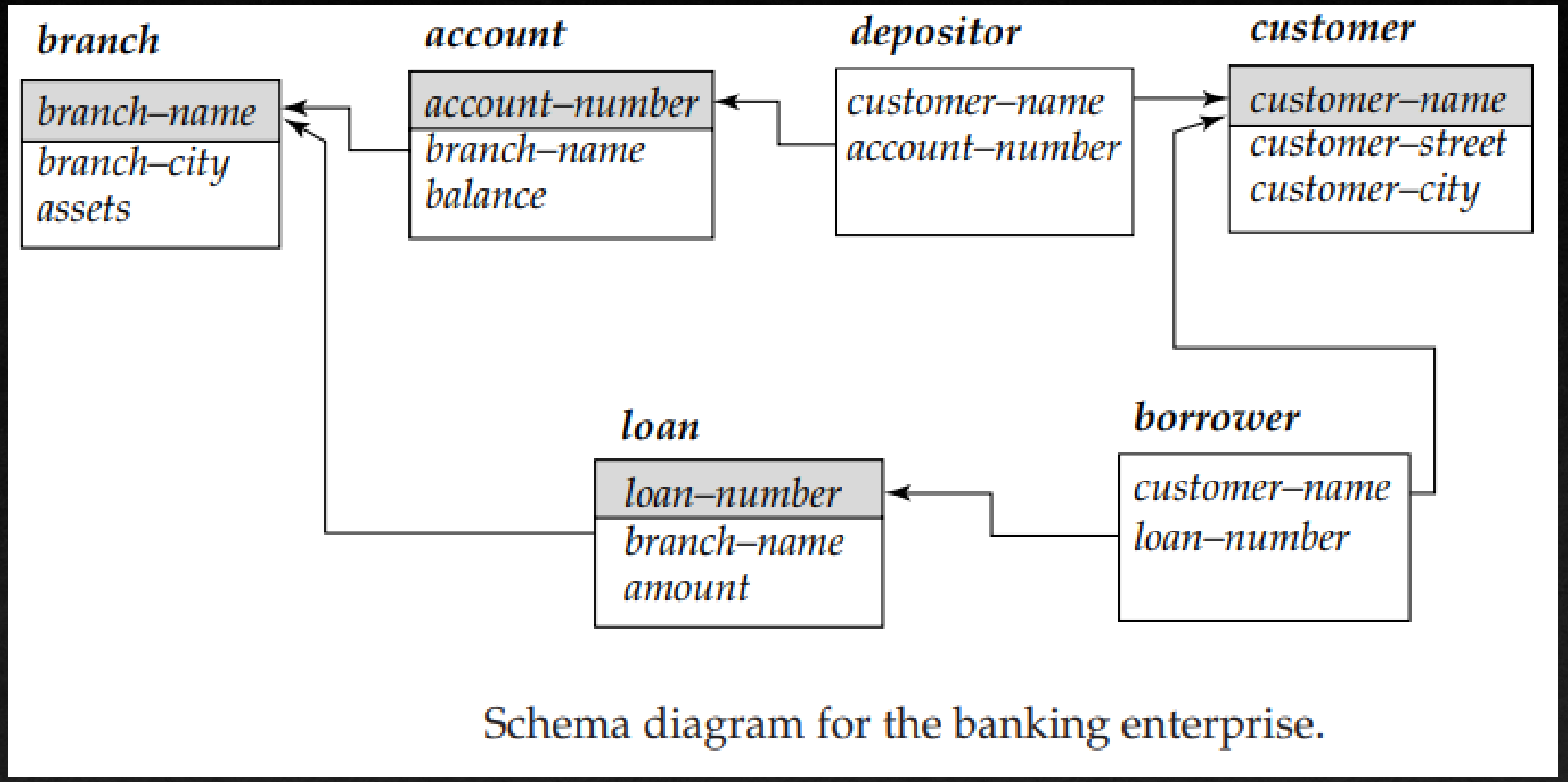
Cntd...

- Foreign Key: One relation includes primary key of another relation as an attribute. This attribute is called as foreign key.
- If r1 relation uses primary key of r2, it is called as foreign key of r1 relation. Relation r1 is called as referencing relation and r2 is called as referenced relation.
- Example- Consider one relation branch with attributes: branch-name, branch-city, assets. Another relation account with attributes: account-number, branch-name, balance. Branch-name is called as foreign key for account relation as it is primary key of relation branch.

Schema diagram

- A database schema along with primary key and foreign key dependencies, can be depicted pictorially by schema diagram.
- In schema diagram, each relation appears as a box, with the attributes listed inside it and the relation name above it. Primary key attributes are listed above one horizontal line inside box. Foreign key dependencies appear as arrows from referencing relation to referenced relation.
- Do not confuse a schema diagram with an E-R diagram. In particular, E-R diagrams do not show foreign key attributes explicitly, whereas schema diagrams show them explicitly.

Schema diagram



Schema diagram for the banking enterprise.

branch-name	branch-city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

The *branch* relation

account-number	branch-name	balance
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

The *account* relation

customer-name	account-number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

The *depositor* relation.

customer-name	customer-street	customer-city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

The *customer* relation.

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

The *loan* relation.

customer-name	loan-number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

The *borrower* relation.

Relations we will be using further as examples.

Relational Algebra: Fundamental Operations

- Relational algebra is a procedural query language. Now, we will learn all type of relational algebra operations.

Fundamental Operations:

1) Select operation: It selects tuples that satisfy a given predicate (condition). 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.

- In loan relation, to select tuples where branch is 'perryridge', we can write:

$$\sigma_{\text{branch-name} = \text{“Perryridge”}} (\text{loan})$$

- In general, comparisons are done using =, =, , ≥ in the selection predicate. Further, we can combine several predicates into a larger predicate by using the connectives and (∧), or (∨), and not (¬). Thus, to find tuples pertaining to loans of more than Rs. 1200 made by the Perryridge branch, we write: $\sigma_{\text{branch-name} = \text{“Perryridge”} \wedge \text{amount}>1200} (\text{loan})$

Cntd…

2) Project operation: It is a unary operation that returns specific attributes of relation or expression (relational algebra expression).

- Since a relation is a set, any duplicate rows are eliminated.
- Projection is denoted by the uppercase Greek letter pi (Π). We denote those attributes that we wish to appear in the result as a subscript to Π. The argument relation is written in parentheses.
- If we want to list all loan numbers and the amount of the loan, we write:

$$\Pi_{\text{loan-number, amount}}(\text{loan})$$

3) Composition of Relational operations: Result of a relational operation is itself a relation. More than one relational algebra operations can be applied one after another to get desired result. Applying multiple operations on relation creates an relational algebra expression.

- Composing relational algebra operations into relational algebra expression is like composing arithmetic operations (like +,-,*, ÷) into arithmetic expressions.

Cntd...

- Example: Consider a question: find those customers who live in Harrison city. We write:

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

4) Union operation: Union operation is used to add output of multiple relational algebra expressions which are compatible. Symbol is- U. Consider a question: find the names of all bank customers who have either an account or a loan or both. To answer this question, we need the information from depositor relation and from borrower relation.

1) Names of all customers with a loan in the bank: $\Pi_{\text{customer-name}}$ (borrower)

2) Names of all customers with an account in the bank: $\Pi_{\text{customer-name}}$ (borrower)

- To answer the question, we need to perform union of these 2 expressions:

$$\Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}} (\text{depositor})$$

- Since relations are sets, duplicate values are eliminated from result. Result does not contain details of customers who neither have account nor loan.

Cntd…

- Union is a binary operation. Before we perform union, we need to ensure that relations which are participating in union are compatible.
- For a union operation r U s to be valid, two conditions must hold:

1) The relations r and s must have the same number of attributes. (Same arity)
2) The domains of the i^{th} attribute of r and the i^{th} attribute of s must be the same, for all i.

5) Set-difference operation: The set-difference operation, denoted by –, allows us to find tuples that are in one relation but are not in another. The expression r – s produces a relation containing tuples those are in r but not in s.

- Consider question: find all customers of the bank who have an account but not a loan. Answer:

$$\Pi_{\text{customer-name}}(\text{depositor}) - \Pi_{\text{customer-name}}(\text{borrower})$$

- Like union operation, set difference is applied on relation which are compatible.
- For a set difference operation r-s to be valid, r and s must have same number of attributes, and domains of the i^{th} attribute of r and the i^{th} attribute of s must be same.

Cntd…

6) Cartesian-product operation: The Cartesian-product operation, denoted by a cross (×), allows us to combine information from any two relations. We write the Cartesian product of relations r1 and r2 as r1 × r2.

- It is possible that same attribute name may appear in both r1 and r2. We can distinguish such attributes by attaching its relation name with an attribute.
- For example, borrower and loan relation has one common attribute: loan-number. Cartesian product of these relations can be written as: r= borrower X loan. So, relation schema becomes (borrower.customer-name, borrower.loan-number, loan.loan-number, loan.branch-name, loan.amount).
- The attributes which are appearing only once, can be written without attaching table name to them. So, relation schema for r= borrower X loan becomes (customer-name, borrower.loan-number, loan.loan-number, branch-name, amount).
- Assume that we have n1 number of tuples in r1 and n2 number of tuples in r2. Then, cartesian product of r1 and r2 i.e. r1 X r2 contains n1 ∗ n2- number of tuples.

Cntd...

- Cartesian product associates every tuple from r1 with every tuple of r2. Thus, it shows all possibilities among two tables.
- Let us perform r = borrower X loan and see the result.

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

The *borrower* relation.

X

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

The *loan* relation.

=

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Cntd…

- Consider the question: find the names of all customers who have a loan at the Perryridge branch.
- We will find answer in 3 steps. First step, we need to find out which customers are having loan. We will get that information from 2 relations: loan and borrower. We write:

$$\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan})$$

- Step two, we will find name of customers who have loan at Perryridge branch.

$$\sigma_{\text{branch-name} = \text{“Perryridge”}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan}))$$

- Step three, we need only customer name in the result. So final query is:

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-name} = \text{“Perryridge”}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan})))$$

Cntd…

7) 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. This is achieved by the rename operator, denoted by the lowercase Greek letter rho (ρ).

- ρ_x (E) returns the result of expression E under the name 'x'.
- A relation 'r' is also a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation 'r' to change the name of that relation.
- Second form of the rename operation is also there. Assume that a relational algebra expression E has 'n' attributes. Then, the expression

$$\rho_{x\,(A1,A2,\ldots,An)}\,(E)$$

returns the result of expression E under the name x, and with the attributes renamed to A1, A2,...,An.

- To understand this question, we consider the question "Find the largest account balance in the bank."

Cntd…

- To solve this question, we use a strategy with 2 steps.

(1) Compute first a temporary relation consisting of those balances that are not the largest.

(2) Take the set difference between the relation $\Pi_{balance}$ (account) and the temporary relation computed in step-1, to obtain the result.

Step-1: To compute the temporary relation, we need to compare the values of all account balances. We achieve this by computing the cartesian product (account × account) and comparing value of any two balances appearing in one tuple. For that we need to distinguish between 'balance' attribute of both tables. For that we can rename the table and create its another reference, thus we can refer the relation twice without any ambiguity. We can now write the temporary relation that consists of the balances that are not the largest:

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

Here, we have created 2 instances of table account. One instance is account itself and another is 'd'.

Cntd…

- Query of step-1 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 largest one.

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

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

Concept of Relational Algebra

- A basic expression in the relational algebra consists of either one of the following: A relation in the database OR A constant relation.
- A constant relation is, listing its tuples within { }.
- For example { (A-101, Downtown, 500) (A-215, Mianus, 700) }
- A general expression in relational algebra is constructed out of smaller subexpressions. Let E1 and E2 be relational-algebra expressions. Then, following are all relational algebra expressions:
- E1 ∪ E2
- E1 – E2
- E1 × E2
- σ_P (E1), where P is a predicate (condition) on attributes in E1
- Π_S(E1), where S is a list consisting of some of the attributes in E1
- ρ_x (E1), where x is the new name for the E1

Additional Operations

- If we restrict ourselves to just the fundamental operations, certain common queries are lengthy to express. Additional operations do not add any power to the algebra, but simplify common queries.

1) Set- intersection operation: This operation can be rewritten in terms of fundamental set-difference operations. Intersection of R & S can be shown as:

$$R \cap S = R-(R-S)$$

Example: find all customers who have both a loan and an account. So, we can write:

$$\Pi_{\text{customer-name}}(\text{borrower}) \cap \Pi_{\text{customer-name}}(\text{depositor})$$

2) Natural Join Operation: The natural join is a binary operation that allows us to combine certain selections and a cartesian product into one operation. It is denoted by "join" and symbol is: ⋈.

- Consider the question: Find the names of all customers who have a loan at the bank.

Cntd…

- To answer this question, first form the Cartesian product of the borrower and loan relations. Then, we select those tuples that pertain to only the same loan-number, followed by the projection of the resulting customer-name.

$$\Pi_{\text{customer-name}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan}))$$

- This query can be re-written using natural join operation:

$$\Pi_{\text{customer-name}} (\text{borrower} \bowtie \text{loan})$$

- Join operation finds common attribute in both tables. Then it checks for same value on common attribute. It considers each such pair which follows condition and combine these pair of tuples into single tuple. Common attribute is considered only once in resultant relation. Natural join removes duplicates from result.
- If two relations R and S have no common attribute means $R \cap S = \phi$ then $R \bowtie S = R \times S$
- We can also perform join of 3 tables.

Cntd…

- Example: find the names of all branches with customers who have an account in the bank and who live in Harrison.

$$\Pi_{\text{branch-name}} (\sigma_{\text{customer-city} = \text{“Harrison”}} (\text{customer} \bowtie \text{account} \bowtie \text{depositor}))$$

- Directly joining 3 tables is not possible. It is done step by step. There are 2 possibilities.

$$(\text{customer} \bowtie \text{account}) \bowtie \text{depositor} \quad \text{OR} \quad \text{customer} \bowtie (\text{account} \bowtie \text{depositor})$$

- Result of both ways are same. So, we can say that natural join is associative.
- Consider a question: find all customers who have both a loan and an account at bank.

The answer is: $\Pi_{\text{customer-name}} (\text{borrower} \bowtie \text{depositor})$

- Remember, we have solved same query by using set intersection. Answer was:

$$\Pi_{\text{customer-name}} (\text{borrower}) \cap \Pi_{\text{customer-name}} (\text{depositor})$$

Cntd…

- Theta join: It is an extension to natural join. Theta join for relations R & S is given as:

  $R \bowtie_{\theta} S = \sigma_{\theta} (R \times S)$ Where θ is some condition or predicate.

- In theta join, predicate θ may contain any relational operator like =, !=, >, <, >=, <= to compare attributes of two relations.

3) Division operation: This operation is suited for queries that include phrase "for all". It is denoted by ÷.

- Consider: find all customers who have an account at all the branches located in Brooklyn.
- We will solve it step by step. First we can obtain all branches in Brooklyn. We can write:

  $\Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{“Brooklyn”}} (\text{branch}))$ (Let's denote this query as r1)

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

  $\Pi_{\text{customer-name, branch-name}} (\text{depositor} \bowtie \text{account})$ (Let's denote this query as r2)

Cntd…

- Final step, 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 means r2 ÷ r1. We formulate the query by writing:

$$\Pi_{\text{customer-name, branch-name}} (\text{depositor} \bowtie \text{account}) \div \Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{“Brooklyn”}} (\text{branch}))$$

- Let us understand this operation based on intermediate and final results.

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

Result of r2 query

Result of $\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie\ account)$.

÷

branch-name
Brighton
Downtown

Result of r1 query

Result of $\Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch))$.

=

customer-name
Johnson

Final result

- The result of this expression is a relation that has the schema (customer-name) and that contains only one tuple (Johnson) because only “Johnson” has account at “all” branches of “Brooklyn” city, no other customer satisfy that condition.

Cntd…

4) Assignment operation: Sometimes it is convenient to assign parts of relational algebra expression to temporary relational variables. The assignment operation denote by '←' works like assignment in programming language.

- We will continue example of division operation to understand assignment operation: Find all customers who have an account at all branches located in Brooklyn.

  $temp1 \leftarrow \Pi_{branch\text{-}name} (\sigma_{branch\text{-}city = \text{“Brooklyn”}} (branch))$

  $temp2 \leftarrow \Pi_{customer\text{-}name,\ branch\text{-}name} (depositor \bowtie account)$

  Final query is: $temp1 \div temp2$

- Assignment operation does not provide any output. It simply assigns result of a relational algebra expression to a relational variable. This variable can be used in further queries.

Cntd…

- In assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.
- For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations leads to database modification.
- Note that the assignment operation does not provide any additional power to the algebra. It is just a convenient way to express complex queries.

Extended Relational Algebra Operations

- The basic relational-algebra operations have been extended in several ways.
- 1) Generalized Projection: The generalized-projection operation extends the projection operation by allowing arithmetic functions to be used in the projection list. The generalized projection operation has the form:

$$\Pi_{F1,F2,\ldots,Fn}(E)$$

where E is any relational-algebra expression, and each of F1, F2,...,Fn is an arithmetic expression involving constant values and attributes in the schema of E. Sometimes, the arithmetic expression may be simply an attribute or a constant.

Example: We want to show 10% increase in salary of employee. Query for this question is given as:

$$\Pi_{emp\_id,\ emp\_name,\ salary+0.1*salary,\ emp\_city}(employee)$$

Cntd...

- The attribute resulting from the expression (salary+0.10*salary) does not have a name. We can apply the rename operation to the result of generalized projection in order to give it a name.

$$\Pi_{\text{emp\_id, emp\_name, (salary+0.1*salary) as salary\_hike, emp\_city}} \text{(employee)}$$

Third attribute of generalized projection has been given a new name 'salary_hike'.

2) Aggregate functions: Such functions take a collection of values and return a single value as a result. For example, the aggregate function sum takes a collection of values and returns the sum of the values. Other aggregate function are sum, avg, count, min and max. Count returns the number of the elements in the collection.

- For aggregate function, symbol '$\mathcal{G}$' is used which is called as- 'calligraphic G'. The relational-algebra operation $\mathcal{G}$ signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied.

Cntd...

- If we want to find out the total sum of balance of all the account holders of the bank. We can write query: $\mathcal{G}_{\text{sum(balance)}}$ (account)
- The result of above query is a relation with a single attribute, containing a single row with a numerical value corresponding to the sum of balance of all the account holders.
- Sometimes we need to eliminate multiple occurrences of a value before computing an aggregate function. If we do want to eliminate duplicates, we use one extra word "distinct".
- Consider a question: Find number of branch-cities appear in the branch table.

$$\mathcal{G}_{\text{count(branchcity)}} \text{(branch)}$$

It is possible that some branch-cities are repeated in the table, so 'count' operation doesn't give actual answer. Now, to retain only unique branch-city values, we use 'distinct':

$$\mathcal{G}_{\text{count-distinct(branchcity)}} \text{(branch)}$$

Cntd…

- Suppose we want to find the sum of balance of all account holders at each branch of the bank separately, rather than the sum for the entire bank. To do so, we need to partition the relation ‘account’ into groups based on the branch, and then we will apply the aggregate function on each group. Following query gives the required result:

$$_{\text{branchname}}\mathcal{G}_{\text{sum(balance)}}(\text{account})$$

- In the above expression, the attribute branchname in the left-hand subscript of $\mathcal{G}$ indicates that the input relation ‘account’ must be divided into groups based on the value of branchname.
- The expression sum(balance) in the right-hand subscript of $\mathcal{G}$ indicates that for each group of tuples (means, for each branch), the aggregation function ‘sum’ must be applied on the collection of values of the ‘balance’ attribute. The output relation consists of tuples with the branchname and the sum of balance for that branch.

Cntd...

- 3) Outer join: It is an extension of natural join operation to deal with missing information.
- Consider two relations with their schemas given here:
- employee (employee-name, street, city) & ft-works (employee-name, branch-name, salary)
- ft-works contains data of full time employees.
- employee and ft_works tables are given below:

employee-name	*street*	*city*
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

employee-name	*branch-name*	*salary*
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

The *employee* and *ft-works* relations.

- Now, we want to generate a single relation with all the information (street, city, branch name, and salary) about full-time employees.

Cntd…

- To get result, a possible approach would be to use the natural join operation:

  employee ⋈ ft_works

- The result of this join operation is:

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

The result of *employee* ⋈ *ft-works*.

- Notice that we have lost the street and city information about Smith, since the tuple describing Smith is absent from the ft-works relation; similarly, we have lost the branch name and salary information about Gates, since the tuple describing Gates is absent from the employee relation.
- We can use the outer-join operation to avoid this loss of information. There are actually three forms of the operation: A) left outer join, denoted as ⟕

  B) right outer join, denoted as ⟖

  C) full outer join, denoted as ⟗

Cntd…

A) Left outer join (⟕) : It considers all the tuples that matched condition as well as all other remaining tuples that did not match condition in right relation and pads the tuples with NULL values for all attributes from right relation. All information from the left relation is present in the result of the left outer join.

- Let's apply left outer join between: employee and ft_works table i.e. Employee ⟕ ft_works

The result is:

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*

Result of *employee* ⟕ *ft-works*.

B) Right outer join (⟖): It considers all tuples that matched condition as well as all other remaining tuples of right relation that did not match condition with left relation and pads tuples with NULL values for all attributes from left table.

- Let's apply right outer join between: employee and ft_works i.e. Employee ⟖ ft_works.

Cntd…

- The result is:

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	*null*	*null*	Redmond	5300

Result of *employee* ⟕ *ft-works*.

- All information from the right relation is present in the result of the right outer join.

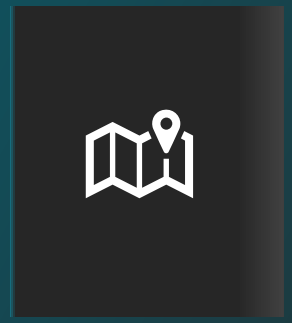
C) Full outer join (⟗): It perform union of left and right outer join, padding tuples from left relation that did not match any tuple from right relation, as well as tuples from right relation that did not match any tuple from left relation.

- Let's apply full outer join between: employee and ft_works i.e. Employee ⟗ ft_works.
- The result is:
- All information from left and right both relations is present in the result of full outer join.

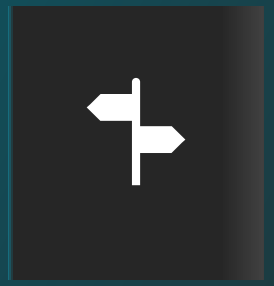
employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*
Gates	*null*	*null*	Redmond	5300

Result of *employee* ⟗ *ft-works*.

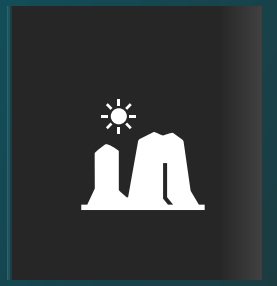
Summery: Relational Algebra Operations



Fundamental Operations: Select, Project, Composition of relational operations, Union, Set-difference, Cartesian Product, Rename Operation



Additional relational algebra operations: Set-Intersection, Natural join, Division, Assignment



Extended relational algebra operations: Generalized operation, Aggregate functions, Outer join.

NULL values

- Whenever we deal with NULL values, sometimes results are unexpected. Hence, operations and comparisons on NULL values should be avoided, whenever possible.
- NULL indicates “value unknown or nonexistent”.
- Any arithmetic operations (such as +, −, ∗, /) involving null values must return a NULL result.
- Similarly, any comparisons (such as <, <=, >, >=, =) involving a null value evaluate to special value NULL (unknown); we cannot say for sure whether the result of the comparison is true or false, so we say result is unknown i.e. NULL.
- Let us check what happens when Boolean operations (and, or, not) involves null values (unknown) as an operand.
- and: (true and unknown) = unknown; (false and unknown) = false;

  (unknown and unknown) = unknown.
- or: (true or unknown) = true; (false or unknown) = unknown;

  (unknown or unknown) = unknown.
- not: (not unknown) = unknown

Cntd…

- How the different relational operations deal with null values.
- select: The selection operation evaluates predicate P in $\sigma_P(E)$ on each tuple t in E. If the predicate returns the value true, t is added to the result. Otherwise, if the predicate returns unknown (null) or false, t is not added to the result.
- join: Joins can be expressed as a cross product followed by a selection. Thus, join operations handle nulls in the same way as selection does. In a natural join of r and s, say $r \bowtie s$, if two tuples, $t_r \in r$ and $t_s \in s$, both have a null value in a common attribute, then the tuples do not match.
- projection: The project operation treats nulls just like any other value when eliminating duplicates. Thus, if two tuples in the projection result are exactly the same, and both have nulls in the same fields, they are treated as duplicates

Cntd…

- union, intersection, set-difference: These operations treat nulls just as the projection operation does. They treat tuples as duplicates that have the same values for all fields (attributes), even if some of the fields have null values in both tuples.
- generalized projection: Arithmetic operations on constants and attributes involving null values result in null value. Duplicate tuples containing null values are handled same as the projection operation.
- aggregate: When nulls occur in grouping attributes, the aggregate operation treats them same as the project operation: If two tuples are the same on all grouping attributes, the operation places them in the same group, even if some of their attribute values are null.
- When nulls occur in aggregated attributes, the operation deletes null values, before applying aggregation.
- outer join: outer join follows same rule as natural join to compare two tuples of relations. Addition to that outer join also includes unmatched tuples of left hand table or right hand table or both tables based on type of outer join. Thus, unmatched tuples are added to the result and for non-existent values of some attributes, nulls are padded.

Modification of the Database

- We can add, remove or change information in database using Insert, delete and update operations.

1) Deletion: We can delete only whole tuples, we cannot delete values of only particular attributes. In relational algebra a deletion is expressed by:

$r \leftarrow r - E$ where r is a relation and E is a relational algebra query

- Let us understand delete operation with some examples. Consider question: Delete all the details of smith's account. We can write:

$$depositor \leftarrow depositor - \sigma_{customer\text{-}name = \text{“Smith”}} (depositor)$$

- Question: Delete all loans where amount is in range of 0 to 50.

$$loan \leftarrow loan - \sigma_{amount \geq 0 \wedge amount \leq 50} (loan)$$

- Delete all accounts at branches located in Brooklyn city.

$$r1 \leftarrow \sigma_{branch\text{-}city = \text{“Brooklyn”}} (account \bowtie branch)$$

$$r2 \leftarrow \Pi_{accountno,\ branchname,\ balance} (r1)$$

$$account \leftarrow account - r2$$

Here, we are using temporary relations r1 and r2 to get the result.

Cntd…

2) 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.

- The attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity (number of attribute values must be same as number of attributes in the relation.)
- Relational algebra expression for Insertion is: $r \leftarrow r \cup E$

  where r is a relation and E is a relational-algebra expression.
- We can insert a single tuple by taking 'E' as a constant relation containing one tuple.
- Suppose that we wish to insert data which shows that Smith has Rs. 1200 in account A-973 at the "Perryridge" branch. We can write:

  account ← account ∪ {(A-973, "Perryridge", 1200)}

  depositor ← depositor ∪ {("Smith", A-973)}

Cntd…

3) Updating: Sometimes we wish to change a value in a tuple without changing all values in the tuple. We can use the generalized-projection operator to achieve this task.

$$r \leftarrow \Pi_{F1,F2,\ldots,Fn}(r)$$

Where, Fi = { i^{th} attribute of r, if the i^{th} attribute is not updated
It is an expression involving only constants and the attributes of 'r', that gives the new value for the attribute.

- Example: Interest of 5% is given to all account holders i.e. balance of all accounts is to be increased by 5%. Query will be:

$$account \leftarrow \Pi_{accountno,\ branchname,\ balance * 1.05}(account)$$

- Suppose, question is: Accounts with balances over Rs. 10,000 receive 6% interest, whereas all others receive 5%. Query will be:

$$account \leftarrow \Pi_{accountno,\ branchname,\ balance * 1.06}(\sigma_{balance>10000}(account)) \cup \Pi_{accountno,\ branchname,\ balance * 1.05}(\sigma_{balance \leq 10000}(account))$$