UNIT-2

Relational Model in DBMS

Relational Model was proposed by E.F. Codd to model data in the form of relations or tables. After designing the conceptual model of Database using ER diagram, we need to convert the conceptual model in the relational model which can be implemented using any RDBMS languages like Oracle SQL, MySQL etc.

RELATIONAL MODEL (RM) represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

Relational Model Concepts

1. **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student_Rollno, NAME,etc.
2. **Relation** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. **Tuple** – It is nothing but a single row of a table, which contains a single record.
4. **Relation Schema:** A relation schema represents the name of the relation with its attributes.
5. **Degree:** The total number of attributes which in the relation is called the degree of the relation.
6. **Cardinality:** Total number of rows present in the Table.
7. **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
8. **Relation key** - Every row has one, two or multiple attributes, which is called relation key.
9. **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain.

Properties of relational databases

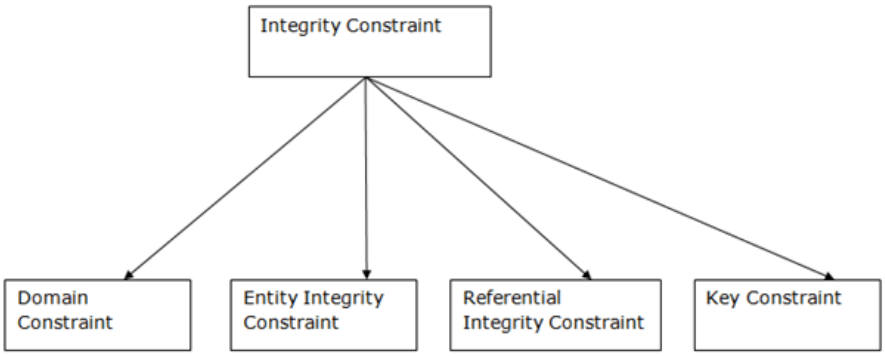
Relational databases have the following properties:

- Values are atomic.
- All of the values in a column have the same data type.
- Each row is unique.
- The sequence of columns is insignificant.
- The sequence of rows is insignificant.
- Each column has a unique name.
- Integrity constraints maintain data consistency across multiple tables.

Integrity Constraints

Integrity constraints are a set of rules. It is used to maintain the quality of information. Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected. Thus, integrity constraint is used to guard against accidental damage to the database.

Types of Integrity Constraint



1. Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.

Example:

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

2. Referential integrity constraints

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation.

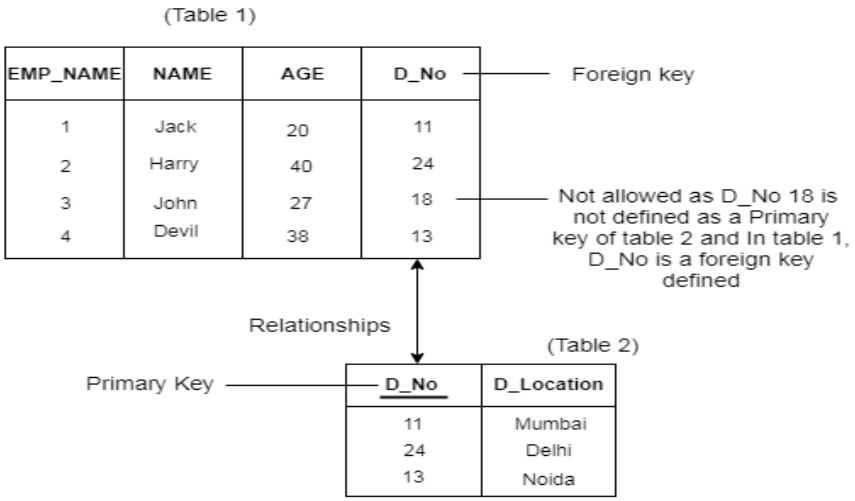
To define referential integrity more formally, first we define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas *R*1 and *R*2. A set of attributes FK in relation schema *R*1 is a **foreign key** of *R*1 that **references** relation *R*2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of *R*2; the attributes FK are said to **reference** or **refer to** the relation *R*2.

2. A value of FK in a tuple *t*1 of the current state *r*1(*R*1) either occurs as a value of PK for some tuple *t*2 in the current state *r*2(*R*2) *or is NULL*. In the former case, we have *t*1[FK] = *t*2[PK], and we say that the tuple *t*1 **references** or **refers to** the tuple *t*2.

In this definition, *R*1 is called the **referencing relation** and *R*2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from *R*1 to *R*2 is said to hold.

Example:



3. Domain constraints

- Domain constraints specify that within each tuple, the value of each attribute *A* must be an atomic value from the domain dom(*A*). Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

  Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

4. Key constraints

A *relation* is defined as a *set of tuples.* By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema *R* with the property that no two tuples in any relation state *r* of *R* should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples *t*1 and *t*2 in a relation state *r* of *R*, we have the constraint that:

*t*1[SK]=*t*2[SK]

Any such set of attributes SK is called a **superkey** of the relation schema *R*. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state *r* of *R* can have the same value for SK.

A **key** *K* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K*_ that is not a superkey of *R* anymore. Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.

2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold. This property is not required by a superkey.

Example: Consider the STUDENT relation with attributes {name, ssn, phone, age, address, cgpa}

The attribute {ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn. Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey.

A relation schema may have more than one key. In this case, each of the key is called a candidate key. Example: CAR relation has two candidate keys license_number, engine_serial_number. It is common to designate one of the candidate keys as the primary key of the relation.

- **NOT NULL Constraint**
  By default, a column can hold NULL values. The NOT NULL constraint enforces a column to NOT accept NULL values.

  The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

  Example:

  CREATE TABLE Persons
  (
  P_Id number NOT NULL,

```
LastName varchar2(25) NOT NULL,
FirstName varchar2(25) NOT NULL,
Address varchar2(30),
City varchar2(15)
)
```

- **Unique Constraint**

The UNIQUE constraint ensures that all values in a column are different. Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns. A PRIMARY KEY constraint automatically has a UNIQUE constraint. However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

Example:

```
CREATE TABLE CUSTOMERS(
        ID   NUMBER          NOT NULL,
        NAME VARCHAR2 (20)    NOT NULL,
        EMAILID  VARCHAR2(20) UNIQUE,
        ADDRESS  CHAR (25) ,
        PHONENO number(10) UNIQUE
      );
```

- **Check Constraint**

The CHECK constraint is used to limit the value range that can be placed in a column. If you define a CHECK constraint on a single column it allows only certain values for this column. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

Example:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar2(25) NOT NULL,
   FirstName varchar2(25),
   Age number CHECK (Age>=18)
);
```

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 algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra.

The relational algebra is very important for several reasons:

1. It provides a formal foundation for relational model operations.

2. It is used as a basic for implementing and optimizing queries in RDBMS.
3. Some of its concepts are incorporated into the SQL standard query language for RDBMS.

Relational algebraic operations can be divided in two categories:

1. Relation Oriented Operations
a. Select (Basic)
b. Project (Basic)
c. Rename (Basic)
d. Join (Extended)
e. Division (Extended)

2. SET Oriented Operations
a. Union (Basic)
b. Intersection (Basic)
c. Cartesian Product (Basic)
d. Difference (Extended)

Select, Project and rename operations are called **Unary** operations because they operate on one relation. Rest of all operations are called as **Binary** operations as they operate on two relations.

UNARY Relational Operations

1. SELECT Operation

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**. The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded. 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.

Syntax: The SELECT operation is denoted by
$\sigma_{<\text{selection condition}>}(R)$

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form:

<attribute name> <comparison op> <constant value>
or
<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of *R*, <comparison op> is normally one of the operators {<, >, <=, >=, =, !=}, and <constant value> is a constant value from the attribute domain.

Consider the following Loan 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

Select the tuples from loan relation for Perryridge branch.

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

Select all tuples in which the amount lent is more than $1200.

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

Find the tuples pertaining to loans of more than $1200 made by Perryridge branch.

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

loan-number	branch-name	amount
L-15	Perryridge	1500
L-16	Perryridge	1300

2. PROJECT Operation

The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. The result of the PROJECT operation can be visualized as a *vertical partition* of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns. Projection is denoted by the uppercase Greek letter pi (Π).We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses.

Syntax: The general form of the PROJECT operation is

$\Pi_{\text{<attribute list>}}(R)$

Write the query to list all loan numbers and the amount of the loan.

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

loan-number	amount
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Composition of Relational Operations

For most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

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

Figure 3.4 The *customer* relation.

Find those customers who live in Harrison.

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

3. Rename Operation

RENAME operation can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

$$\rho_{S(B1,\,B2,\,\ldots,\,Bn)}(R) \quad \text{or} \quad \rho_{S}(R) \quad \text{or} \quad \rho_{(B1,\,B2,\,\ldots,\,Bn)}(R)$$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and $B1$, $B2$, ..., Bn are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are ($A1$, $A2$, ..., An) in that order, then each Ai is renamed as Bi.

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

Figure 3.1 The *account* relation.

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

$$\Pi_{account.balance} \left(\sigma_{account.balance < d.balance} \left(account \times \rho_d \left(account\right)\right)\right)$$

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

$$\Pi_{balance} \left(account\right) - \Pi_{account.balance} \left(\sigma_{account.balance < d.balance} \left(account \times \rho_d \left(account\right)\right)\right)$$

balance
900

Largest account balance in the bank.

SET Operations (Binary Operations)

The set operations union, intersection and difference operations require that the tables involved be "Union Compatible". Two relations are said to be union compatible if the following conditions are satisfied:

- The two relations/tables must contain the same number of columns i.e. they have same degree.
- Each column of the relation must be either the same data type as the corresponding column of second relation or convertible to the same type as corresponding of the second.

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

Figure 3.5 The *depositor* 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

Figure 3.7 The *borrower* relation.

1. UNION

The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

Consider a query to find the names of all bank customers who have either an account or a loan or both.

We know how to find the names of all customers with a loan in the bank:

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

We also know how to find the names of all customers with an account in the bank:

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

To answer the query, we need the **union** of these two sets; that is, we need all customer names that appear in either or both of the two relations.We find these data by the binary operation union, denoted, as in set theory, by $\cup$. So the expression needed is

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

Output:

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Names of all customers who have either a loan or an account.

2. Intersection Operation

The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write

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

Output:

customer-name
Hayes
Jones
Smith

Customers with both an account and a loan at the bank.

Note that we can rewrite any relational algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$$r \cap s = r - (r - s)$$

Thus, set intersection is not a fundamental operation and does not add any power to the relational algebra. It is simply more convenient to write $r \cap s$ than to write $r - (r - s)$.

3. 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 those tuples in r but not in s.

We can find all customers of the bank who have an account but not a loan by writing

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

Output:

customer-name
Johnson
Lindsay
Turner

Customers with an account but no loan.

4. 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 $r1 \times r2$.

This set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A_1, A_2, ..., A_n) \times S(B_1, B_2, ..., B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

Suppose that we want to find the names of all customers who have a loan at the Perryridge branch. We need the information in both the *loan* relation and the *borrower* relation to do so. If we write

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

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

Figure 3.14 Result of $borrower \times loan$.

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Figure 3.15 Result of $\sigma_{branch\text{-}name = \text{“Perryridge”}} (borrower \times loan)$.

Binary Relational Operations

1. JOIN Operation

The JOIN operation, denoted by ⋈, is used to combine *related tuples* from two relations into single “longer” tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations.
The general form of a JOIN operation on two relations $R(A_1, A_2, ..., A_n) \times S(B_1, B_2, ..., B_m)$ is

$$R \bowtie_{\text{<join condition>}} S$$

The result of join is a relation Q with n+m attributes $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$.

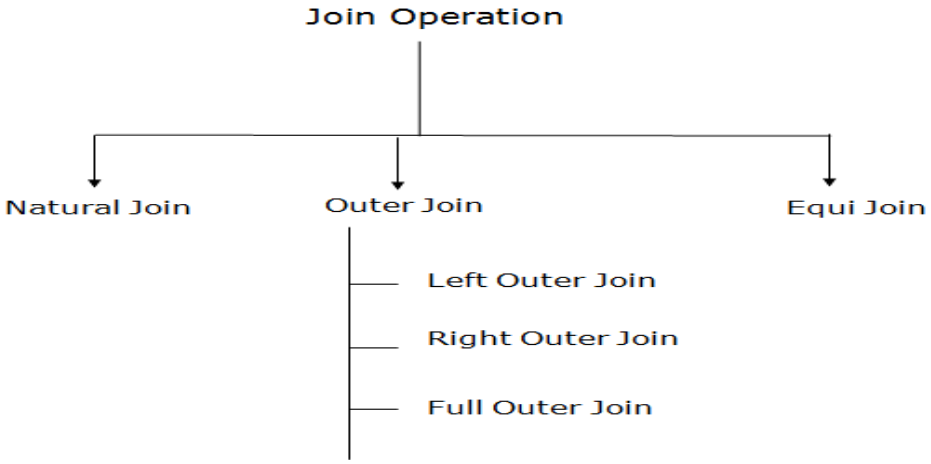
In JOIN, only combinations of tuples satisfying the join condition appear in the result, whereas in the Cartesian Product all combination of tuples are included in the result. Each tuple combination for which the join condition evaluates to TRUE is included in the result relation Q as a single combined tuple.

A general join condition is of the form

<condition> AND <condition> AND...AND <condition>

where each <condition> is of the form $A_i\ \theta\ B_j$, Ai is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. A JOIN operation with such a general join condition is called a THETA JOIN.

Types of JOIN



a. NATURAL JOIN and EQUI JOIN

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 the "join" symbol ⋈. The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.
It is also denoted by symbol '*' and also known as **INNER JOIN**.

Example:

Student

S_id	Name	Class	Age	C_id
1	Andrew	5	25	11
2	Angel	10	30	11
3	Anamika	8	35	22

Course

C_id	C_name
11	Foundation C
21	C++

Student ⋈ Course

S_id	Name	Class	Age	C_id	C_name
1	Andrew	5	25	11	Foundation C
2	Angel	10	30	11	Foundation C

Query: Find the names of all customers who have a loan at the bank, and find the amount of the loan.

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

Output:

customer-name	*loan-number*	*amount*
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Result of $\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie\ loan)$.

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN.**

b. OUTER JOIN

An inner join includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use outer joins to include all the tuples from the participating relations in the resulting relation.
There are three kinds of outer joins − left outer join, right outer join, and full outer join.

i) Left Outer Join

Left Outer Join is a type of join in which all the tuples from left relation are included and only those tuples from right relation are included which have a common value in the common attribute on which the join is being performed. It is denoted by ⟕.

Student ⟕ Course

Student

S_id	Name	Class	Age	C_type
1	Andrew	5	25	A
2	Angel	10	30	A
3	Anamika	8	35	C

Course

C_type	C_name
A	Foundation C
B	C++

Student ⟕ Course

S_id	Name	Class	Age	C_type	C_name
1	Andrew	5	25	A	Foundation C
2	Angel	10	30	A	Foundation C
3	Anamika	8	35	C	-

ii) Right Outer Join

Right Outer Join is a type of join in which all the tuples from right relation are included and only those tuples from left relation are included which have a common value in the common attribute on which the right join is being performed. It is denoted by ⟖.

Student ⟖ Course

Student

S_id	Name	Class	Age	C_type
1	Andrew	5	25	A
2	Angel	10	30	A
3	Anamika	8	35	C

Course

C_type	C_name
A	Foundation C
B	C++

Student ⟖ Course

S_id	Name	Class	Age	C_type	C_name
1	Andrew	5	25	A	Foundation C
2	Angel	10	30	A	Foundation C
-	-	-	-	B	C++

iii) Full Outer Join

Full Outer Join is a type of join in which all the tuples from the left and right relation which are having the same value on the common attribute. Also, they will have all the remaining tuples which are not common on in both the relations. It is denoted by ⟗.

Student ⟗ Course

Student

S_id	Name	Class	Age	C_type
1	Andrew	5	25	A
2	Angel	10	30	A
3	Anamika	8	35	C

Course

C_type	C_name
A	Foundation C
B	C++

Student ⟗ Course

S_id	Name	Class	Age	C_type	C_name
1	Andrew	5	25	A	Foundation C
2	Angel	10	30	A	Foundation C
3	Anamika	8	35	C	-
-	-	-	-	B	C++

2. DIVISION Operation

The **division** operation, denoted by ÷, is suited to queries that include the phrase "for all."

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

1. t is in $\Pi_{R-S}(r)$

2. For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:

a. $t_r[S] = t_s[S]$

b. $t_r[R - S] = t$

Query: Find all customers who have an account at *all* the branches located in Brooklyn.

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

Figure 3.3 The *branch* relation.

We can obtain all branches in Brooklyn by the expression

$$\Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch))$$

branch-name
Brighton
Downtown

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

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

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

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

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

Customer_name
Johnson

Extended Relational-Algebra Operations

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:

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

where E is any relational-algebra expression, and each of F_1, F_2, . . . , F_n is an arithmetic expression involving constants and attributes in the schema of E.

Query: Given relation instructor(ID, name, dept_name, salary) where salary is annual salary, Get the same information but with monthly salary.

$$\prod_{\text{ID, name, dept\_name, salary/12}} (\text{instructor})$$

2. Aggregate Functions

The aggregate operation, $\mathcal{G}$, permits the use of aggregate functions such as min, average, sum, max, etc which work on sets of values.

Aggregate functions take a collection of values and return a single value as a result.

Query: Find the total balance of all accounts.

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

Query: Find the average balance of all accounts where balance is more than 50000.

$$\mathcal{G}_{\text{avg(balance)}}(\sigma_{\text{balance>50000}}\ (\text{account}))$$

There are circumstances which involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group.

Thus, the general form of the aggregation operation $\mathcal{G}$ is as follows:

$$G_1, G_2, \ldots, G_n \mathcal{G}_{F_1(A_1),\ F_2(A_2), \ldots,\ F_m(A_m)}(E)$$

where E is any relational-algebra expression; G_1,G_2, . . .,G_n constitute a list of attributes on which to group; each F_i is an aggregate function; and each A_i is an attribute name.

Query: Find the total balance of each branch.

$$\text{bname}\ \mathcal{G}_{\text{sum(bal)}}(\text{account})$$

Example Queries in Relational Algebra

Consider the following relational schema:

EMPLOYEE(Fname, lname, ssn, bdate, address, gender, salary, super_ssn, dno)

DEPARTMENT(Dname, Dnumber, Mgr_ssn, Mgr_start_date)

PROJECT(Pname, Pnumber, Plocation, Dnum)

DEPT_LOCATIONS(Dnumber, Dlocation)

WORKS_ON(Essn, Pno, Hours)

DEPENDENT(Essn, Dependent_name, gender, bdate, Relationship)

Query1. Retrieve the name and address of all employees who work for the 'Research' department.

$$\pi_{Fname,\ Lname,\ Address}\ (\sigma_{Dname='Research'}(DEPARTMENT \bowtie_{Dnumber=Dno}(EMPLOYEE))$$

Query2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$\Pi_{Pnumber,\ Dnum,\ Lname,\ Address,\ Bdate}\ (\sigma_{\ Plocation='Stafford'}\ (PROJECT \bowtie_{Dnum=Dnumber} DEPARTMENT \bowtie_{Mgr\text{-}ssn=ssn} EMPLOYEE))$

Query3. Find the names of employees who work on *all* the projects controlled by department number 5.

$R1 \leftarrow \rho_{(Pno)}\ (\Pi_{\ Pnumber}\ (\sigma_{\ Dnum=5}\ (PROJECT))$

$R2 \leftarrow \rho_{(ssn,\ Pno)}\ (\Pi_{\ Essn,\ Pno}\ (WORKS\_ON)$

$R3 \leftarrow R2 \div R1$

$R4 \leftarrow \Pi_{\ lname,\ fname}\ (R3 \bowtie_{\ Essn=ssn} EMPLOYEE)$

Query4. List the names of managers who have at least one dependent.

$\Pi_{\ lname,\ fname}\ (\Pi_{Mgr\text{-}ssn}\ (DEPARTMENT) \cap\ \rho_{(Mgr\text{-}ssn)}\ (\Pi_{Essn}\ (DEPENDENT) \bowtie_{\ Mgr\text{-}ssn=ssn} (EMPLOYEE)))$

Query5. Retrieve the names of employees who have no dependents.

$R1 \leftarrow \rho_{(Essn)}\ \Pi_{\ ssn}\ (EMPLOYEE)$
$R2 \leftarrow \Pi_{\ Essn}\ (DEPENDENT)$

$R3 \leftarrow R1 - R2$

$R4 \leftarrow \Pi_{lname,\ fname}\ (R3 \bowtie_{Essn=ssn} EMPLOYEE)$

Query6. List the names of all employees with two or more dependents.

$R1 \leftarrow \rho_{(Essn,\ No\text{-}of\text{-}Dependents)}\ (_{Essn}\mathcal{G}_{count(Dependent\text{-}name)}\ (DEPENDENT)$
$R2 \leftarrow \sigma_{No\text{-}of\text{-}Dependents > 2}\ (R1)$
$R3 \leftarrow \Pi_{lname,\ fname}\ (R2 \bowtie_{Essn=ssn} EMPLOYEE)$

Relational Calculus

The relational calculus is another formal query language for the relational model. In relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how, or *in what order*, to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request *in a particular order* of applying the operations; thus, it can be considered as a **procedural** way of stating a query.

Any query that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the languages is *identical*. This led to the definition of the concept of a *relationally complete* language. A relational query language *L* is considered **relationally complete** if we can express in *L* any query that can be expressed in relational calculus.

A relational calculus expression may be written in two ways:
1. Tuple Calculus
2. Domain Calculus

1. Tuple Calculus

The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form:

$$\{t \mid \text{COND}(t)\}$$

where t is a tuple variable and COND(t) is a conditional (Boolean) expression involving t.

A general **expression** of the tuple relational calculus is of the form

$$\{t_1.A_j, t_2.A_k, ..., t_n.A_m \mid \text{COND}(t_1, t_2, ..., t_n, t_{n+1}, t_{n+2}, ..., t_{n+m})\}$$

where $t_1, t_2, ..., t_n, t_{n+1}, t_{n+2}, ..., t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula** of the tuple relational calculus. A formula is made up of predicate calculus **atoms**, which can be one of the following:

1. An atom of the form $R(t_i)$, where R is a relation name and t_i is a tuple variable. This atom identifies the range of the tuple variable t_i as the relation whose name is R. It evaluates to TRUE if *ti* is a tuple in the relation R, and evaluates to FALSE otherwise.

2. An atom of the form $t_i.A$ **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, and B is an attribute of the relation on which t_j ranges.

3. An atom of the form $t_i.A$ **op** c or c **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, B is an attribute of the relation on which t_j ranges, and c is a constant value.

A **formula** (Boolean condition) is made up of one or more atoms connected via the logical operators **AND**, **OR**, and **NOT.**

Query: Find all employees whose salary is above $50,000,

$$\{t \mid \text{EMPLOYEE}(t) \textbf{ AND } t.\text{Salary}>50000\}$$

The above query retrieves all attribute values for each selected EMPLOYEE tuple *t*. To retrieve only *some* of the attributes—say, the first and last name, the expression is:

$$\{t.\text{Fname}, t.\text{Lname} \mid \text{EMPLOYEE}(t) \textbf{ AND } t.\text{Salary}>50000\}$$

Query: Retrieve the birth date and address of the employee (or employees) whose name is John Smith.

$\{t.\text{Bdate}, t.\text{Address} \mid \text{EMPLOYEE}(t)$ **AND** $t.\text{Fname}=\text{‘John’}$ **AND** $t.\text{Lname}=\text{‘Smith’}\}$

The Existential and Universal Quantifiers

In tuple relational calculus, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** ($\forall$) and the **existential quantifier** ($\exists$).
A tuple variable *t* is bound if it is quantified, meaning that it appears in an ($\exists t$) or ($\forall t$) clause; otherwise, it is free.

For example, consider the following formulas:
$F1$: $d.\text{Dname}=\text{‘Research’}$
$F2$: $(\exists t)(d.\text{Dnumber}=t.\text{Dno})$
$F3$: $(\forall d)(d.\text{Mgr\_ssn}=\text{‘333445555’})$

The tuple variable *d* is free in both $F1$ and $F2$, whereas it is bound to the ($\forall$) quantifier in $F3$. Variable *t* is bound to the ($\exists$) quantifier in $F2$.

Existential Quantifier: If F is a formula, then so is $(\exists t)(F)$, where *t* is a tuple variable. The formula $(\exists t)(F)$ is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurrences of *t* in F; otherwise, $(\exists t)(F)$ is FALSE.

Universal Quantifier: If F is a formula, then so is $(\forall t)(F)$, where t is a tuple variable. The formula $(\forall t)(F)$ is TRUE if the formula F evaluates to TRUE for *every tuple* in the universe) assigned to free occurrences of t in F; otherwise, $(\forall t)(F)$ is FALSE.

Queries in Tuple Relational Calculus

Consider the following relational schema:

EMPLOYEE(Fname, lname, ssn, bdate, address, gender, salary, super_ssn, dno)

DEPARTMENT(Dname, Dnumber, Mgr_ssn, Mgr_start_date)

PROJECT(Pname, Pnumber, Plocation, Dnum)

DEPT_LOCATIONS(Dnumber, Dlocation)

WORKS_ON(Essn, Pno, Hours)

DEPENDENT(Essn, Dependent_name, gender, bdate, Relationship)

Query1. List the name and address of all employees who work for the 'Research' department.

{*t*.Fname, *t*.Lname, *t*.Address | EMPLOYEE(*t*) **AND** (∃*d*)(DEPARTMENT(*d*) **AND** *d*.Dname='Research' **AND** *d*.Dnumber=*t*.Dno)}

Query2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

{*p*.Pnumber, *p*.Dnum, *m*.Lname, *m*.Bdate, *m*.Address | PROJECT(*p*) **AND** EMPLOYEE(*m*) **AND** *p*.Plocation='Stafford' **AND** ((∃*d*)(DEPARTMENT(*d*) **AND** *p*.Dnum=*d*.Dnumber **AND** *d*.Mgr_ssn=*m*.Ssn))}

Query3. List the name of each employee who works on *some* project controlled by department number 5.

{*e*.Lname, *e*.Fname | EMPLOYEE(*e*) **AND** ((∃*x*)(∃*w*)(PROJECT(*x*) **AND** WORKS_ON(*w*) **AND** *x*.Dnum=5 **AND** *w*.Essn=*e*.Ssn **AND** *x*.Pnumber=*w*.Pno))}

Query4. List the names of employees who work on *all* the projects controlled by department number 5.

{*e*.Lname, *e*.Fname | EMPLOYEE(*e*) **AND** ((∀*x*)(**NOT**(PROJECT(*x*)) **OR NOT** (*x*.Dnum=5) **OR** ((∃*w*)(WORKS_ON(*w*) **AND** *w*.Essn=*e*.Ssn **AND** *x*.Pnumber=*w*.Pno))))}

Query5. List the names of employees who have no dependents.

{*e*.Fname, *e*.Lname | EMPLOYEE(*e*) **AND** (**NOT** (∃*d*)(DEPENDENT(*d*) **AND** *e*.Ssn=*d*.Essn))}

Safe Expressions

Whenever we use universal quantifiers, existential quantifiers, or negation of predicates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe**.
For example, the expression

$\{t \mid \textbf{NOT}\ (\text{EMPLOYEE}(t))\}$

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression:* This is the set of all values that either appear as constant values in the expression or exist in any tuple in the relations referenced in the expression.

An expression is said to be **safe** if all values in its result are from the domain of the expression.

The Domain Relational Calculus

Domain calculus differs from tuple calculus in the *type of variables* used in formulas: Rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree *n* for a query result, we must have *n* of these **domain variables**—one for each attribute.An expression of the domain calculus is of the form

$\{x_1, x_2, ..., x_n \mid \text{COND}(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m})\}$

where $x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m}$ are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

A formula is made up of **atoms**. The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form $\text{R}(x_1, x_2, ..., x_j)$, where *R* is the name of a relation of degree *j* and each x_i, $1 \leq i \leq j$, is a domain variable.

2. An atom of the form x_i **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, and x_i and x_j are domain variables.

3. An atom of the form x_i **op** *c* or *c* **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, x_i and x_j are domain variables, and *c* is a constant value.

Queries in Domain Relational Calculus

Consider the following relational schema:

EMPLOYEE(Fname, lname, ssn, bdate, address, gender, salary, super_ssn, dno)

DEPARTMENT(Dname, Dnumber, Mgr_ssn, Mgr_start_date)

PROJECT(Pname, Pnumber, Plocation, Dnum)

DEPT_LOCATIONS(Dnumber, Dlocation)

WORKS_ON(Essn, Pno, Hours)

DEPENDENT(Essn, Dependent_name, gender, bdate, Relationship)

Query1. List the birth date and address of the employee whose name is 'John Smith'.

$$\{u, v \mid (\exists q)\ (\exists r)\ (\exists s)\ (\exists t)\ (\exists w)\ (\exists x)\ (\exists y)\ (\exists z)\ (\text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ q=\text{'John'}\ \textbf{AND}\ r=\text{'B'}\ \textbf{AND}\ s=\text{'Smith'})\}$$

Note: We can quantify only those variables that actually appearing in a condition.

Query2. Retrieve the name and address of all employees who work for the 'Research' department.

$$\{q, s, v \mid (\exists z)\ (\exists l)\ (\exists m)\ (\text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ \text{DEPARTMENT}(lmno)\ \textbf{AND}\ l=\text{'Research'}\ \textbf{AND}\ m=z)\}$$

Query3. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

$$\{i, k, s, u, v \mid (\exists j)(\exists m)(\exists n)(\exists t)(\text{PROJECT}(hijk)\ \textbf{AND}\ \text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ \text{DEPARTMENT}(lmno)\ \textbf{AND}\ k=m\ \textbf{AND}\ n=t\ \textbf{AND}\ j=\text{'Stafford'})\}$$

Query4. List the names of employees who have no dependents.

$$\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ (\textbf{NOT}(\exists l)(\text{DEPENDENT}(lmnop)\ \textbf{AND}\ t=l)))\}$$

Query5. List the names of managers who have at least one dependent.

$$\{s, q \mid (\exists t)(\exists j)(\exists l)(\text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ \text{DEPARTMENT}(hijk)\ \textbf{AND}\ \text{DEPENDENT}(lmnop)\ \textbf{AND}\ t=j\ \textbf{AND}\ l=t)\}$$