Introduction to Relational Model

Codd proposed the relational model in 1970. At this time, most database systems were based on one of two older data models (the hierarchical model and the network model). The relational model created the revolution in the database field and largely supplanted these earlier models. Today, the relational model is the dominant data model and the foundation for the leading DBMS products, including IBM's DB2,Informix,Oracle, Sybase, MS Access, and SQL Server.

In the relational model, a database is a collection of one or more *relations*, where each relation is a table with rows and columns. The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

Terminology

Relation: A relation is a table with columns and rows.

Attribute: An attribute is a named column of a relation.

Domain: A domain is the set of allowable values for one or more attributes.

Tuple: A tuple is arrow of a relation.

Degree: The degree of a relation is the number of attributes it contains.

Cardinality: The cardinality of a relation is the number of tuples it contains.

Relational Database: A collection of normalized relations with distinct relation names.

A relation consists of a **relation schema** and a **relation instance**. The ***schema*** specifies the relation's name, the name of each field and the domain of each field.

Ex : for Schema:

Students(*sid:*integer,*sname:*string,*login:*string,*age:*integer,*gpa:*real)

An ***instance*** is the snapshot of the database. An instance is a collection of information stored in the database at particular instant of time.

Ex: for instance:

FIELDS(ATTRIBUTES,COLUMNS)

TUPLES

sid	***sname***	***login***	***age***	***Gpa***
567	Adithya	adithya@mrits.co.nr	19	6.2
574	Ayush	ayush@mrits.co.nr	20	6.5
597	Rajasekhar	rajasekhar@mrits.co.nr	18	5.5
5C1	Sunil	sunil@mrits.co.nr	21	6.8

Integrity Constraints over Relations

An **integrity constraint** (**IC**) is a condition specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specifies on the database schema, it is a **legal** instance. A DBMS permits only legal instances to be stored in the database.

Many kinds of integrity constraints can be specified in the relational model:

1. **Domain Constraints**
   A relation schema specifies the domain of each field in the relation instance. These **domain constraints** in the schema specify the condition that each instance of the relation has to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the type of that field.
2. **Key Constraints**
   A **Key Constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple.

Super Key:

An attribute, or set of attributes, that uniquely identifies a tuple with in a relation.
However, a super key may contain additional attributes that are not necessary for a unique identification.

Ex: The *customer_id* of the relation *customer* is sufficient to distinguish one tuple from other .Thus, *customer_id*is a super key. Similarly, the combination of *customer_id* and *customer_name* is a super key for the relation *customer*. Here the *customer_name* is not a super key, because several people may have the same name.

Weareoftenin terestedinsuperkeysforwhichnopropersubsetisasuperkey.Suchminimalsuperkeys are called **candidate keys.**

Candidate Key:

A superkey such that no proper subset is a super key with in the relation.

There are two parts of the candidate key definition:

i. Two distinct tuples in a legal instance cannot have identical values in all the fields of a key.
ii. No subset of the set of fields in a candidate key is a unique identifier for a tuple.

A relation may have several candidate keys.

Ex: The combination of *customer_name* and *customer_street* is sufficient to distinguish the members of the *customer* relation. Then both, {*customer_id*} and {*customer_name,customer_street*} are candidate keys. Although *customer_id* and *customer_name* together can

Distinguish *customer* tuples, their combination does not form a candidate key ,since the *customer_id* alone is a candidate key.

Primary Key:

The candidate key that is selected to identify tuples uniquely within the relation. Out of all the available candidate keys, a database designer can identify a *primary key*. The candidate keys that are not selected as the *primary key* are called as ***alternate keys.***

Features of the primary key:

i. Primary key will not allow duplicate values.
ii. Primary key will not allow null values.
iii. Only one primary key is allowed per table.

Ex: For the *student* relation, we can choose *student_id* as the primary key.

Foreign Key:

Foreign keys represent the relationships between tables. A foreign key is a column (or a group of columns) whose values are derived from the *primary key* of some other table.

The table in which *foreign key* is defined is called a **Foreign table** or **Details table**. The table that defines the *primary key* and is referenced by the *foreign key* is called the **Primary table** or **Master table**.

Features of foreign key:

i. Records cannot be **inserted** into a **detail table** if corresponding records in the master table do not exist.
ii. Records of the **master table** cannot be **deleted** or **updated** if corresponding records in the detail table actually exist.

Handling Foreign Key violations:

1. What should we do if an Enrolled row is inserted, with a studid column value that does not appear in any row of the Students table?

In this case, the INSERT command is simply rejected.

2. What should we do if a Students row is deleted?

The options are:

• Delete all Enrolled rows that refer to the deleted Students row.
• Disallow the deletion of the Students row if an Enrolled row refers to it.
• Set the studid column to the sid of some (existing) 'default' student, for every Enrolled row that refers to the deleted Students row.

3. What should we do if the primary key value of a Students row is updated?

The options here are similar to the previous case.

General Constraints:

Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model. Sometimes, however, it is necessary to specify more general constraints.

For example, we may require that student ages be within a certain range of values. Giving such an IC, the DBMS rejects inserts and updates that violate the constraint.

Current database systems support such general constraints in the form of *table constraints* and *assertions.* Table constraints are associated with a single table and checked whenever that table is modified. In contrast, assertions involve several tables and are checked whenever any of the tables is modified.

Ex for table constraint, which ensures always the *salary of an employee is above 1000*:

CREATE TABLE employee(*eid* integer, *ename* varchar2(20), *salary* real, CHECK(salary>1000));

Ex for assertion, which enforce a constraint that the *number of boats plus the number of sailors should be less than 100.*

CREATE ASSERTION small Club CHECK ((SELECT COUNT(S.sid) FROM Sailors S) + (SELECT COUNT(B.bid) FROM Boats B) < 100);

- SQL allows a constraint to be in DEFERRED or IMMEDIATE mode.
- A constraint in deferred mode is checked at commit time.

Querying Relational Data

Relational Database Query

A *relational database query* (query, for short) is a question about the data, and the answer consists of a new relation containing the result.

For example, we might want to find all students younger than 18 or all students enrolled in DBMS.

Query Languages

A *query language* is a specialized language for writing queries. This is the language in which a user requests information from the database.

Query languages can be categorized as either *procedural* or *non-procedural*. In a **procedural language,** the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a **non-procedural language,** the user describes the desired information without giving a specific procedure for obtaining that information.

SQL is the most popular language for a relational DBMS. Consider the instance of the student's relation as shown in figure given in page#1 of this unit. We can retrieve the rows corresponding to students who are younger than 20 with the following query:

SELECT * FROM Students S WHERE S.age<20;

In addition to selecting a subset of rows, a query can extract a subset of the columns of each selected row. We can compute the names and logins of students who are younger than 20 with the following query:

SELECT S.sname, S.login FROM Students S WHERE S.age<20;

There are a number of "pure" languages. The *relational algebra* is procedural, where as the *tuple relational calculus* and *domain relational calculus* are non-procedural. These query languages are formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

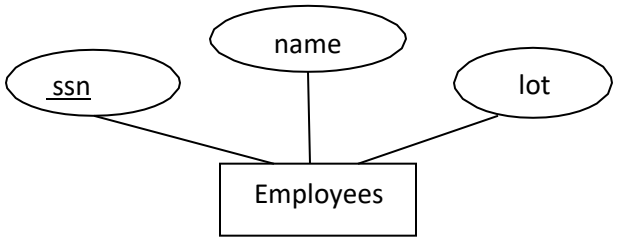
Logical Database Design: ER to Relational

Given an ER diagram describing a database, a standard approach is taken to generate a relational database schema that closely approximates the ER design.

Entity Sets to Tables

An entity set is mapped to a relation in a straight forward way: Each attribute of the entity set becomes a column of the table.

Consider the Employees entity set with attributes *ssn*, *name*, and *lot* shown in figure:



The following SQL statement captures the preceding information, including domain constraints and key information:

CREATE TABLE Employees (ssn INTEGER, name CHAR(20), lot INTEGER, PRIMARY KEY (ssn));

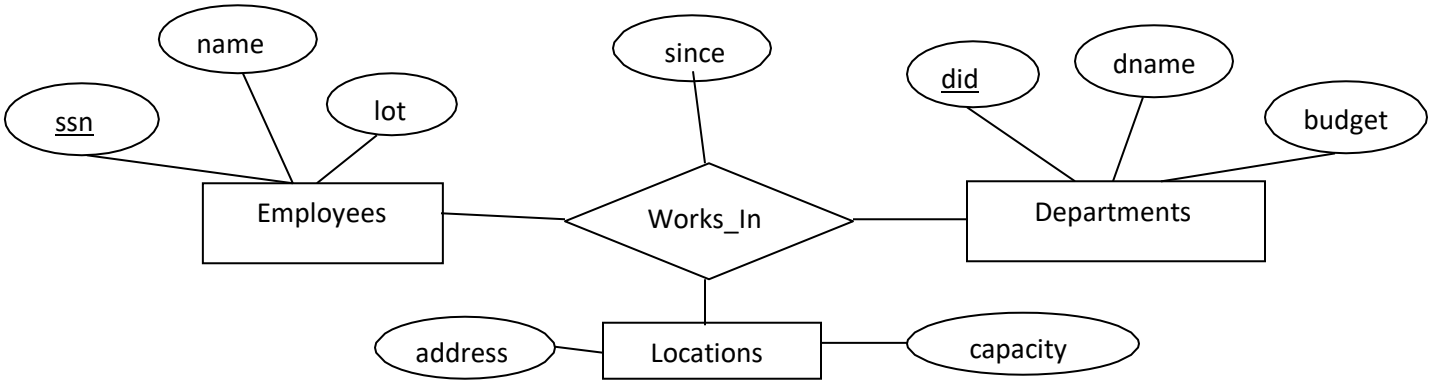
Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model. To represent a relationship without key and participation constraints, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship. Thus, the attributes of the relation include:

i. The primary key attributes of each participating entity set, as foreign key fields.

ii. The descriptive attributes of the relationship set.

Consider the following Works_In relationship set shown in figure:

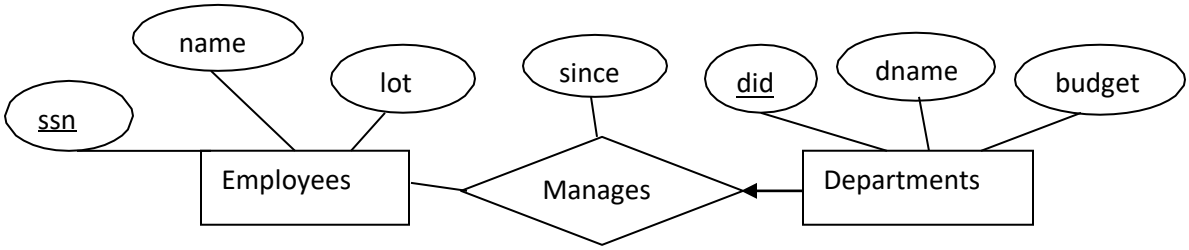


All the available information about the Works_In table is captured by the following SQL statement:

CREATE TABLE Works_In (ssn INTEGER, did INTEGER, address VARCHAR2(20), since DATE, PRIMARY KEY (ssn, did, address), FOREIGN KEY (ssn) REFERENCES Employees, FOREIGN KEY(address) REFERENCES Locations, FOREIGNKEY(did) REFERENCES Departments);

Translating Relationship Sets with Key Constraints

Consider the (one-to-many) relationship set Manages as shown in figure:



The table corresponding to *Manages* has the attributes *ssn, did, since*. However, because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value. A consequence of this observation is that *did* is itself a key for *Manages*, indeed, the set *did, ssn* is not a key.

The Manages relation can be defined using the following SQL statement:

CREATE TABLE Manages (ssn INTEGER, did INTEGER, since DATE, PRIMARY KEY (did), FOREIGN KEY (ssn) REFERENCES Employees, FOREIGN KEY (did) REFERENCES Departments);

A second approach to translate a relationship set with key constraints is often superior because it avoids creating a separate table for the relationship set.

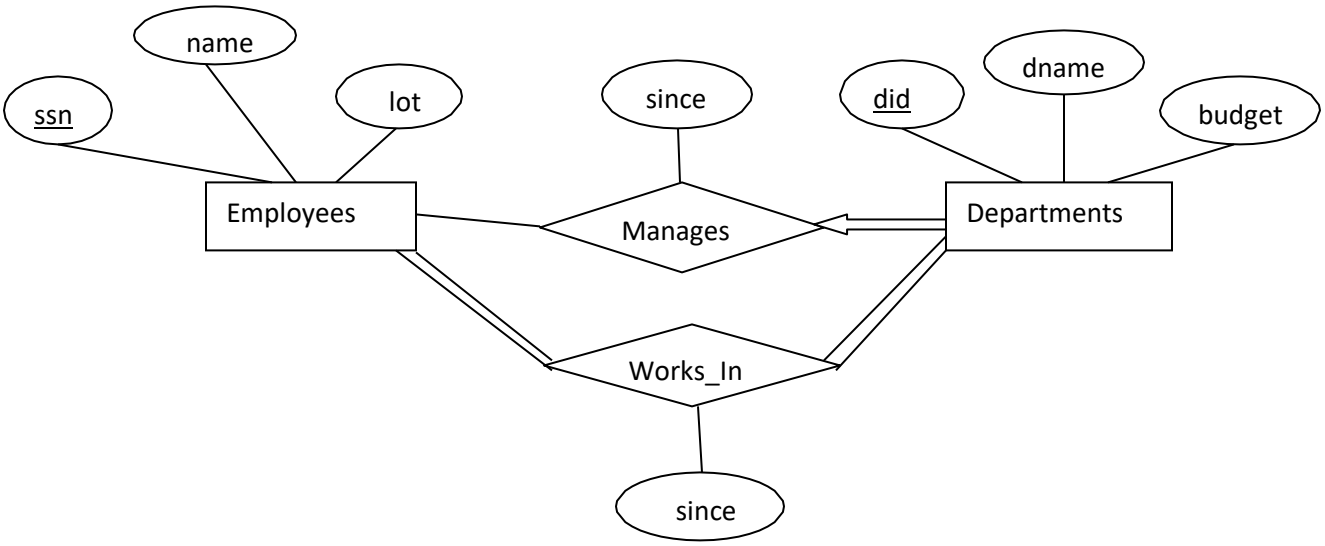
In the Manages example, because a department has at most one manager, we can add the *since* attribute to the Departments tuple. This approach eliminates the need for a separate Manages relation. The only drawback to this approach is that space could be wasted if several departments have no managers.

The following SQL statement defines the Dept_Mgr relation that captures the information in both Departments and Manages, illustrates the second approach:

CREATE TABLE Dept_Mgr (did INTEGER, dname CHAR(20), budget REAL, ssn INTEGER, since DATE, PRIMARY KEY(did), FOREIGN KEY (ssn) REFERENCES Employees);

Translating Relationship Sets with Participation Constraints

Consider the ER diagram as shown below, which show the two relationship sets, Manages and Workd_In:



Every department is required to have manager, due to the participation constraint (total participation), and atmost one manager, due to the key constraint(one-to-many).

The following SQL statement reflects the second translation approach:

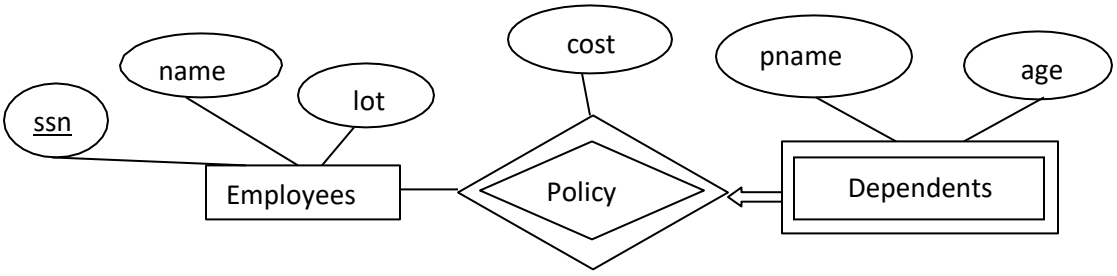
CREATE TABLE Dept_Mgr (did INTEGER, dname CHAR(20), budget REAL, ssn INTEGER NOT NULL, since DATE, PRIMARY KEY (did), FOREIGN KEY (ssn) REFERENCES Employees ON DELETE NO ACTION);

It also captures the participation constraint that every department must have a manager: Because *ssn* cannot take on *null* values, each tuple of Dept_Mgr identifies a tuple in Employees. The NO ACTION specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while It is pointed by a Dept_Mgr tuple.

Translating Weak Entity Sets

A weak entity set always participation in a one-to-many binary relationship and has a key constraint and total participation. The second translation approach is ideal in this case, but we must take into account that the weak entity has only a partial key. Also, when an owner entity is deleted, we want all owned weak entities to be deleted.

Consider the Dependents weak entity set shown in figure with partial key *pname.*



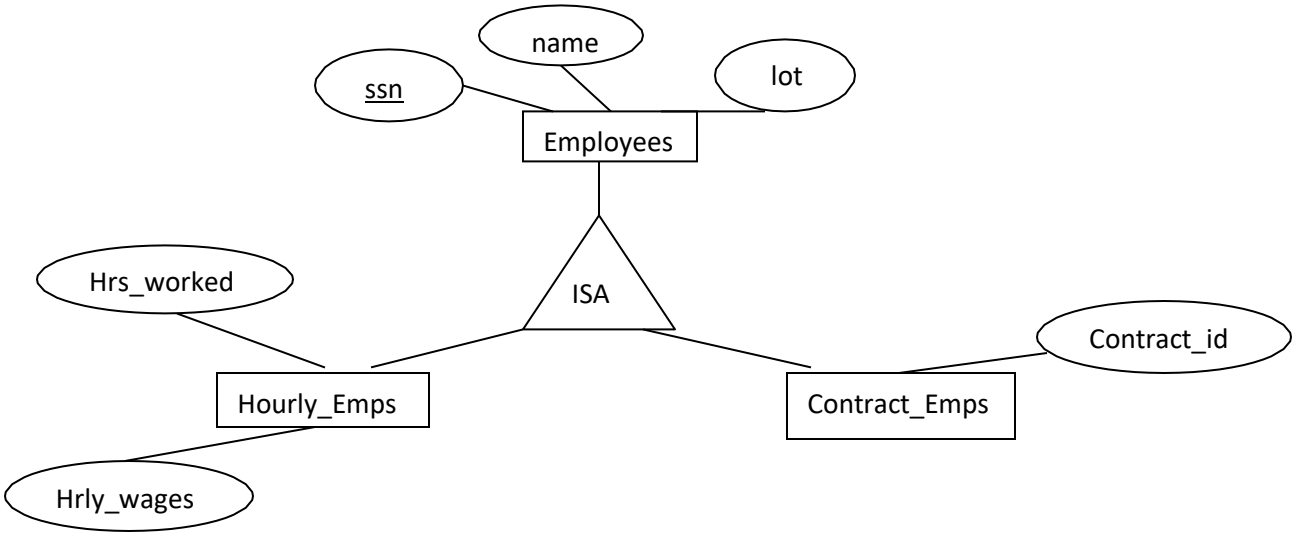
A Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity, and the Dependents entity must be deleted if the owning Employees entity is deleted.

We can capture Dept_Policy relation information by the following SQL statement:

CREATE TABLE Dep_Policy (pname CHAR(20), age REAL, cost REAL, ssn INTEGER, PRIMARY KEY (pname,ssn), FOREIGN KEY(ssn) REFERENCES Employees ON DELETE CASCADE);

Translating Class Hierarchies

We present the two basic approaches to handle IS A hierarchies in the ER diagram shown below:



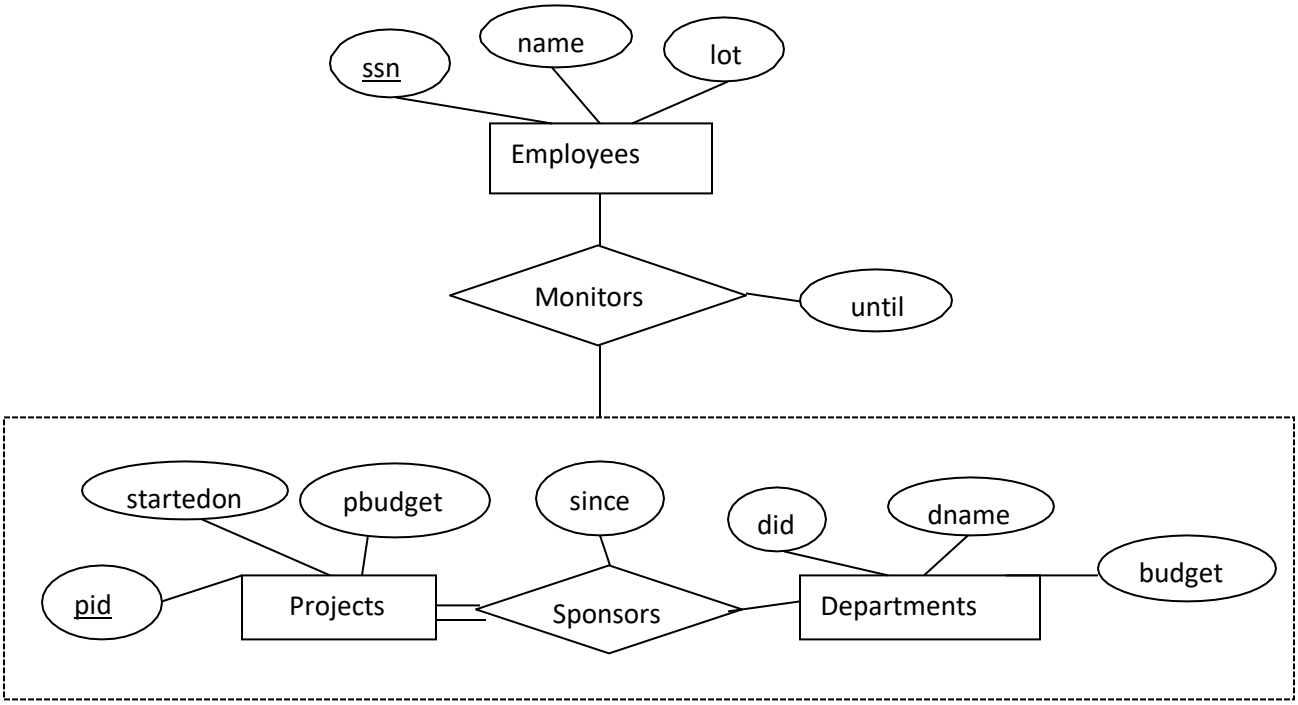
1. We can map each of the entity sets Employees, Hourly_Emps, and Contract_Emps to a distinct relation. The relation for Hourly_Emps includes the *hourly_wages* and *hours_worked* attributes of Hourly_Emps. It also contains the key attributes of the super class (ssn, in this example), which serves as the primary key for Hourly_Emps, as well as a foreign key referencing the super class (Employees). Note that if a super class tuple is deleted, the delete must be cascaded to Hourly_Emps.

2. Alternatively, we can create just two relations, corresponding to Hourly_Emps and Contract_Emps. The relation for Hourly_Emps includes all the attributes of Hourly_Emps as well as all the attributes of Employees(i.e .*ssn, name, lot, hourly_wages, hours_worked*).

The first approach is general and always applicable. The second approach is not applicable if we have employees who are neither hourly employees nor contract employees, since there is no way to store such employees. A query that needs to examine all employees must now examine two relations.

Translating ER Diagrams with Aggregation

Consider the ER diagram shown in figure:



The Employees, Projects and Departments entity sets and the Sponsos relationship set are mapped as described in previous section. For the Monitors relationship set, we create a relation with the following attributes: the key attributes of Employees (*ssn*), the key attributes of Sponsors (*did, pid*), and the descriptive attributes of Monitors(*until*).

Introduction to Views

A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **View definition.**

Creating a View

The format of the CREATE VIEW statement is:

CREATE VIEW ViewName [(col1,col2,…)] AS SELECT Statement;

Consider the Students and Enrolled relations. Suppose we are interested in finding the names and student ids of students who got a grade of B in some course, together with the course identifier. We can define the view for this as follows:

CREATE VIEW B_Students (name, sid, course) AS SELECT S.sname, S.sid, E.cid FROM Students S, Enrolled E WHERE S.sid=E.studid AND E.grade='B';

Removing a View

A view is removed from the database with the DROP VIEW statement:

DROP VIEW ViewName;

We could remove the B_Students view using the following statement:

DROP VIEW B_Students;

Updatable Views

The SQL allows updates to be specified only on views that are defined on a single base table using just selection and projection, with no use of aggregate operators. Such views are called **updatable views.**

A view is updatable if and only if:

- DISTINCT is not specified.
- The FROM clause specifies only one table.
- There is no GROUP BY or HAVING clause in the defining query.
- View must not violate the integrity constraints of the basetable.

Advantages of Views

- Data Independence
- Improved Security
- Reduced Complexity

Disadvantages of Views

- Update restriction
- Performance

Relational Algebra

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts one or more relational instances as arguments and returns a relation instance as the result. The relational algebra is *procedural.*

Consider the following instances to illustrate the relational algebra operators:

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Fig: Instance S1 Sailors

sid	sname	rating	age
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	35.0
58	Rusty	10	35.0

Fig: Instance S2 of Sailors

sid	bid	Day
22	101	10/10/96
58	103	11/12/96

Fig: Instance R1 of Reserves

The various **operators** that form the relational algebraic query are:

1) Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation.

In general, the selection operator σ specifies the tuples to get through a *selection condition.* Here the selection condition is a Boolean combination of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2,* where op is one of the relational operators <,<=, =,≠, >=, or >.

Consider the instance S2 of the Sailors relation shown in figure, we can find the Sailors with rating above 8 by the following expression:

$\sigma_{rating> 8}(S2)$

This evaluates to the relation shown below:

Sid	sname	rating	Age
28	Yuppy	9	35.0
58	Rusty	10	35.0

The *projection* operator π allows us to extract columns from a relation. For example, we can find out all sailors names and ratings by the following expression:

$\Pi_{sname,rating}(S2)$

This evaluates to the relation shown below:

sname	rating
Yuppy	9
Lubber	8
Guppy	5
Rusty	10

Similarly, we can find the names and ratings of sailors with rating above 8 by the following expression:

$$\Pi_{sname,rating}(\sigma_{rating>8}(S2))$$

This evaluates to the relation shown below:

sname	rating
Yuppy	9
Rusty	10

2) Set Operations

The following standard operations on sets are available in relational algebra: *union*(U), *intersection*(∩), *set-difference*(—), and *cross-product* (X).

Union:

RUS returns a relation instance containing all tuples that occur in either relation instance R or relation instance S (or both).

R and S must be *union-compatible,* and the schema of the result is identical to the schema of R.

Two relation instances are said to be ***union-compatible*** if the following conditions hold:

- They have the same number of fields, and
- Corresponding fields, taken in order from left to right, have the same *domains*

Ex :The *union* of S1and S2(S1 U S2) is shown below:

Sid	*Sname*	*rating*	*Age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	Yuppy	9	35.0
44	Guppy	5	35.0

Intersection:

R∩S returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be *union-compatible* ,and the scheme of the result is same as the schema of R.

Ex: The *intersection* of S1and S2(S1 ∩S2) is shown below:

Sid	Sname	rating	age
31	Lubber	8	55.5
58	Rusty	10	35.0

Set-difference:

R—S returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be *union-compatible,* and the schema of the result is identical to the schema of R.

Ex: The *Set-difference* of S1and S2 (S1—S2) is shown below:

Sid	Sname	rating	age
22	Dustin	7	45.0

Cross-product:

RX S returns a relation instance whose schema contains all the fields of R followed by all the fields of S. The cross product operation is sometimes called as ***Cartesian product.***

Ex: The *Cross-product* of S1and R1(S1XR1) is shown below:

(sid)	Sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

3) Renaming

The result of the relational algebra expression includes the field names in such a way that naming conflicts can arise in some cases. For example, in S1 X R1. Hence we have to rename the fields or rename the relation. Relation algebra provides ***renaming*** operator ρ for this purpose.

The expression ρ (R (F), E) takes a relational algebra expression E and returns an instance of a relation R. R contains the same tuples as the result of E and has the same schema as E, but some fields are renamed. F is the list of fields renamed and is in the form *oldname —>newname* or *position—>newname.*

For example, the expression ρ (*C* (*1 —>sid1, 5 —> sid2*), S1 X R1) returns a relation with the following schema:

C(*sid1:*integer, *sname:* string, *rating:*integer, *age:*real, *sid2:*integer,*bid:*integer,*day:* date)

4) Joins

The *join* operation is used to combine the information from two or more relations. *Join* can be defined as a cross-product followed by selection and projection. There are several variants of join operation:

Conditional Join:

The most general version of the join operation accepts a *join condition c* and a pair of relational instances as arguments and returns a relation instance. The operation is defined as follows:

$$R \bowtie_C S = \sigma_C(RXS)$$

Thus $\bowtie$ is defined to be a cross-product followed by a selection.

For example, the result of $S1 \bowtie_{S1.sid<R1.sid} R1$ is shown

below:

(sid)	Sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Equijoin:

A common special case of the join operation $R \bowtie S$ is when the *join condition* consists solely of equalities of the form *R.name1 = S.name2*, that is, equalities between two fields in R and S. The join operation with such equality condition is called ***equijoin.***

For example, the result of $S1 \bowtie_{R.sid=S.sid} R1$ is shown below:

sid	Sname	rating	age	bid	day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Note that the fields in the *equijoin condition* appears only once in the resultant instance.

Natural Join:

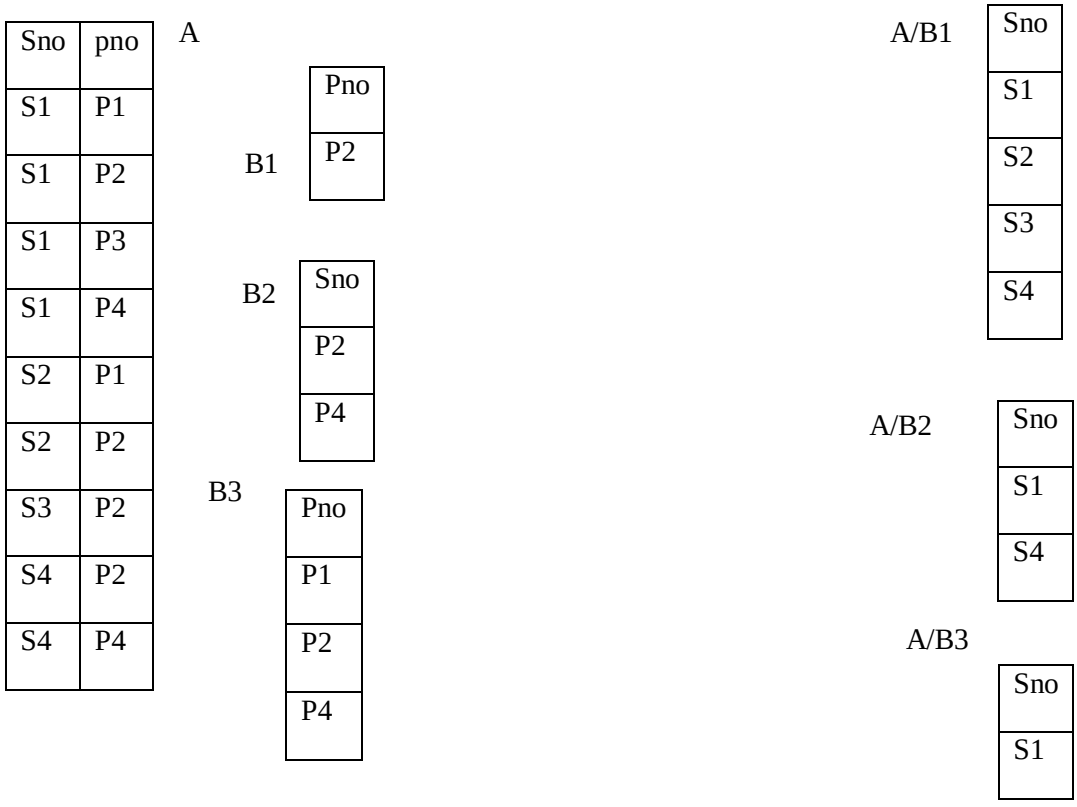
A further special case of the join operation $R \bowtie S$ is an equijoin in which equalities are specified on *all common* fields of R and S. In this case, we can simply omit the join condition. By default, the equality condition is employed on all common fields.

We call this as *natural join* and can simply be denoted as $S1 \bowtie R1$. The result of this expression is same as above, since the only common field is *sid.*

5) Division

We discuss division through an example. Consider two relation instances A and B in which A has two fields x and y and B has just one filed y, with the same domain as in A. We define the *division* operation A/B as the set of all x values such that for *every* value in B, there is a tuple <x,y> in A.

Division is illustrated in figure below. Consider the relation A listing the parts(pid) supplied by suppliers(sid) and the relation B listing the parts(pid). A/B_i computes suppliers who supply *all* parts listed in relation instance B_i.



Relational Calculus

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which Is procedural, the calculus is non-procedural or declarative.

There are two variants of the calculus:

1) Tuple Relational Calculus(TRC): Variables in TRC take tuples as the values.

2) Domain Relational Calculus(DRC): Variables in DRC range over field values.

1) Tuple Relational Calculus(TRC)

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. A TRC query has the form { T / p(T) }, where T is a tuple variable and p(T) denotes a *formula* that describes T.

Syntax of TRC Queries:

- A nonprocedural query language, where each query is of the form
  $\{t \mid P(t)\}$
- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus

Let *Rel* be a relation, *R* and *S* be tuple variables, *a* be an attribute of *R*, and *b* be an attribute of *S*. Let op denotes a relational operator. An **atomic formula** is one of the following:

- *R*∈*Rel*
- *R.a* op *S.b*
- *R.a* op *constant*, or *constant* op *R.a*

A **formula** is one of the following:

- Any atomic formula
- ¬p, pΛq, pVq, orp =>q
- ∋R(p(R)),where R is a tuple variable
- ∀R(p(R)),where R is a tuple variable

Where p and q are themselves are formulas and p(R) denotes a formula with the variable R.

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<, \leq, =, \neq, >, \geq$)
3. Set of connectives: and ($\wedge$), or (v), not ($\neg$)
4. Implication ($\Rightarrow$): x $\Rightarrow$ y, if x if true, then y is true
   $x \Rightarrow y \equiv \neg x \text{ v } y$
5. Set of quantifiers:
   - $\exists t \in r\ (Q(t)) \equiv$"there exists" a tuple in t in relation r such that predicate $Q(t)$ is true
   - $\forall t \in r\ (Q(t)) \equiv Q$ is true "for all" tuples t in relation r

Example Relations:

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

1. Find the loan-number, branch-name, and amount for loans of over $1200
   $\{t \mid t \in loan \wedge t\ [amount] > 1200\}$
2. Find the loan number for each loan of an amount greater than $1200
   $\{t \mid \exists s \in$ loan $(t[loan\text{-}number] = s[loan\text{-}number] \wedge s\ [amount] > 1200)\}$
3. Find the names of all customers having a loan, an account, or both at the bank
   $\{t \mid \exists s \in borrower(\ t[customer\text{-}name] = s[customer\text{-}name]) \vee \exists u \in depositor(\ t[customer\text{-}name] = u[customer\text{-}name])$
4. Find the names of all customers who have a loan and an accountat the bank
   $\{t \mid \exists s \in borrower(\ t[customer\text{-}name] = s[customer\text{-}name]) \wedge \exists u \in depositor(\ t[customer\text{-}name] = u[customer\text{-}name])$
5. Find the names of all customers having a loan at the Perryridge branch
   $\{t \mid \exists s \in borrower(t[customer\text{-}name] = s[customer\text{-}name] \wedge \exists u \in loan(u[branch\text{-}name] =$ "Perryridge" $\wedge u[loan\text{-}number] = s[loan\text{-}number]))\}$
6. Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank
   $\{t \mid \exists s \in borrower(\ t[customer\text{-}name] = s[$customer-name$] \wedge \exists u \in loan(u[branch\text{-}name] =$ "Perryridge" $\wedge u[loan\text{-}number] = s[$loan-$number])) \wedge$ **not** $\exists v \in depositor\ (v[customer\text{-}name] = t[$customer-name$])\ \}$

2) Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute. ADRC query has the form { <x1, x2, …, xn> | p(<x1, x2, …, xn>), where xi is either a *domain variable* or a constant and p(<x1, x2, …,xn>) denotes a **DRC formula**.

Syntax of DRC Queries:

Let *Rel* be a relation, X and Y be domain variables, xi, $1 \le i \le n$, be an attribute of *Rel*.
Let op denotes a relational operator. An **atomic formula** is one of the following:

- <x1,x2, …,xn> Є *Rel*
- *X* op *Y*
- *X* op *constant,* or *constant* op *X*

A **formula** is one of the following:

- Any atomic formula
- ¬p, pΛq, pVq, or p =>q
- ∋X(p(X)),where X is a domain variable
- ∀X(p(X)),where X is a domain variable

Example Queries:

1. Find the *loan-number, branch-name,* and *amount* for loans of over $1200
   $\{<l, b, a> \mid <l, b, a> \in loan \wedge a > 1200\}$
2. Find the names of all customers who have a loan of over $1200
   $\{<c> \mid \exists l, b, a\ (<c, l> \in borrower \wedge <l, b, a> \in loan \wedge a > 1200)\}$
3. Find the names of all customers who have a loan from the Perryridge branch and the loan amount:
   $\{<c, a> \mid \exists l\ (<c, l> \in borrower \wedge \exists b(<l, b, a> \in loan \wedge b = \text{“Perryridge”}))\}$
   or
   $\{<c, a> \mid \exists l\ (<c, l> \in borrower \wedge <l,$ “Perryridge”, $a> \in loan)\}$
4. Find the names of all customers having a loan, an account, or both at the Perryridge branch:
   $\{<c> \mid \exists l\ (\{<c, l> \in borrower \wedge \exists b,a(<l, b, a> \in loan \wedge b = \text{“Perryridge”})) \vee \exists a(<c, a> \in depositor \wedge \exists b,n(<a, b, n> \in account \wedge b = \text{“Perryridge”}))\}$
5. Find the names of all customers who have an account at all branches located in Brooklyn:
   $\{<c> \mid \exists\ s, n\ (<c, s, n> \in \text{customer}) \wedge \forall x,y,z(<x, y, z> \in branch \wedge y = \text{“Brooklyn”}) \Rightarrow \exists a,b(<x, y, z> \in account \wedge <c,a> \in depositor)\}$