Defining a Relation Schema in SQL

Data types

- **VARCHAR** is variable length, while **CHAR** is fixed length.
- CHAR is a fixed length string data type, so any remaining space in the field is padded with blanks. CHAR takes up 1 byte per character. So, a CHAR(100) field (or variable) takes up 100 bytes on disk, regardless of the string it holds.
- VARCHAR is a variable length string data type, so it holds only the characters you assign to it. VARCHAR takes up 1 byte per character, + 2 bytes to hold length information. For example, if you set a VARCHAR(100) data type = 'Jen', then it would take up 3 bytes (for J, E, and N) plus 2 bytes, or 5 bytes in all.
- You can see how the use of VARCHAR in most cases is preferred, to save space.

Defining a Relation Schema in SQL

Data types

BOOLEAN : Zero is considered as false, nonzero values are considered as true.

INT / INTEGER: A medium integer.

SHORTINT : Smaller than int

FLOAT / REAL : for typical floating point value

DOUBLE(size, d) : A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter

Defining a Relation Schema in SQL

Simple Table Declarations

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30),
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE
    );
```

Figure 6.16: Declaring the relation schema for the `MovieStar` relation

6.6.3 Modifying Relation Schemas

We can delete a relation R by the SQL statement:

```
DROP TABLE R;
```

Relation R is no longer part of the database schema, and we can no longer access any of its tuples.

Defining a Relation Schema in SQL

Alter command

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

```
ALTER TABLE MovieStar DROP birthdate;
```

Defining a Relation Schema in SQL

Default Values

In general, any place we declare an attribute and its data type, we may add the keyword DEFAULT and an appropriate value.

```
4)      gender CHAR(1) DEFAULT '?',
5)      birthdate DATE DEFAULT DATE '0000-00-00'
```

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

Set operations on Relations

- $R \cup S$, the *union* of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.
- $R \cap S$, the *intersection* of R and S, is the set of elements that are in both R and S.
- $R - S$, the *difference* of R and S, is the set of elements that are in R but not in S. Note that $R - S$ is different from $S - R$; the latter is the set of elements that are in S but not in R.

Set operations on Relations

When we apply these operations to relations, we need to put some conditions on R and S:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.

2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

Set operations on Relations

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation R

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Relation S

Figure 5.2: Two relations

Set operations on Relations

union $R \cup S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Note that the two tuples for Carrie Fisher from the two relations appear only once in the result.

Set operations on Relations

The intersection $R \cap S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Now, only the Carrie Fisher tuple appears, because only it is in both relations.

The difference $R - S$ is

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

That is, the Fisher and Hamill tuples appear in R and thus are candidates for $R - S$. However, the Fisher tuple also appears in S and so is not in $R - S$. □

Database Modifications

1. Insert tuples into a relation.
2. Delete certain tuples from a relation.
3. Update values of certain components of certain existing tuples.

We refer to these three types of operations collectively as *modifications*.

Database Modifications

The basic form of insertion statement consists of:

1. The keywords `INSERT INTO`,
2. The name of a relation R,
3. A parenthesized list of attributes of the relation R,
4. The keyword `VALUES`, and
5. A tuple expression, that is, a parenthesized list of concrete values, one for each attribute in the list (3).

That is, the basic insertion form is

$$\texttt{INSERT INTO } R(A_1, \ldots, A_n) \texttt{ VALUES } (v_1, \ldots, v_n);$$

A tuple is created using the value v_i for attribute A_i, for $i = 1, 2, \ldots, n$. If the list of attributes does not include all attributes of the relation R, then the tuple created has default values for all missing attributes. The most common default value is `NULL`, the null value, but there are other options to be discussed in Section 6.6.4.

Database Modifications

The basic form of insertion statement consists of:

1. The keywords `INSERT INTO`,
2. The name of a relation R,
3. A parenthesized list of attributes of the relation R,
4. The keyword `VALUES`, and
5. A tuple expression, that is, a parenthesized list of concrete values, one for each attribute in the list (3).

That is, the basic insertion form is

$$\texttt{INSERT INTO } R(A_1, \ldots, A_n) \texttt{ VALUES } (v_1, \ldots, v_n);$$

A tuple is created using the value v_i for attribute A_i, for $i = 1, 2, \ldots, n$. If the list of attributes does not include all attributes of the relation R, then the tuple created has default values for all missing attributes. The most common default value is `NULL`, the null value, but there are other options to be discussed in Section 6.6.4.

Database Modifications

```
1) INSERT INTO StarsIn(movieTitle, movieYear, starName)
2) VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

```
INSERT INTO StarsIn
VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

Database Modifications

Example 6.35 : Suppose we want to add to the relation

```
Studio(name, address, presC#)
```

all movie studios that are mentioned in the relation

```
Movie(title, year, length, inColor, studioName, producerC#)
```

but do not appear in Studio. Since there is no way to determine an address or a president for such a studio, we shall have to be content with value NULL for attributes address and presC# in the inserted Studio tuples. A way to make this insertion is shown in Fig. 6.15.

```
1)  INSERT INTO Studio(name)
2)      SELECT DISTINCT studioName
3)      FROM Movie
4)      WHERE studioName NOT IN
5)          (SELECT name
6)           FROM Studio);
```

Figure 6.15: Adding new studios

Like most SQL statements with nesting, Fig. 6.15 is easiest to examine from the inside out. Lines (5) and (6) generate all the studio names in the relation Studio. Thus, line (4) tests that a studio name from the Movie relation is none of these studios.

Now, we see that lines (2) through (6) produce the set of studio names found in Movie but not in Studio. The use of DISTINCT on line (2) assures that each studio will appear only once in this set, no matter how many movies it owns. Finally, line (1) inserts each of these studios, with NULL for the attributes address and presC#, into relation Studio. □

Database Modifications

6.5.2 Deletion

A deletion statement consists of:

1. The keywords `DELETE FROM`,
2. The name of a relation, say R,
3. The keyword `WHERE`, and
4. A condition.

That is, the form of a deletion is

`DELETE FROM` R `WHERE` <condition>;

Database Modifications

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942 AND
      starName = 'Sydney Greenstreet';
```

Notice that unlike the insertion statement of Example 6.34, we cannot simply specify a tuple to be deleted. Rather, we must describe the tuple exactly by a WHERE clause. □

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

deletes all movie executives whose net worth is low — less than ten million dollars. □

Database Modifications

6.5.3 Updates

While we might think of both insertions and deletions of tuples as "updates" to the database, an *update* in SQL is a very specific kind of change to the database: one or more tuples that already exist in the database have some of their components changed. The general form of an update statement is:

1. The keyword `UPDATE`,
2. A relation name, say R,
3. The keyword `SET`,
4. A list of formulas that each set an attribute of the relation R equal to the value of an expression or constant,
5. The keyword `WHERE`, and
6. A condition.

That is, the form of an update is

`UPDATE` R `SET` <new-value assignments> `WHERE` <condition>;

Database Modifications

```
Studio(name, address, presC#)
```

Example 6.38: Let us modify the relation

```
MovieExec(name, address, cert#, netWorth)
```

by attaching the title `Pres.` in front of the name of every movie executive who is the president of a studio. The condition the desired tuples satisfy is that their certificate numbers appear in the `presC#` component of some tuple in the `Studio` relation. We express this update as:

```
1)  UPDATE MovieExec
2)  SET name = 'Pres. ' || name
3)  WHERE cert# IN (SELECT presC# FROM Studio);
```

Line (3) tests whether the certificate number from the `MovieExec` tuple is one of those that appear as a president's certificate number in `Studio`.

Line (2) performs the update on the selected tuples. Recall that the operator `||` denotes concatenation of strings, so the expression following the = sign in line (2) places the characters `Pres.` and a blank in front of the old value of the `name` component of this tuple. The new string becomes the value of the `name` component of this tuple; the effect is that `'Pres. '` has been prepended to the old value of `name`. □

Constraints in SQL

Keys and Foreign Keys

Perhaps the most important kind of constraint in a database is a declaration that a certain attribute or set of attributes forms a key for a relation. If a set of attributes S is a key for relation R, then any two tuples of R must disagree in at least one attribute in the set S. Note that this rule applies even to duplicate tuples; i.e., if R has a declared key, then R cannot have duplicates.

A key constraint, like many other constraints, is declared within the `CREATE TABLE` command of SQL. There are two similar ways to declare keys: using the keywords `PRIMARY KEY` or the keyword `UNIQUE`. However, a table may have *only* one primary key but any number of "unique" declarations.

Constraints in SQL

Primary Key

7.1.1 Declaring Primary Keys

A relation may have only one primary key. There are two ways to declare a primary key in the `CREATE TABLE` statement that defines a stored relation.

1. We may declare one attribute to be a primary key when that attribute is listed in the relation schema.

2. We may add to the list of items declared in the schema (which so far have only been attributes) an additional declaration that says a particular attribute or set of attributes forms the primary key.

Constraints in SQL

Keys and Foreign Keys

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE
    );
```

Figure 7.1: Making name the primary key

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30),
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE,
6)     PRIMARY KEY (name)
    );
```

Figure 7.2: A separate declaration of the primary

```
PRIMARY KEY (title, year)
```

Constraints in SQL

Unique constraint

1. WE may have any number of UNIQUE declarations for a table, but only one

primary key.

2. While PRIMARY KEY forbids NULL'S in the attributes of the key,

Example 7.2: Line (2) of Fig. 7.1 could have been written

```
2)      name CHAR(30) UNIQUE,
```

We could also change line (3) to

```
3)      address VARCHAR(255) UNIQUE,
```

```
6)      UNIQUE (name)
```

```
1)  CREATE TABLE MovieStar (
2)      name CHAR(30) PRIMARY KEY,
3)      address VARCHAR(255),
4)      gender CHAR(1),
5)      birthdate DATE
    );
```

Figure 7.1: Making name the

Constraints in SQL

Declaring Foreign key constraint

Example 7.3: Suppose we wish to declare the relation

```
Studio(name, address, presC#)
```

whose primary key is name and which has a foreign key presC# that references cert# of relation

```
MovieExec(name, address, cert#, netWorth)
```

We may declare presC# directly to reference cert# as follows:

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
);
```

An alternative form is to add the foreign key declaration separately, as

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

Maintaining Referential Integrity

The Default Policy: Reject Violating Modifications

```
MovieExec(name, address, cert#, netWorth)

Studio(name, address, presC#)
```

1. We try to insert a new `Studio` tuple whose `presC#` value is not `NULL` and is not the `cert#` component of any `MovieExec` tuple. The insertion is rejected by the system, and the tuple is never inserted into `Studio`.

2. We try to update a `Studio` tuple to change the `presC#` component to a non-`NULL` value that is not the `cert#` component of any `MovieExec` tuple. The update is rejected, and the tuple is unchanged.

3. We try to delete a `MovieExec` tuple, and its `cert#` component appears as the `presC#` component of one or more `Studio` tuples. The deletion is rejected, and the tuple remains in `MovieExec`.

4. We try to update a `MovieExec` tuple in a way that changes the `cert#` value, and the old `cert#` is the value of `presC#` of some movie studio. The system again rejects the change and leaves `MovieExec` as it was.

Maintaining Referential Integrity

The Cascade Policy

```
MovieExec(name, address, cert#, netWorth)

Studio(name, address, presC#)
```

Under the cascade policy, when we delete the `MovieExec` tuple for the president of a studio, then to maintain referential integrity the system will delete the referencing tuple(s) from `Studio`. Updates are handled analogously. If we change the `cert#` for some movie executive from c_1 to c_2, and there was some `Studio` tuple with c_1 as the value of its `presC#` component, then the system will also update this `presC#` component to have value c_2.

The Set-Null Policy

Yet another approach to handling the problem is to change the `presC#` value from that of the deleted or updated studio president to `NULL`; this policy is called *set-null*.

These options may be chosen for deletes and updates, independently, and they are stated with the declaration of the foreign key. We declare them with `ON DELETE` or `ON UPDATE` followed by our choice of `SET NULL` or `CASCADE`.

Maintaining Referential Integrity

The Cascade and Set Null Policy

MovieExec(name, address, cert#, netWorth)

Studio(name, address, presC#)

```
1)  CREATE TABLE Studio (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     presC# INT REFERENCES MovieExec(cert#)
5)         ON DELETE SET NULL
6)         ON UPDATE CASCADE
    );
```

Figure 7.3: Choosing policies to preserve referential integrity

Note that in this example, the set-null policy makes more sense for deletes, while the cascade policy seems preferable for updates. We would expect that if, for instance, a studio president retires, the studio will exist with a "null" president for a while. However, an update to the certificate number of a studio president is most likely a clerical change. The person continues to exist and to be the president of the studio, so we would like the **presC#** attribute in **Studio** to follow the change. □

Maintaining Referential Integrity

ON DELETE CASCADE

the referencing rows in the child table is deleted when the referenced row is deleted in the parent table which has a primary key.

ON UPDATE CASCADE

referencing rows are updated in the child table when the referenced row is updated in the parent table which has a primary key.

ON DELETE SET NULL

child data is set to NULL when the parent data is deleted. The child data is NOT deleted.

ON UPDATE SET NULL

the child data is set to NULL when the parent data is updated.

Constraints on Attributes and Tuples

```
1)  CREATE TABLE Studio (
2)      name CHAR(30) PRIMARY KEY,
3)      address VARCHAR(255),
4)      presC# INT REFERENCES MovieExec(cert#)
5)          ON DELETE SET NULL
6)          ON UPDATE CASCADE
    );
```

7.2.1 Not-Null Constraints

One simple constraint to associate with an attribute is NOT NULL. The effect is to disallow tuples in which this attribute is NULL. The constraint is declared by the keywords NOT NULL following the declaration of the attribute in a CREATE TABLE statement.

Example 7.7: Suppose relation Studio required presC# not to be NULL, perhaps by changing line (4) of Fig. 7.3 to:

```
4)      presC# INT REFERENCES MovieExec(cert#) NOT NULL
```

This change has several consequences. For instance:

- We could not insert a tuple into Studio by specifying only the name and address, because the inserted tuple would have NULL in the presC# component.
- We could not use the set-null policy in situations like line (5) of Fig. 7.3, which tells the system to fix foreign-key violations by making presC# be NULL.

Constraints on Attributes and Tuples

Attribute based CHECK constraint

Suppose we want to require that certificate numbers be at least six digits

```
Studio(name, address, presC#)
```

to be

```
4)    presC# INT REFERENCES MovieExec(cert#)
                CHECK (presC# >= 100000)
```

For another example, the attribute gender of relation

```
MovieStar(name, address, gender, birthdate)
```

was declared in Fig. 6.16 to be of data type CHAR(1) — that is, a single character. However, we really expect that the only characters that will appear there are 'F' and 'M'. The following substitute for line (4) of Fig. 6.16 enforces the rule:

```
4) gender CHAR(1) CHECK (gender IN ('F', 'M')),
```

Constraints on Attributes and Tuples

Tuple based CHECK constraint

- If you have a constraint that include multiple columns it will be tuple-based.
- The condition of a tuple-based CHECK constraint is checked every time a tuple is inserted into R and every time a tuple of R is updated
- If the condition is false for that tuple, then the constraint is violated and the insertion or update statement that caused the violation is rejected.
- This constraint says that if the star's gender is male, then his name must not begin with 'Ms. '.

```
1)  CREATE TABLE MovieStar (
2)      name CHAR(30) PRIMARY KEY,
3)      address VARCHAR(255),
4)      gender CHAR(1),
5)      birthdate DATE,
6)      CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
    );
```

In line (2), name is declared the primary key for the relation. Then line (6) declares a constraint. The condition of this constraint is true for every female movie star and for every star whose name does not begin with 'Ms.'. The only tuples for which it is *not* true are those where the gender is male and the name *does* begin with 'Ms.'. Those are exactly the tuples we wish to exclude from MovieStar. □

Constraints on Attributes and Tuples

Tuple-Based Constraints

Constraints on values for `Relation.Tuple`

Similar to attribute-based constraints except `CHECK` applies to entire tuples.
=> Specified separately in table definition
Example:

```
CREATE TABLE Campus(location char(25),
                    enrollment integer,
                    rank integer,
                    CHECK(enrollment >= 10,000 OR rank > 5)
```

Constraints on Attributes and Tuples

To create a PC table such that records whose speed is less than 2.0 and price greater than 600 are rejected is as follows:

```
SQL> create table PC(model varchar(15),
  2  speed number(7,2),
  3  ram number(4),
  4  hd number(5),
  5  price number(8,2),
  6  primary key(model,speed),
  7  check (speed>=2.0 or price<=600));

Table created.
```

Query Explanation:

- **Line 1:** create table key word is used to create a new table with specified attributes.
- **Line 6:** model, speed, attributes are declared as the primary key for the table PC.
- **Line 7:** 'check' constraint has been imposed. The condition states that only records whose speed value is greater than 2.0 will be inserted or records with price value less than or equal to 600 are inserted. In other words, that is only records with speed value < 2.0 and price value> 600 are rejected. All other records violating this check constraint are rejected.

Output:

Modification of Constraints

7.3.1 Giving Names to Constraints

In order to modify or delete an existing constraint, it is necessary that the constraint have a name. To do so, we precede the constraint by the keyword CONSTRAINT and a name for the constraint.

Example 7.11: We could rewrite line (2) of Fig. 7.1 to name the constraint that says attribute name is a primary key, as

```
2)    name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY,
```

Similarly, we could name the attribute-based CHECK constraint that appeared in Example 7.8 by:

```
4) gender CHAR(1) CONSTRAINT NoAndro
                  CHECK (gender IN ('F', 'M')),
```

Finally, the following constraint:

```
6)    CONSTRAINT RightTitle
          CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

Modification of Constraints

```
ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;
ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;
ALTER TABLE MovieStar DROP CONSTRAINT RightTitle;

ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey
    PRIMARY KEY (name);
ALTER TABLE MovieStar ADD CONSTRAINT NoAndro
    CHECK (gender IN ('F', 'M'));
ALTER TABLE MovieStar ADD CONSTRAINT RightTitle
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

Basics of Relational Algebra

The operations of the traditional relational algebra fall into four broad classes:

a) **Set operations** - union, intersection, and difference - applied to relations.

b) Operations that remove parts of a relation: "selection" eliminates some rows (tuples), and "projection" eliminates some columns.

c) Operations that combine the tuples of two relations: "**Cartesian product**" which pairs the tuples of two relations in all possible ways, and various kinds of "**join**" operations, which selectively pair tuples from two relations.

d) An operation called 'renaming' that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

Set operations on Relations

- $R \cup S$, the *union* of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.
- $R \cap S$, the *intersection* of R and S, is the set of elements that are in both R and S.
- $R - S$, the *difference* of R and S, is the set of elements that are in R but not in S. Note that $R - S$ is different from $S - R$; the latter is the set of elements that are in S but not in R.

Set operations on Relations

When we apply these operations to relations, we need to put some conditions on R and S:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.

2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

Set operations on Relations

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation R

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Relation S

Figure 5.2: Two relations

Set operations on Relations

union $R \cup S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Note that the two tuples for Carrie Fisher from the two relations appear only once in the result.

Set operations on Relations

The intersection $R \cap S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Now, only the Carrie Fisher tuple appears, because only it is in both relations.

The difference $R - S$ is

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

That is, the Fisher and Hamill tuples appear in R and thus are candidates for $R - S$. However, the Fisher tuple also appears in S and so is not in $R - S$. □

Set operations on Relations

R

Cust_name	Cust_status
Sham	Good
Rahul	Excellent
Mohan	Bad
Sachin	Excellent
Dinesh	Bad

S

Cust_name	Cust_status
Karan	Bad
Sham	Good
Sachin	Excellent
Rohan	Average

Projection

Projection

Projection operator is used to produce from a relation R, a new relation that has unique some of R's columns

$$\pi_{title,year,length}(\text{Movie})$$

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

Figure 5.3: The relation Movie

Example 5.2: Consider the relation Movie with the relation schema described in Section 5.1. An instance of this relation is shown in Fig. 5.3. We can project this relation onto the first three attributes with the expression

$$\pi_{title,year,length}(\text{Movie})$$

The resulting relation is

title	*year*	*length*
Star Wars	1977	124
Mighty Ducks	1991	104
Wayne's World	1992	95

As another example, we can project onto the attribute inColor with the expression $\pi_{inColor}(\text{Movie})$. The result is the single-column relation

inColor
true

Selection

Selection

Selection operator when applied to a relation R produces a new relation with a subset of R's tuples.

$$\sigma_C(R).$$

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

Example 5.3: Let the relation Movie be as in Fig. 5.3. Then the value of expression $\sigma_{length \geq 100}(\text{Movie})$ is

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890

Selection

Selection

The selection operator, applied to a relation R, produces a new relation with a subset of R's tuples. The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of R. We denote this operation $\sigma_C(R)$.

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

Figure 5.3: The relation Movie

$$\sigma_{length \geq 100 \text{ AND } studioName = \text{'Fox'}}(\text{Movie})$$

The tuple

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345

is the only one in the resulting relation. □

Cartesian Product

Cartesian Product

The Cartesian product (or cross-product, or just product) of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S. This product is denoted R x S. When R and S are relations, the product is essentially the same.

A	B
1	2
3	4

Relation R

B	C	D
2	5	6
4	7	8
9	10	11

Relation S

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Result $R \times S$

Natural Join

Natural Join

- Is denoted by the symbol ⋈
- Consider the Relation R and S, the natural join is represented as R ⋈ S
- We pair only those tuples from R and S that agree in attributes which are common to the schema.
- The tuple that fails to pair with any tuple of the other relation in a join is called dangling tuple.

Natural Join

A	B
1	2
3	4

Relation R

B	C	D
2	5	6
4	7	8
9	10	11

Relation S

Example 5.6: The natural join of the relations R and S from Fig. 5.4 is

A	B	C	D
1	2	5	6
3	4	7	8

The only attribute common to R and S is B. Thus, to pair successfully, tuples need only to agree in their B components. If so, the resulting tuple has components for attributes A (from R), B (from either R or S), C (from S), and D (from S).

In this example, the first tuple of R successfully pairs with only the first tuple of S; they share the value 2 on their common attribute B. This pairing yields the first tuple of the result: $(1, 2, 5, 6)$. The second tuple of R pairs successfully only with the second tuple of S, and the pairing yields $(3, 4, 7, 8)$. Note that the third tuple of S does not pair with any tuple of R and thus has no effect on the result of $R \bowtie S$. A tuple that fails to pair with any tuple of the other relation in a join is said to be a *dangling tuple.* □

Natural Join

A	B	C
1	2	3
6	7	8
9	7	8

Relation U

B	C	D
2	3	4
2	3	5
7	8	10

Relation V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

Result $U \bowtie V$

Figure 5.6: Natural join of relations

Theta Join

Theta Join

The notation for a theta-join of relations R and S based on condition C is $R \overset{\bowtie}{}_{C} S$. The result of this operation is constructed as follows:

1. Take the product of R and S.
2. Select from the product only those tuples that satisfy the condition C.

Theta Join

Theta Join

A	B	C
1	2	3
6	7	8
9	7	8

Relation U

B	C	D
2	3	4
2	3	5
7	8	10

Relation V

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

Figure 5.7: Result of $U \overset{\bowtie}{}_{A<D} V$

Theta Join

Theta Join

A	B	C
1	2	3
6	7	8
9	7	8

Relation U

B	C	D
2	3	4
2	3	5
7	8	10

Relation V

$$U \underset{A<D \text{ AND } U.B \neq V.B}{\bowtie} V$$

A	U.B	U.C	V.B	V.C	D
1	2	3	7	8	10

Relational Operations on Bags

When we refer to a "set," we mean a relation without duplicate tuples; a "bag" means a relation that may (or may not) have duplicate tuples.

Example 5.15: The relation in Fig. 5.14 is a bag of tuples. In it, the tuple $(1,2)$ appears three times and the tuple $(3,4)$ appears once. If Fig. 5.14 were a set-valued relation, we would have to eliminate two occurrences of the tuple $(1,2)$. In a bag-valued relation, we *do* allow multiple occurrences of the same tuple, but like sets, the order of tuples does not matter. □

A	*B*
1	2
3	4
1	2
1	2

Figure 5.14: A bag

Relational Operations on Bags

Why Bags?

SQL, the most important query language for relational databases, is actually a bag language.

Some operations, like projection, are more efficient on bags than sets.

As a simple example of why bags can lead to implementation efficiency, if you take the union of two relations but do not eliminate duplicates, then you can just copy the relations to the output.

If you insist that the result, be a set, you have to sort the relations, or do something similar to detect identical tuples that come from the two relations.

Union, Intersection and Difference on Bags

A	B
1	2
3	4
1	2
1	2

A	B
1	2
3	4
3	4
5	6

Then the bag union $R \cup S$ is the bag in which $(1,2)$ appears four times (three times for its occurrences in R and once for its occurrence in S); $(3,4)$ appears three times, and $(5,6)$ appears once.

The bag intersection $R \cap S$ is the bag

A	B
1	2
3	4

with one occurrence each of $(1,2)$ and $(3,4)$. That is, $(1,2)$ appears three times in R and once in S, and $\min(3,1) = 1$, so $(1,2)$ appears once in $R \cap S$. Similarly, $(3,4)$ appears $\min(1,2) = 1$ time in $R \cap S$. Tuple $(5,6)$, which appears once in S but zero times in R appears $\min(0,1) = 0$ times in $R \cap S$.

Union, Intersection and Difference on Bags

A	B
1	2
3	4
1	2
1	2

A	B
1	2
3	4
3	4
5	6

The bag difference $R - S$ is the bag

A	B
1	2
1	2

To see why, notice that $(1, 2)$ appears three times in R and once in S, so in $R - S$ it appears $\max(0, 3 - 1) = 2$ times. Tuple $(3, 4)$ appears once in R and twice in S, so in $R - S$ it appears $\max(0, 1 - 2) = 0$ times. No other tuple appears in R, so there can be no other tuples in $R - S$.

As another example, the bag difference $S - R$ is the bag

A	B
3	4
5	6

Tuple $(3, 4)$ appears once because that is the difference in the number of times it appears in S minus the number of times it appears in R. Tuple $(5, 6)$ appears once in $S - R$ for the same reason. The resulting bag happens to be a set in this case. □

Projection on Bags

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

the bag-projection $\pi_{A,B}(R)$

three occurrences of tuple (1, 2),

Selection on Bags

Example 5.18: If R is the bag

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

then the result of the bag-selection $\sigma_{C \geq 6}(R)$ is

A	B	C
3	4	6
1	2	7
1	2	7

Product on Bags

A	B
1	2
1	2

(a) The relation R

B	C
2	3
4	5
4	5

(b) The relation S

A	$R.B$	$S.B$	C
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

(c) The product $R \times S$

Joins on Bags

A	B
1	2
1	2

(a) The relation R

B	C
2	3
4	5
4	5

(b) The relation S

Example 5.20: The natural join $R \bowtie S$ of the relations R and S seen in Fig. 5.16 is

A	B	C
1	2	3
1	2	3

That is, tuple $(1,2)$ of R joins with $(2,3)$ of S. Since there are two copies of $(1,2)$ in R and one copy of $(2,3)$ in S, there are two pairs of tuples that join to give the tuple $(1,2,3)$. No other tuples from R and S join successfully.

Joins on Bags

As another example on the same relations R and S, the theta-join

$$R \underset{R.B<S.B}{\bowtie} S$$

produces the bag

A	$R.B$	$S.B$	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

A	B
1	2
1	2

(a) The relation R

B	C
2	3
4	5
4	5

(b) The relation S

The computation of the join is as follows. Tuple $(1, 2)$ from R and $(4, 5)$ from S meet the join condition. Since each appears twice in its relation, the number of times the joined tuple appears in the result is 2×2 or 4. The other possible join of tuples — $(1, 2)$ from R with $(2, 3)$ from S — fails to meet the join condition, so this combination does not appear in the result. □

Renaming

The rename operation is used to rename the output relation. It is denoted by rho (ρ).

Example: We can use the rename operator to rename STUDENT relation to STUDENT1.

ρSTUDENT1(STUDENT)

In order to control the names of the attributes used for relations that are constructed by applying relational-algebra operations, it is often convenient to use an operator that explicitly renames relations. We shall use the operator $\rho_{S(A_1,A_2,\ldots,A_n)}(R)$ to rename a relation R. The resulting relation has exactly the same tuples as R, but the name of the relation is S. Moreover, the attributes of the result relation S are named $A_1, A_2, \ldots, A_n$, in order from the left. If we only want to change the name of the relation to S and leave the attributes as they are in R, we can just say $\rho_S(R)$.

Renaming

Suppose, however, that we do not wish to call the two versions of B by names $R.B$ and $S.B$; rather we want to continue to use the name B for the attribute that comes from R, and we want to use X as the name of the attribute B coming from S. We can rename the attributes of S so the first is called X. The result of the expression $\rho_{S(X,C,D)}(S)$ is a relation named S that looks just like the relation S from Fig. 5.4, but its first column has attribute X instead of B.

A	B
1	2
3	4

Relation R

B	C	D
2	5	6
4	7	8
9	10	11

Relation S

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Result $R \times \rho_{S(X,C,D)}(S)$

Renaming

Renaming: ρ

- Given relation $R(A, B, C)$, $\rho_{S(X,Y,Z)}(R)$ renames it to $S(X, Y, Z)$

Actors

actor	ayear
Cage	1964
Hanks	1956
Maguire	1975
McDormand	1957

$\rho_{Stars(name,yob)}(\mathbf{Actors})$

Stars

name	yob
Cage	1964
Hanks	1956
Maguire	1975
McDormand	1957

Renaming

Renaming

$\rho_{S(A_1,\ldots,A_n)}(R)$ produces a relation identical to R but named S and with attributes, in order, named $A_1, \ldots, A_n$.

Example

`Bars` =

name	addr
Joe's	Maple St.
Sue's	River Rd.

$\rho_{R(bar,addr)}$(`Bars`) =

bar	addr
Joe's	Maple St.
Sue's	River Rd.

- The name of the above relation is R.

Extended Operators of Relational Algebra

- The ***duplicate-elimination*** *operator* δ
- ***Aggregation operators like*** *sum, average*
- ***Grouping of tuples:*** *The grouping operator* γ *is an operator that combines the effect of grouping and aggregation.*

- *The* ***sorting operator*** τ
- ***Extending the projection operator*** *gives additional power*
- *The* ***outer join operator*** *is a variant of the join that avoids losing dangling tupes.*

Extended Operators of Relational Algebra

Duplicate-elimination operator δ

- Sometimes, we need an operator that converts a bag to a set.
- δ(R) to return the set consisting of one copy of every tuple that appears one or more times in relation R.

If *R is the relation*

A	B
1	2
3	4
1	2
1	2

δ(R)

A	B
1	2
3	4

Extended Operators of Relational Algebra

Aggregation operators

- There are several operators that apply to sets or bags of atomic values.
- These operators are used to summarize or "aggregate" the values in one column of a relation, and thus are referred to as *aggregation operators*
- **SUM** produces the sum of a column with numerical values.
- **AVG** produces the average of a column with numerical values.
- **MIN and MAX** applied to a column with numerical values, produces the smallest or largest value, respectively. When applied to a column with character-string values, they produce the lexicographically (alphabetically) first or last value, respectively.
- **COUNT** produces the number of (not necessarily distinct) values in a column.

Extended Operators of Relational Algebra

Aggregation operators

A	B
1	2
3	4
1	2
1	2

1. SUM(B) = 2 + 4 + 2 + 2 = 10.
2. AVG(A) = (1 + 3 + 1 + 1)/4 = 1.5.
3. MIN(A) = 1.
4. MAX(B) = 4.
5. COUNT(A) = 4.

Extended Operators of Relational Algebra

Sorting τ L(R) = list of tuples of R, ordered according to attributes on list L.

$R =$

A	B
1	3
3	4
5	2

$\tau_B(R) = [(5,2),\ (1,3),\ (3,4)].$

Example: Sorting

R =

A	B
1	2
3	4
5	2

τ_B (R) =

A	B
5	2
1	2
3	4

Extended Operators of Relational Algebra

Extended Projection

Allow the columns in the projection to be functions of one or more columns in the argument relation.

Example

$R =$

A	B
1	2
3	4

$\pi_{A+B,A,A}(R) =$

$A+B$	$A1$	$A2$
3	1	1
7	3	3

Extended Operators of Relational Algebra

Extended Projection

A	B	C
0	1	2
0	1	2
3	4	5

Then the result of $\pi_{A,B+C \to X}(R)$ is

A	X
0	3
0	3
3	9

The result's schema has two attributes. One is A, the first attribute of R, not renamed. The second is the sum of the second and third attributes of R, with the name X.

For another example, $\pi_{B-A \to X, C-B \to Y}(R)$ is

X	Y
1	1
1	1
1	1

Extended Operators of Relational Algebra

Grouping operator

Example

Let $R =$

bar	beer	price
Joe's	Bud	2.00
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00
Mel's	Miller	3.25

Compute $\gamma_{beer,AVG(price)}(R)$.

Group by the grouping attribute(s), `beer` in this case:

bar	beer	price
Joe's	Bud	2.00
Sue's	Bud	2.50
Joe's	Miller	2.75
Mel's	Miller	3.25
Sue's	Coors	3.00

beer	AVG(price)
Bud	2.25
Miller	3.00
Coors	3.00

Extended Operators of Relational Algebra

Outer Join

Outerjoin

The normal join can "lose" information, because a tuple that doesn't join with any from the other relation (*dangles*) has no vestage in the join result.

- The *null value* $\perp$ can be used to "pad" dangling tuples so they appear in the join.
- Gives us the *outerjoin* operator $\overset{\circ}{\bowtie}$.
- Variations: theta-outerjoin, left- and right-outerjoin (pad only dangling tuples from the left (resp., right).

Extended Operators of Relational Algebra

Outer Join

We shall consider the "natural" case first, where the join is on equated values of all attributes in common to the two relations. The *outerjoin* $R \overset{\circ}{\bowtie} S$ is formed by starting with $R \bowtie S$, and adding any dangling tuples from R or S. The added tuples must be padded with a special *null* symbol, $\perp$, in all the attributes that they do not possess but that appear in the join result.[5]

$R =$

A	B
1	2
3	4

$S =$

B	C
4	5
6	7

$R \overset{\circ}{\bowtie} S =$

A	B	C
3	4	5
1	2	$\perp$
$\perp$	6	7

Extended Operators of Relational Algebra

Outer Join

A	B	C
1	2	3
4	5	6
7	8	9

Relation U

B	C	D
2	3	10
2	3	11
6	7	12

Relation V

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$
$\perp$	6	7	12

Result $U \overset{\circ}{\bowtie} V$

Extended Operators of Relational Algebra

Left and Right Outer Join

There are many variants of the basic (natural) outerjoin idea. The *left outerjoin* $R \overset{\circ}{\bowtie}_L S$ is like the outerjoin, but only dangling tuples of the left argument R are padded with $\perp$ and added to the result. The *right outerjoin* $R \overset{\circ}{\bowtie}_R S$ is like the outerjoin, but only the dangling tuples of the right argument S are padded with $\perp$ and added to the result.

A	B	C
1	2	3
4	5	6
7	8	9

Relation U

B	C	D
2	3	10
2	3	11
6	7	12

Relation V

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$

$U \overset{\circ}{\bowtie}_L V$

A	B	C	D
1	2	3	10
1	2	3	11
$\perp$	6	7	12

$U \overset{\circ}{\bowtie}_R V$

r =

attr1	attr2
a	r1
b	r2
c	r3

s =

attr1	attr3
b	s2
c	s3
d	s4

$r \bowtie s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3

$r \bowtie s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	*null*	s4

$r \bowtie s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3
d	*null*	s4

Extended Operators of Relational Algebra

Theta Outer Join

A	B	C
1	2	3
4	5	6
7	8	9

Relation U

B	C	D
2	3	10
2	3	11
6	7	12

Relation V

$$U \overset{\circ}{\bowtie}_{A>V.C} V$$

A	U.B	U.C	V.B	V.C	D
4	5	6	2	3	10
4	5	6	2	3	11
7	8	9	2	3	10
7	8	9	2	3	11
1	2	3	⊥	⊥	⊥
⊥	⊥	⊥	6	7	12

Combining Operations to form Queries

Select the tuples for all employees who work in department 4

Select * from Employee where Dno = 4.

σ(Dno=4) (EMPLOYEE)

Select the tuples for all employees who work in department 4 and make over $25,000 per year

σ(Dno=4 AND Salary>25000) (EMPLOYEE)

Select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000

σ(Dno=4 AND Salary>25000) OR (Dno=5 AND Salary>30000) (EMPLOYEE)

List the last and first name for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000

πLname, Fname(σ(Dno=4 AND Salary>25000) OR (Dno=5 AND Salary>30000)(EMPLOYEE))

Combining Operations to form Queries

For most queries, we need to apply several relational algebra operations one after the other.

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.

Combining Operations to form Queries

For example, **to retrieve the first name, last name, and salary of all employees who work in department number 5.**

we must apply a SELECT and a PROJECT operation.

We can write a single relational algebra expression

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$$

It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.

Combining Operations to form Queries

Sailors(sid,sname,rating,age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance $S3$ of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance $B1$ of Boats

Combining Operations to form Queries

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:

Sailors(sid,sname,rating,age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

$$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$$

We can break this query into smaller pieces using the renaming operator ρ:

$$\rho(Temp1, \sigma_{bid=103} Reserves)$$

$$\rho(Temp2, Temp1 \bowtie Sailors)$$

$$\pi_{sname}(Temp2)$$

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

In this version we first compute the natural join of Reserves and Sailors and then apply the selection and the projection.

Combining Operations to form Queries

This example offers a glimpse of the role played by algebra in a relational DBMS. Queries are expressed by users in a language such as SQL. The DBMS translates an SQL query into (an extended form of) relational algebra, and then looks for other algebra expressions that will produce the same answers but are cheaper to evaluate. If the user's query is first translated into the expression

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

a good query optimizer will find the equivalent expression

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie Sailors)$$

Further, the optimizer will recognize that the second expression is likely to be less expensive to compute because the sizes of intermediate relations are smaller, thanks to the early use of selection.

Combining Operations to form Queries

(Q2) Find the names of sailors who have reserved a red boat.

$$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

This query involves a series of two joins. First we choose (tuples describing) red boats. Then we join this set with Reserves (natural join, with equality specified on the bid column) to identify reservations of red boats. Next we join the resulting intermediate relation with Sailors (natural join, with equality specified on the sid column) to retrieve the names of sailors who have made reservations of red boats. Finally, we project the sailors' names.

Sailors(sid,sname,rating,age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

Combining Operations to form Queries

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'} Sailors) \bowtie Reserves \bowtie Boats)$$

This query is very similar to the query we used to compute sailors who reserved red boats.

Sailors(sid,sname,rating,age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi_{sname}(Sailors \bowtie Reserves)$$

The join of Sailors and Reserves creates an intermediate relation in which tuples consist of a Sailors tuple 'attached to' a Reserves tuple. A Sailors tuple appears in (some tuple of) this intermediate relation only if at least one Reserves tuple has the same *sid* value, that is, the sailor has made some reservation.

Combining Operations to form Queries

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(Tempboats, (\sigma_{color='red'} Boats) \cup (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

$$\rho(Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

Combining Operations to form Queries

Expression Tree

Is used to represent the various components of relational algebra equations.

Leaf node – relation name

Intermediate node – condition

Root node – Tuples to be retrieved.

Select firstname, lastname, salary from emp where deptno = 5.

Combining Operations to form Queries

Expression Tree

What are the titles and years of movies made by Fox that are at least 100 minutes long?

Movies(name, title, year, length, studioName)

1. Select those `Movies` tuples that have $length \geq 100$.
2. Select those `Movies` tuples that have $studioName =$ `'Fox'`.
3. Compute the intersection of (1) and (2).
4. Project the relation from (3) onto attributes `title` and `year`.

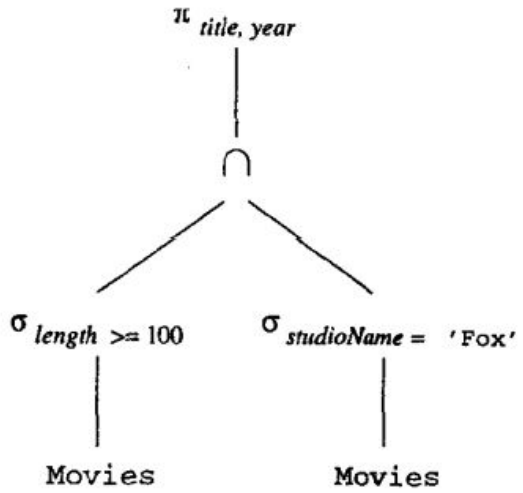


Figure 5.8: Expression tree for a relational algebra expression

Combining Operations to Form Queries

In Fig. 5.8 we see the above steps represented as an expression tree. The two selection nodes correspond to steps (1) and (2). The intersection node corresponds to step (3), and the projection node is step (4).

Alternatively, we could represent the same expression in a conventional, linear notation, with parentheses. The formula

$$\pi_{title,year}\left(\sigma_{length\geq 100}(\texttt{Movies}) \cap \sigma_{studioName=\texttt{'Fox'}}(\texttt{Movies})\right)$$

represents the same expression.

Incidentally, there is often more than one relational algebra expression that represents the same computation. For instance, the above query could also be written by replacing the intersection by logical **AND** within a single selection operation. That is,

$$\pi_{title,year}\left(\sigma_{length\geq 100\ \texttt{AND}\ studioName=\texttt{'Fox'}}(\texttt{Movies})\right)$$

is an equivalent form of the query. □

Combining Operations to form Queries

Movies1 with schema {title, year, length, filmType, studioName}
Movies2 with schema {title, year, starName}

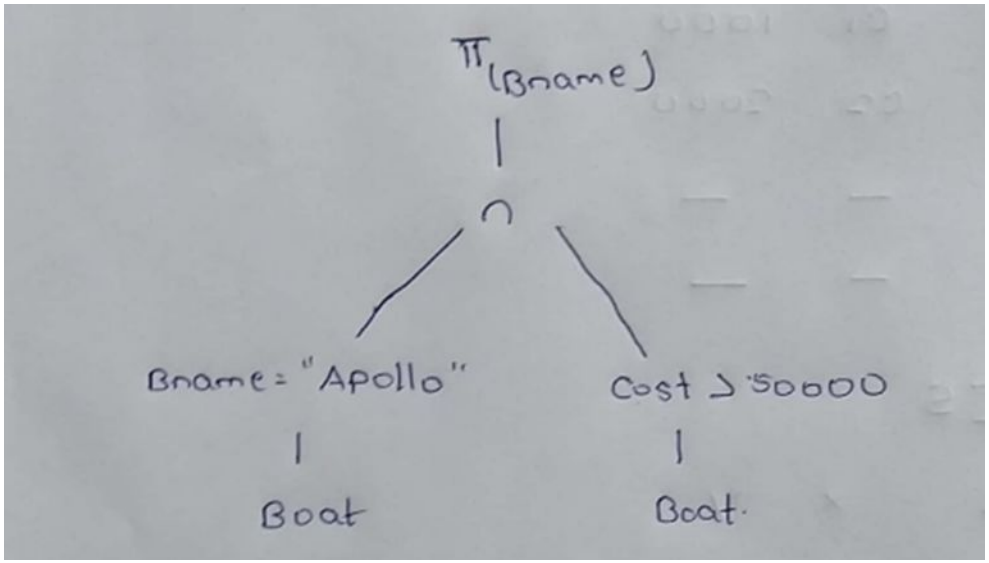
write an expression to answer the query "Find the stars of movies that are at least 100 minutes long."

$$\pi_{starName}\left(\sigma_{length \geq 100}(\text{Movies1} \bowtie \text{Movies2})\right)$$

Combining Operations to form Queries

Draw the expression tree for the given Query.

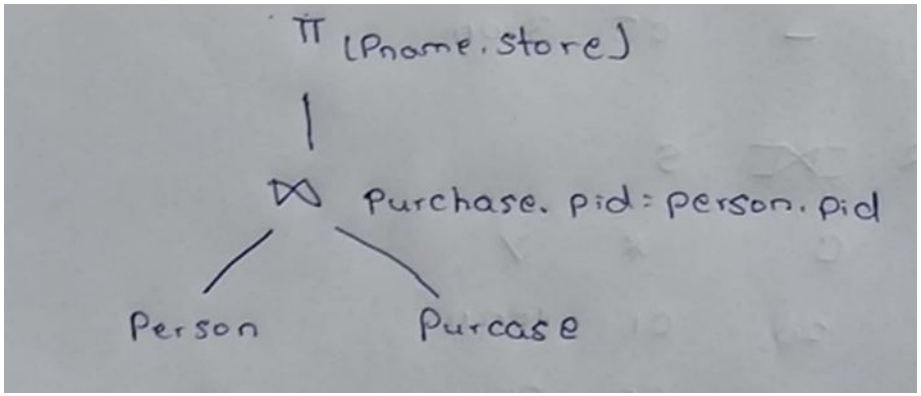
Boat(Bid, Bname, Cost)
Person(Pid, Pname, City)
Purchase(Bid, Pid, Store)
Select bname from Boat where Bname = “Apollo” and Cost > 50000



Combining Operations to form Queries

Draw the expression tree for the given Query.

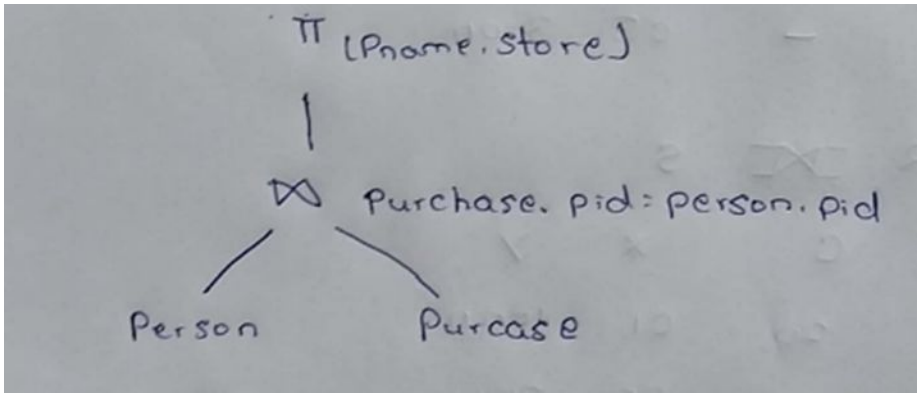
Boat(Bid, Bname, Cost)
Person(Pid, Pname, City)
Purchase(Bid, Pid, Store)
Select Pname, Store from Person, Purchase where Purchase.Pid=Person.Pid



Combining Operations to form Queries

Draw the expression tree for the given Query.

Boat(Bid, Bname, Cost)
Person(Pid, Pname, City)
Purchase(Bid, Pid, Store)
Select Pname, Store from Person, Purchase where Purchase.Pid=Person.Pid

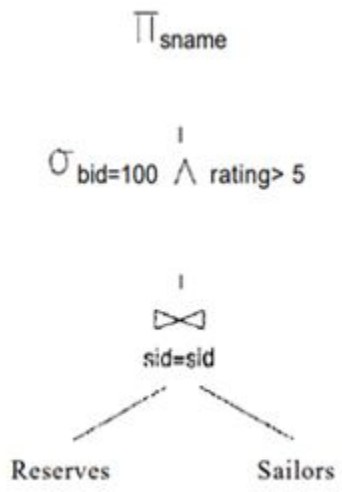


Combining Operations to form Queries

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
```

This query can be expressed in relational algebra as follows:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$



Simple Queries in SQL

Database Schema

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Simple Queries in SQL

Simple Queries in SQL

```
Movie(title, year, length, inColor, studioName, producerC#)
```

for all movies produced by Disney Studios in 1990. In SQL, we say

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

This query exhibits the characteristic select-from-where form of most SQL queries.

Simple Queries in SQL

Projection in SQL

Example 6.2: Suppose we wish to modify the query of Example 6.1 to produce only the movie title and length. We may write

```
SELECT title, length
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

The result is a table with two columns, headed `title` and `length`. The tuples in this table are pairs, each consisting of a movie title and its length, such that the movie was produced by Disney in 1990. For instance, the relation schema and one of its tuples looks like:

title	*length*
Pretty Woman	119
...	...

Simple Queries in SQL

Projection in SQL

Example 6.3: We can modify Example 6.2 to produce a relation with attributes `name` and `duration` in place of `title` and `length` as follows.

```
SELECT title AS name, length AS duration
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

The result is the same set of tuples as in Example 6.2, but with the columns headed by attributes `name` and `duration`. For example, the result relation might begin:

name	*duration*
Pretty Woman	119

Simple Queries in SQL

Projection in SQL

Example 6.4: Suppose we wanted output as in Example 6.3, but with the length in hours. We might replace the SELECT clause of that example with

```
SELECT title AS name, length*0.016667 AS lengthInHours
```

Then the same movies would be produced, but lengths would be calculated in hours and the second column would be headed by attribute `lengthInHours`, as:

name	*lengthInHours*
Pretty Woman	1.98334
...	...

Simple Queries in SQL

Projection in SQL

Example 6.5: We can even allow a constant as an expression in the SELECT clause. It might seem pointless to do so, but one application is to put some useful words into the output that SQL displays. The following query:

```
SELECT title, length*0.016667 AS length, 'hrs.' AS inHours
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

produces tuples such as

title	*length*	*inHours*
Pretty Woman	1.98334	hrs.
...	...	...

We have arranged that the third column is called `inHours`, which fits with the column header `length` in the second column. Every tuple in the answer will have the constant `hrs.` in the third column, which gives the illusion of being the units attached to the value in the second column. □

Simple Queries in SQL

Selection in SQL

The simple SQL queries that we have seen so far all have the form:

```
SELECT L
FROM R
WHERE C
```

in which L is a list of expressions, R is a relation, and C is a condition. The meaning of any such expression is the same as that of the relational-algebra expression

$$\pi_L(\sigma_C(R))$$

Simple Queries in SQL

Selection in SQL

Example 6.6: The following query asks for all the movies made after 1970 that are in black-and-white.

```
SELECT title
    FROM Movie
    WHERE year > 1970 AND NOT inColor;
```

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90) AND studioName = 'MGM';
```

Simple Queries in SQL

Comparison of Strings

```
SELECT title

       FROM Movie
       WHERE title LIKE 'Star ____';
```

This query asks if the title attribute of a movie has a value that is nine characters long, the first five characters being `Star` and a blank. The last four characters may be anything, since any sequence of four characters matches the four _ symbols. The result of the query is the set of complete matching titles, such as *Star Wars* and *Star Trek*. □

Simple Queries in SQL

Comparison of Strings

Example 6.8: Let us search for all movies with a possessive ('s) in their titles. The desired query is

```
SELECT title
FROM Movie
WHERE title LIKE '%''s%';
```

To understand this pattern, we must first observe that the apostrophe, being the character that surrounds strings in SQL, cannot also represent itself. The convention taken by SQL is that two consecutive apostrophes in a string represent a single apostrophe and do not end the string. Thus, `''s` in a pattern is matched by a single apostrophe followed by an `s`.

The two `%` characters on either side of the `'s` match any strings whatsoever. Thus, any title with `'s` as a substring will match the pattern, and the answer to this query will include films such as *Logan's Run* or *Alice's Restaurant*. □

Simple Queries in SQL

Dates and Times

A *date* constant is represented by the keyword DATE followed by a quoted string of a special form. For example, DATE '1948-05-14' follows the required form. The first four characters are digits representing the year. Then come a

A *time* constant is represented similarly by the keyword TIME and a quoted string. This string has two digits for the hour, on the military (24-hour) clock. Then come a colon, two digits for the minute, another colon, and two digits for the second. If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like. For instance, TIME '15:00:02.5' represents the time at which all students will have left a class that ends at 3 PM: two and a half seconds past three o'clock.

To combine dates and times we use a value of type TIMESTAMP. These values consist of the keyword TIMESTAMP, a date value, a space, and a time value. Thus, TIMESTAMP '1948-05-14 12:00:00' represents noon on May 14, 1948.

We can compare dates or times using the same comparison operators we use for numbers or strings. That is, < on dates means that the first date is earlier than the second; < on times means that the first is earlier (within the same day) than the second.

Simple Queries in SQL

Ordering the Output

```
ORDER BY <list of attributes>
```

The order is by default ascending. but we can get the output highest-first by appending the keyword **DESC** (for "descending") to an attribute. Similarly. we can specify ascending order with the keyword **ASC**, but that word is unnecessary.

To get the movies listed by length, shortest first, and among movies of equal length, alphabetically, we can say:

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

Queries Involving More Than One Relation

Products and Joins in SQL

Example 6.12: Suppose we want to know the name of the producer of *Star Wars*. To answer this question we need the following two relations from our running example:

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

Queries Involving More Than One Relation

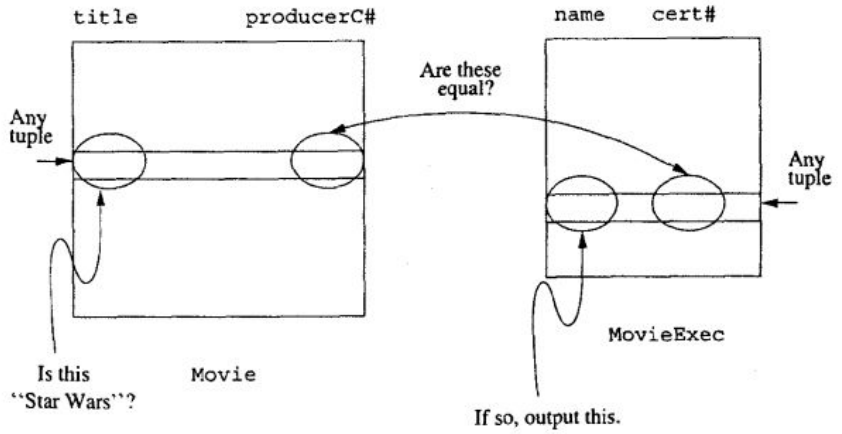
Products and Joins in SQL

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

This query asks us to consider all pairs of tuples, one from `Movie` and the other from `MovieExec`. The conditions on this pair are stated in the `WHERE` clause:

1. The `title` component of the tuple from `Movie` must have value `'Star Wars'`.
2. The `producerC#` attribute of the `Movie` tuple must be the same certificate number as the `cert#` attribute in the `MovieExec` tuple. That is, these two tuples must refer to the same producer.



Queries Involving More Than One Relation

Disambiguating Attributes

Example 6.13: The two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

each have attributes name and address. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job.

```
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

Queries Involving More Than One Relation

Disambiguating Attributes

Example 6.14: While Example 6.13 asked for a star and an executive sharing an address, we might similarly want to know about two stars who share an address. The query is essentially the same, but now we must think of two tuples chosen from relation `MovieStar`, rather than tuples from each of `MovieStar` and `MovieExec`. Using tuple variables as aliases for two uses of `MovieStar`, we can write the query as

`MovieStar(name, address, gender, birthdate)`

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
      AND Star1.name < Star2.name;
```

We see in the `FROM` clause the declaration of two tuple variables, `Star1` and `Star2`; each is an alias for relation `MovieStar`. The tuple variables are used in the `SELECT` clause to refer to the `name` components of the two tuples. These aliases are also used in the `WHERE` clause to say that the two `MovieStar` tuples represented by `Star1` and `Star2` have the same value in their `address` components.

The second condition in the `WHERE` clause, `Star1.name < Star2.name`, says that the name of the first star precedes the name of the second star alphabetically. If this condition were omitted, then tuple variables `Star1` and `Star2` could both refer to the same tuple. We would find that the two tuple variables

Self Join

- A self join is a join in which a table is joined with itself
- To join a table itself means that each row of the table is combined with itself and with every other row of the table.
- The self join can be viewed as a join of two copies of the same table. The table is not actually copied, but SQL performs the command as though it were.

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and T2 are different table aliases for the same table.

Self Join

Study Table

sid	cid	since
S1	C1	2016
S2	C2	2017
S1	C2	2017

Find studentid who is enrolled in atleast two courses.

Select T1.sid from study as T1, study as T2 where T1.sid = T2.sid and T1.cid <> T2.cid

Self Join

Study Table

sid	cid	since
S1	C1	2016
S2	C2	2017
S1	C2	2017

Cross Product

T1		T2	
S1	C1	S1	C1
S1	C1	S2	C2
S1	C1	S1	C2
S2	C2	S1	C1
S2	C2	S2	C2
S2	C2	S1	C2
S1	C2	S1	C1
S1	C2	S2	C2
S1	C2	S1	C2

Find studentid who is enrolled in atleast two courses.

Select T1.sid from study as T1, study as T2 where T1.sid = T2.sid and T1.cid <> T2.cid

IN, NOT IN, ANY AND ALL OPERATOR

The SQL ANY and ALL Operators

The `ANY` and `ALL` operators allow you to perform a comparison between a single column value and a range of other values.

The SQL ANY Operator

The `ANY` operator:

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition

`ANY` means that the condition will be true if the operation is true for any of the values in the range.

ANY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

IN, NOT IN, ANY AND ALL OPERATOR

The SQL ALL Operator

The `ALL` operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with `SELECT`, `WHERE` and `HAVING` statements

`ALL` means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

IN, NOT IN, ANY AND ALL OPERATOR

The `IN` operator allows you to specify multiple values in a `WHERE` clause.

The `IN` operator is a shorthand for multiple `OR` conditions.

IN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

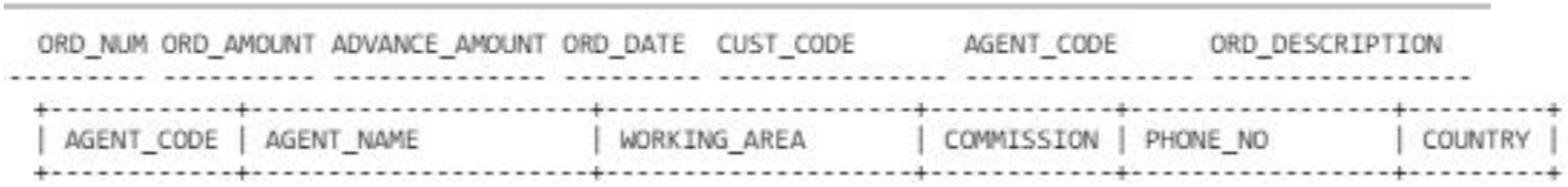
or:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

IN, NOT IN, ANY AND ALL OPERATOR

IN, NOT IN, ANY AND ALL OPERATOR

IN in subquery

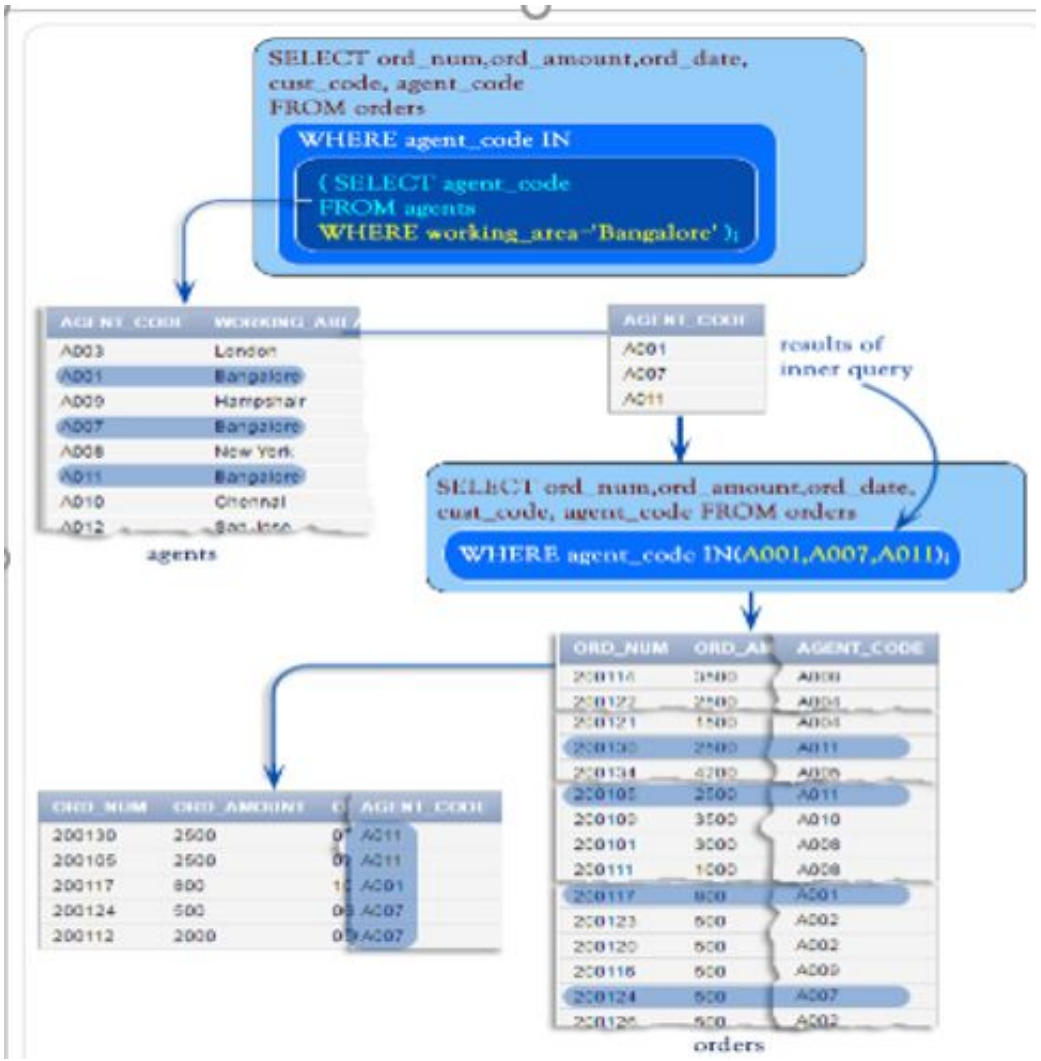


ORD_NUM	ORD_AMOUNT	ADVANCE_AMOUNT	ORD_DATE	CUST_CODE	AGENT_CODE	ORD_DESCRIPTION

AGENT_CODE	AGENT_NAME	WORKING_AREA	COMMISSION	PHONE_NO	COUNTRY

SELECT ord_num,ord_amount,ord_date, cust_code, agent_code FROM orders WHERE agent_code IN(SELECT agent_code FROM agents WHERE working_area='Bangalore');

IN, NOT IN, ANY AND ALL OPERATOR



IN, NOT IN, ANY AND ALL OPERATOR

```
SELECT ord_num,ord_amount,ord_date,
cust_code, agent_code FROM orders WHERE
agent_code NOT IN( SELECT agent_code
FROM agents WHERE
working_area='Bangalore');
```

IN, NOT IN, ANY AND ALL OPERATOR

- ‘ANY’ operator to compare a value with any values in a list.
- You must place an =, <>, >, <, <= or >= operator before ANY/ALL in your query.

Subqueries

```
SELECT *
FROM employee
WHERE salary > ANY (2000, 3000, 4000);
```

```
SELECT *
FROM employee
WHERE salary IN (2000, 3000, 4000);
```

But with the IN operator you cannot use =, <>, <, >, <=, or >=

IN, NOT IN, ANY AND ALL OPERATOR

SELECT empno, sal FROM emp WHERE sal > ALL (2000, 3000, 4000);

It will return result equivalent to query:

SELECT empno, sal FROM emp WHERE sal > 2000 AND sal > 3000 AND sal > 4000;

SELECT empno, sal FROM emp WHERE sal > ANY (2000, 3000, 4000);

SELECT empno, sal FROM emp WHERE sal > 2000 OR sal > 3000 OR sal > 4000;

IN, NOT IN, ANY AND ALL OPERATOR

```
mysql> Select * from Customers;
+-------------+----------+
| Customer_Id | Name     |
+-------------+----------+
|           1 | Rahul    |
|           2 | Yashpal  |
|           3 | Gaurav   |
|           4 | Virender |
+-------------+----------+
```

```
mysql> Select * from Reservations;
+------+-------------+------------+
| ID   | Customer_id | Day        |
+------+-------------+------------+
|    1 |           1 | 2017-12-30 |
|    2 |           2 | 2017-12-28 |
|    3 |           2 | 2017-12-29 |
|    4 |           1 | 2017-12-25 |
|    5 |           3 | 2017-12-26 |
+------+-------------+------------+
```

List the customerid from customers not equal to reservations customer id

SELECT customer_id from Customers WHERE customer_id <> ALL(Select customer_id from reservations);

```
+-------------+
| customer_id |
+-------------+
|           4 |
+-------------+
```

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

IN, NOT IN, ANY AND ALL OPERATOR

The following example uses ANY to check if any of the agent who belongs to the country 'UK'.

```
SELECT agent_code,agent_name , working_area, commission FROM agents
WHERE
agent_code=ANY
( SELECT agent_code FROM customer WHERE cust_country='UK');
```

IN, NOT IN, ANY AND ALL OPERATOR

Multiple Column Subqueries -You can write subqueries that return multiple columns.

Select all the rows where the order amount with the lowest price, group by agent code.

```
select ord_num, agent_code, ord_date,
ord_amount from orders where(agent_code,
ord_amount) IN (SELECT agent_code,
MIN(ord_amount) FROM orders GROUP BY
agent_code);
```

Queries Involving More Than One Relation

Union, Intersection, and Difference of Queries

Example 6.16: Suppose we wanted the names and addresses of all female movie stars who are also movie executives with a net worth over $10,000,000. Using the following two relations:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

we can write the query as in Fig. 6.5. Lines (1) through (3) produce a relation whose schema is `(name, address)` and whose tuples are the names and addresses of all female movie stars.

```
1)  (SELECT name, address
2)   FROM MovieStar
3)   WHERE gender = 'F')
4)       INTERSECT
5)  (SELECT name, address
6)   FROM MovieExec
7)   WHERE netWorth > 10000000);
```

Figure 6.5: Intersecting female movie stars with rich executives

Similarly, lines (5) through (7) produce the set of "rich" executives, those with net worth over $10,000,000. This query also yields a relation whose schema has the attributes `name` and `address` only. Since the two schemas are the same, we can intersect them, and we do so with the operator of line (4). □

Queries Involving More Than One Relation

Union, Intersection, and Difference of Queries

Example 6.17: In a similar vein, we could take the difference of two sets of persons, each selected from a relation. The query

```
(SELECT name, address FROM MovieStar)
    EXCEPT
(SELECT name, address FROM MovieExec);
```

gives the names and addresses of movie stars who are not also movie executives, regardless of gender or net worth. □

Queries Involving More Than One Relation

Union, Intersection, and Difference of Queries

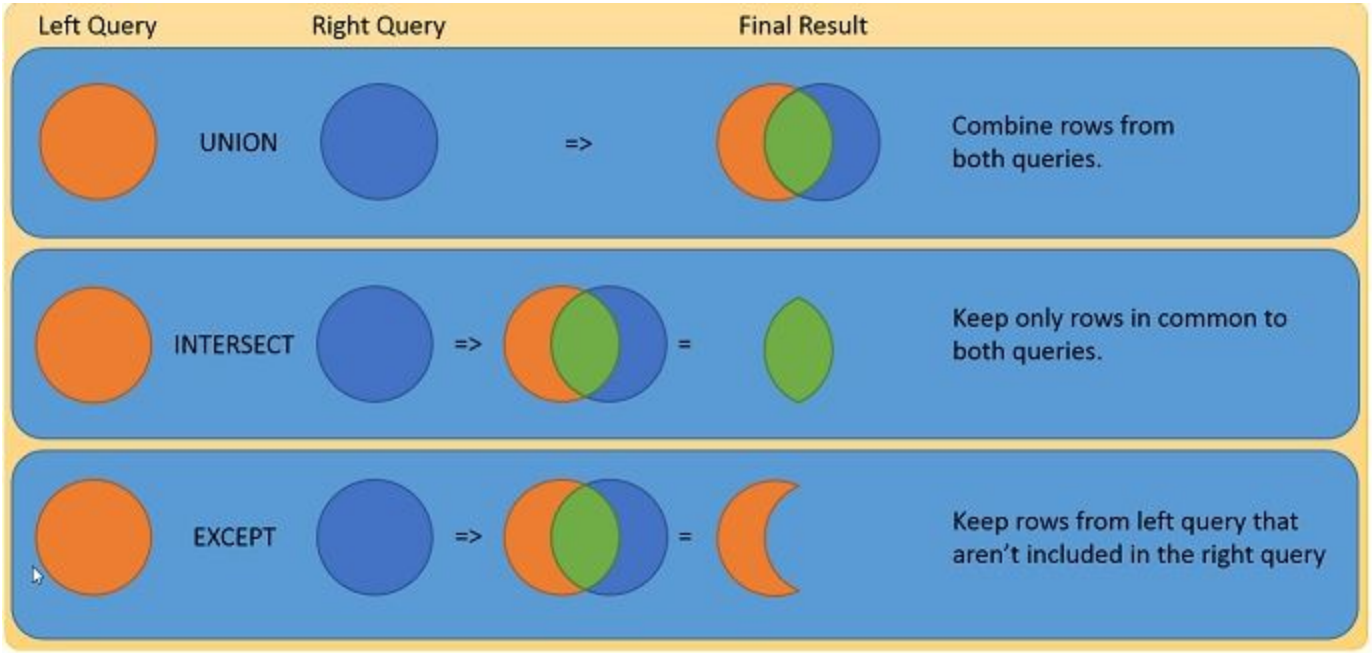
Example 6.18: Suppose we wanted all the titles and years of movies that appeared in either the `Movie` or `StarsIn` relation of our running example:

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

```
(SELECT title, year FROM Movie)
    UNION
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

The result would be all movies mentioned in either relation, with `title` and `year` as the attributes of the resulting relation. □

Union, Intersection, and Difference of Queries



Union, Intersection, and Difference of Queries

The UNION operator is used to combine the result-set of two or more SELECT statements without returning any duplicate rows.

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

Union, Intersection, and Difference of Queries

The SQL **INTERSECT** clause/operator is used to combine two SELECT statements.

- INTERSECT returns only common rows returned by the two SELECT statements.
- Just as with the UNION operator, the same rules apply when using the INTERSECT operator.

Union, Intersection, and Difference of Queries

- The SQL **EXCEPT** clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.
- This means EXCEPT returns only rows, which are not available in the second SELECT statement.

Subqueries

- A Query which is part of another query is called subquery.
- Subquery can have another subquery and so on.
- A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
- The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.
- For single row subqueries use the comparison operators, such as >, <, or =,<>,>=,<=.
- For multiple-row subqueries use the operator such as IN, ANY, ALL, EXISTS, NOT EXISTS

Subqueries

```
SELECT    select_list
FROM      table
WHERE     expr operator
                    (SELECT    select_list
                     FROM      table);
```

- The subquery (inner query) executes once before the main query (outer query) executes.
- The main query (outer query) use the subquery result.

Subqueries

- Find the producer of starwars.

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# =
4)      (SELECT producerC#
5)       FROM Movie
6)       WHERE title = 'Star Wars'
        );
```

Subqueries

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
```

Finding the producers of Harrison Ford's movies.

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# IN
4)      (SELECT producerC#
5)       FROM Movie
6)       WHERE (title, year) IN
7)           (SELECT movieTitle, movieYear
8)            FROM StarsIn
9)            WHERE starName = 'Harrison Ford'
             )
        );
```

```
SELECT name
FROM MovieExec, Movie, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford';
```

Figure 6.7: Finding the producers of Harrison Ford's movies

SQL Statements

Eliminating Duplicates

SELECT **DISTINCT** column1, column2, ...
FROM table_name;

The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table.

SELECT DISTINCT Country FROM Customers;

The following SQL statement lists the number of different (distinct) customer countries

SELECT COUNT(DISTINCT Country) FROM Customers;

Full-Relation Operations

Duplicates in Unions, Intersections, and Differences

- In previous operations of union, intersection and differences it eliminates duplicates ie bags are converted to sets.
- To prevent elimination of duplicates we must follow the operator UNION, INTERSECT or EXCEPT by keyword all.

Unlike the SELECT statement, which preserves duplicates as a default and only eliminates them when instructed to by the DISTINCT keyword, the union, intersection, and difference operations, which we introduced in Section 6.2.5, normally eliminate duplicates. That is, bags are converted to sets, and the set version of the operation is applied. In order to prevent the elimination of duplicates, we must follow the operator UNION, INTERSECT, or EXCEPT by the keyword ALL. If we do, then we get the bag semantics of these operators as was discussed in Section 5.3.2.

Full-Relation Operations

Duplicates in Unions, Intersections, and Differences

```
(SELECT title, year FROM Movie)
    UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

Now, a title and year will appear as many times in the result as it appears in each of the relations `Movie` and `StarsIn` put together. For instance, if a movie appeared once in the `Movie` relation and there were three stars for that movie

Full-Relation Operations
Duplicates in Unions, Intersections, and Differences

As for union, the operators INTERSECT ALL and EXCEPT ALL are intersection and difference of bags. Thus, if R and S are relations, then the result of expression

$$R \text{ INTERSECT ALL } S$$

is the relation in which the number of times a tuple t appears is the minimum of the number of times it appears in R and the number of times it appears in S.

The result of expression

$$R \text{ EXCEPT ALL } S$$

has tuple t as many times as the difference of the number of times it appears in R minus the number of times it appears in S, provided the difference is positive. Each of these definitions is what we discussed for bags in Section 5.3.2.

Full-Relation Operations

- SQL use five aggregation operators SUM, AVG, MIN, MAX and COUNT.
  - AVG – calculates the average of a set of values.
  - COUNT – counts rows in a specified table or view.
  - MIN – gets the minimum value in a set of values.
  - MAX – gets the maximum value in a set of values.
  - SUM – calculates the sum of values.

Full-Relation Operations

- SQL use five aggregation operators SUM, AVG, MIN, MAX and COUNT.
- COUNT(*) – Counts all the tuples in the relation that is constructed from FROM AND WHERE clause of the query.
- COUNT(DISTINCT x) – counts the number of distinct values in column x (eliminates duplicates)

Full-Relation Operations

- The GROUP BY statement groups rows that have the same values into summary rows.
- The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

Full-Relation Operations

Grouping and Aggregation in SQL

```
SELECT AVG(netWorth)
FROM MovieExec;
```

Note that there is no WHERE clause at all, so the keyword WHERE is properly omitted. This query examines the netWorth column of the relation

```
MovieExec(name, address, cert#, netWorth)
```

Full-Relation Operations
Grouping and Aggregation in SQL

Example 6.30: The following query:

```
SELECT COUNT(*)
FROM StarsIn;
```

counts the number of tuples in the `StarsIn` relation. The similar query:

```
SELECT COUNT(starName)
FROM StarsIn;
```

counts the number of values in the `starName` column of the relation. Since duplicate values are not eliminated when we project onto the `starName` column in SQL, this count should be the same as the count produced by the query with `COUNT(*)`.

If we want to be certain that we do not count duplicate values more than once, we can use the keyword `DISTINCT` before the aggregated attribute. as:

```
SELECT COUNT(DISTINCT starName)
FROM StarsIn;
```

Now, each star is counted once, no matter in how many movies they appeared. □

Full-Relation Operations

Grouping and Aggregation in SQL

Example 6.31: The problem of finding, from the relation

```
Movie(title, year, length, inColor, studioName, producerC#)
```

the sum of the lengths of all movies for each studio is expressed by

```
SELECT studioName, SUM(length)
FROM Movie
GROUP BY studioName;
```

We may imagine that the tuples of relation `Movie` are reorganized and grouped so that all the tuples for Disney studios are together, all those for MGM are together, and so on, as was suggested in Fig. 5.17. The sums of the length components of all the tuples in each group are calculated, and for each group, the studio name is printed along with that sum. □

```
SELECT studioName
FROM Movie
GROUP BY studioName;
```

```
SELECT DISTINCT studioName
FROM Movie;
```

Full-Relation Operations

Grouping and Aggregation in SQL

Example 6.32: Suppose we wish to print a table listing each producer's total length of film produced. We need to get information from the two relations

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerC# = cert#
GROUP BY name;
```

Figure 6.13: Computing the length of movies for each producer

Each MoviExec tuple is paired with Movie tuples for all the movies that producer. Then we group the selected tuples of this relation according to the name of the producer. Finally, we sum the length of the movies in each group.

Full-Relation Operations

Grouping and Aggregation in SQL

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

Example 6.33: Suppose we want to print the total film length for only those producers who made at least one film prior to 1930. We may append to Fig. 6.13 the clause

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

Figure 6.14: Computing the total length of film for early produ

Full-Relation Operations

Grouping and Aggregation in SQL

The SQL HAVING Clause

The `HAVING` clause was added to SQL because the `WHERE` keyword cannot be used with aggregate functions.

HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Full-Relation Operations

Grouping and Aggregation in SQL

Grouping, Aggregation, and Nulls

When tuples have nulls, there are a few rules we must remember:

- The value NULL is ignored in any aggregation. It does not contribute to a sum, average, or count, nor can it be the minimum or maximum in its column. For example, `COUNT(*)` is always a count of the number of tuples in a relation, but `COUNT(A)` is the number of tuples with non-NULL values for attribute A.

- On the other hand, NULL is treated as an ordinary value in a grouped attribute. For example, `SELECT a, AVG(b) FROM R GROUP BY a` will produce a tuple with NULL for the value of a and the average value of b for the tuples with a = NULL, if there is at least one tuple in R with a component NULL.

Summary :

- Subqueries are embedded queries inside another query. The embedded query is known as the inner query and the container query is known as the outer query.

- Sub queries are easy to use, offer great flexibility and can be easily broken down into single logical components making up the query which is very useful when Testing and debugging the queries.

- MySQL supports three types of subqueries, scalar, row and table subqueries.

- Scalar sub queries only return a single row and single column.
- Row sub queries only return a single row but can have more than one column.
- Table subqueries can return multiple rows as well as columns.

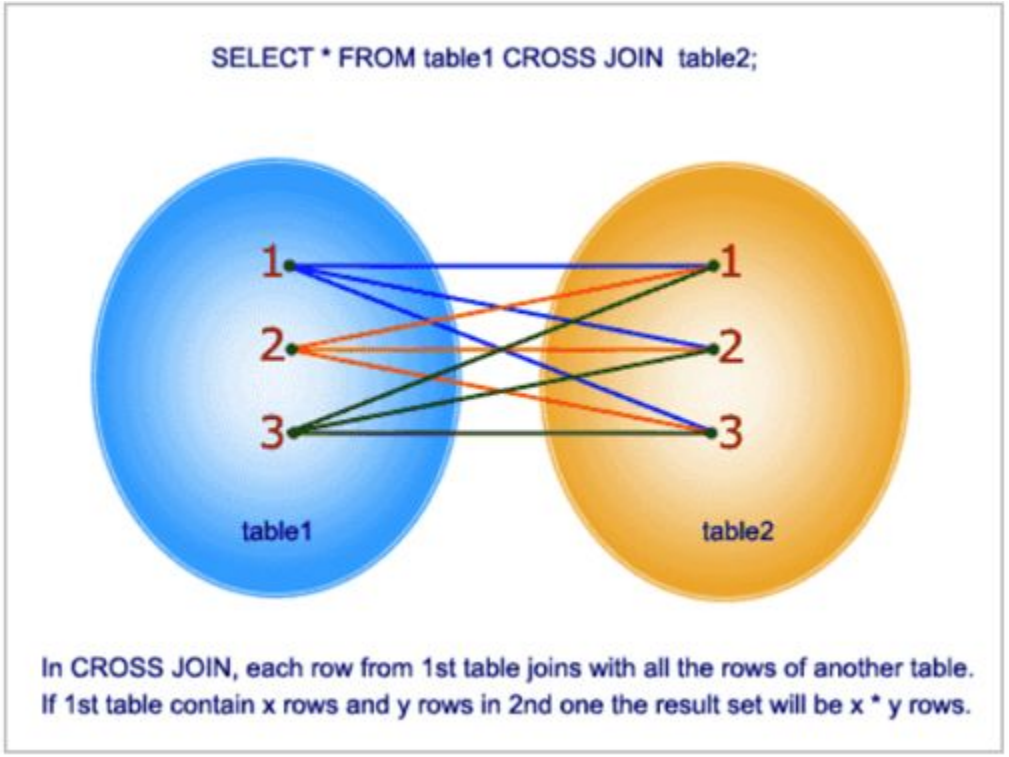
- Subqueries can also be used in INSERT, UPDATE and DELETE queries.

- For performance issues, when it comes to getting data from multiple tables, it is strongly recommended to use JOINs instead of subqueries.

SQL Join Expressions

The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN. This kind of result is called as Cartesian

```
SELECT *
FROM table1
CROSS JOIN table2;
```



SQL Join Expressions

Table1 - MatchScore

Player	Department_id	Goals
Franklin	1	2
Alan	1	3
Priyanka	2	2
Rajesh	3	5

Table2 - Departments

Department_id	Department_name
1	IT
2	HR
3	Marketing

```
SELECT * FROM MatchScore CROSS JOIN Departments
```

After executing this query , you will find the following result:

Player	Department_id	Goals	Depatment_id	Departı
Franklin	1	2	1	IT
Alan	1	3	1	IT
Priyanka	2	2	1	IT
Rajesh	3	5	1	IT
Franklin	1	2	2	HR
Alan	1	3	2	HR
Priyanka	2	2	2	HR
Rajesh	3	5	2	HR
Franklin	1	2	3	Marketir
Alan	1	3	3	Marketir
Priyanka	2	2	3	Marketir
Rajesh	3	5	3	Marketir

SQL Join Expressions

Movie(title, year, length, inColor, studioName, producerC#)

StarsIn(movieTitle, movieYear, starName)

```
Movie CROSS JOIN StarsIn;
```

```
Movie CROSS JOIN StarsIn;
```

and the result will be a nine-column relation with all the attributes of `Movie` and `StarsIn`. Every pair consisting of one tuple of `Movie` and one tuple of `StarsIn` will be a tuple of the resulting relation.

- The above is an Cartesian product of both the relations.
- theta-join is obtained with the keyword ON.
- We put JOIN between two relation names R and *S and follow them by ON and a condition. The meaning*
- of JOIN. . .ON is that the product of R x S is followed by a selection for whatever condition follows ON.

SQL Join Expressions

```
Movie JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

The result is again a nine-column relation with the obvious attribute names. However, now a tuple from `Movie` and one from `StarsIn` combine to form a tuple of the result only if the two tuples agree on both the title and year. As a

```
SELECT title, year, length, inColor, studioName,
       producerC#, starName
FROM Movie JOIN StarsIn ON
       title = movieTitle AND year = movieYear;
```

SQL Join Expressions

Natural Join

When we combine rows of two or more tables based on a common column between them, this operation is called joining. A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type.

Example 6.24: Suppose we want to compute the natural join of the relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

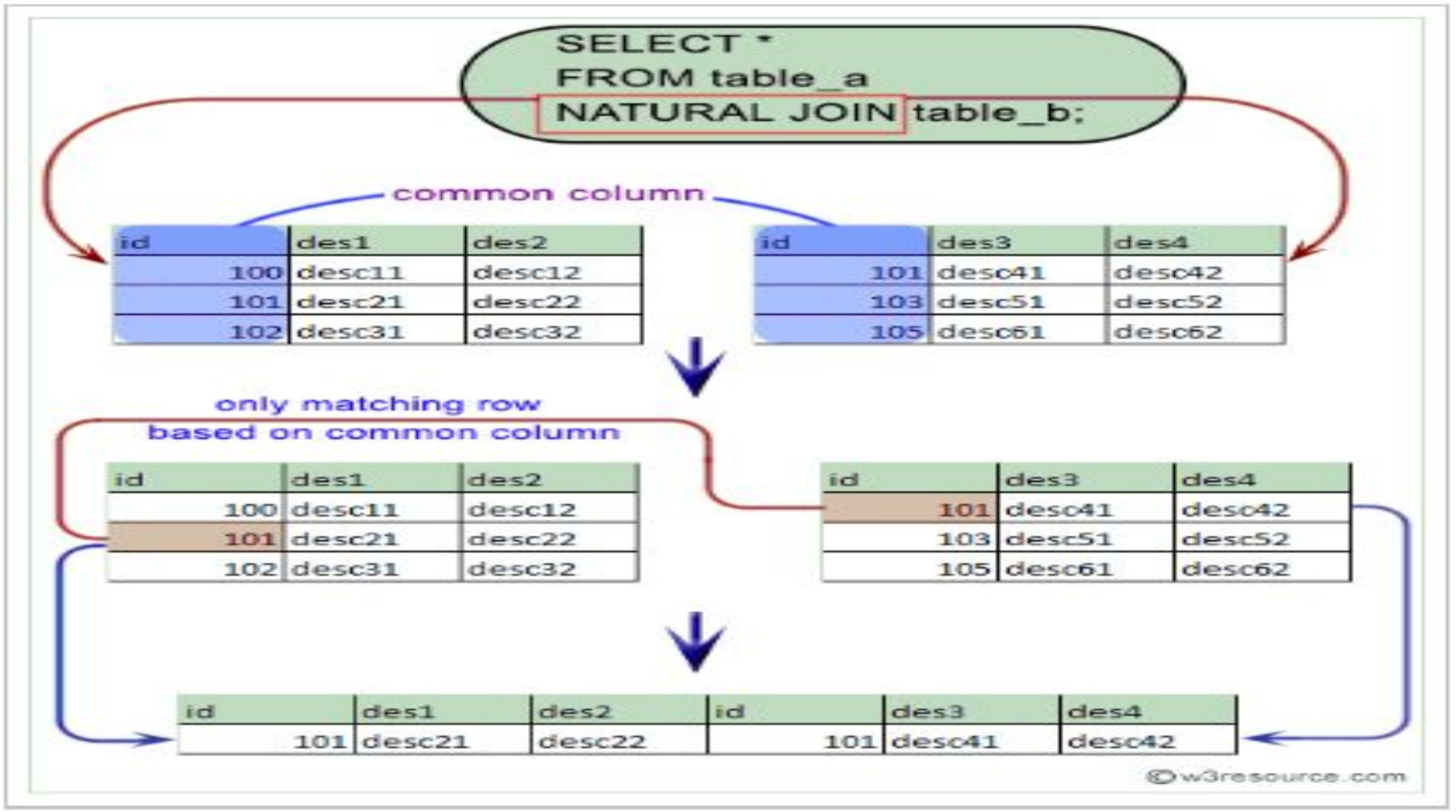
The result will be a relation whose schema includes attributes `name` and `address` plus all the attributes that appear in one or the other of the two relations. A tuple of the result will represent an individual who is both a star and an executive and will have all the information pertinent to either: a name, address, gender, birthdate, certificate number, and net worth. The expression

```
MovieStar NATURAL JOIN MovieExec;
```

SQL Join Expressions

Natural Join

```
SELECT [column_names | *]
FROM table_name1
NATURAL JOIN table_name2;
```



id	des1	des2
100	desc11	desc12
101	desc21	desc22
102	desc31	desc32

id	des3	des4
101	desc41	desc42
103	desc51	desc52
105	desc61	desc62

id	des1	des2	id	des3	des4
101	desc21	desc22	101	desc41	desc42

SQL Join Expressions

SQL FULL JOIN /Full outer Join

The SQL full join is the result of combination of both left and right outer join and the join tables have all the records from both tables. It puts NULL on the place of matches not found.

In the SQL outer JOIN all the content of the both tables are integrated together either they are matched or not.

```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

SQL Join Expressions

SQL Full outer join

Because this is a full outer join so all rows (both matching and non-matching) from both tables are included in the output. Here only one row of output displays values in all columns because there is only one match between table_A and table_B.

table_A

A	M
1	m
2	n
4	o

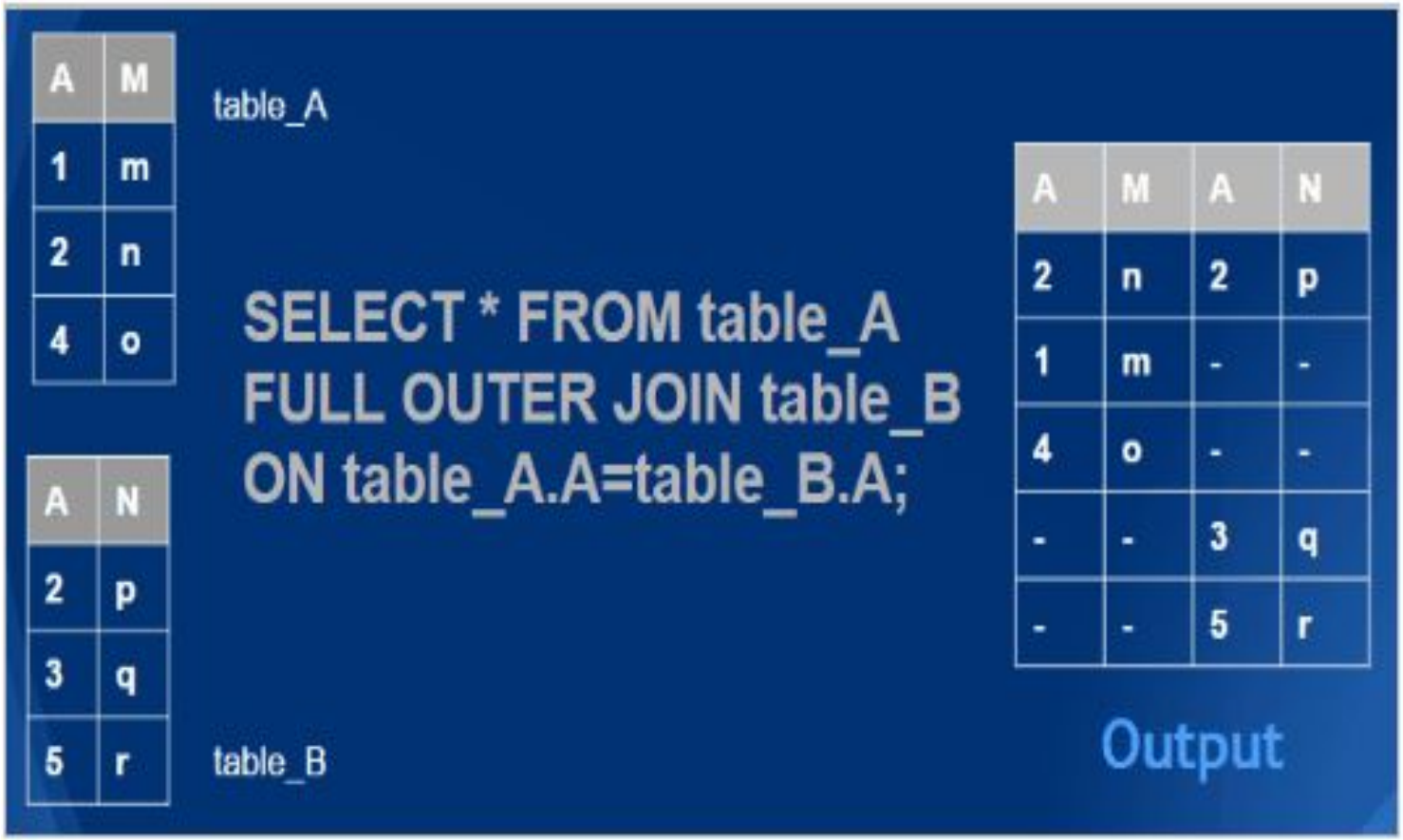
table_B

A	N
2	p
3	q
5	r

A	M	A	N
2	n	2	p
1	m	-	-
4	o	-	-
-	-	3	q
-	-	5	r

SQL Join Expressions

Outer Join



SQL Join Expressions

SQL LEFT JOIN

The SQL left join returns all the values from the left table and it also includes matching values from right table, if there are no matching join value it returns NULL.

BASIC SYNTAX FOR LEFT JOIN:

```
SELECT table1.column1, table2.column2....
FROM table1
LEFTJOIN table2
ON table1.column_field = table2.column_field;
```

SQL Join Expressions

SQL LEFT JOIN

CUSTOMER TABLE:

ID	NAME	AGE	SALARY
1	ARYAN	51	56000
2	AROHI	21	25000
3	VINEET	24	31000
4	AJEET	23	32000
5	RAVI	23	42000

ORDER TABLE:

O_ID	DATE	CUSTOMER_ID	AMOUNT
001	20-01-2012	2	3000
002	12-02-2012	2	2000
003	22-03-2012	3	4000
004	11-04-2012	4	5000

```
SQL SELECT ID, NAME, AMOUNT,DATE
FROM CUSTOMER
LEFT JOIN ORDER
ON CUSTOMER.ID = ORDER.CUSTOMER_ID;
```

ID	NAME	AMOUNT	DATE
1	ARYAN	NULL	NULL
2	AROHI	3000	20-01-2012
2	AROHI	2000	12-02-2012
3	VINEET	4000	22-03-2012
4	AJEET	5000	11-04-2012
5	RAVI	NULL	NULL

SQL Join Expressions

SQL RIGHT JOIN

The SQL right join returns all the values from the rows of right table. It also includes the matched values from left table but if there is no matching in both tables, it returns NULL.

Basic syntax for right join:

```
SELECT table1.column1, table2.column2.....
FROM table1
RIGHT JOIN table2
ON table1.column_field = table2.column_field;
```

SQL Join Expressions

CUSTOMER TABLE:

ID	NAME	AGE	SALARY
1	ARYAN	51	56000
2	AROHI	21	25000
3	VINEET	24	31000
4	AJEET	23	32000
5	RAVI	23	42000

ORDER TABLE:

DATE	O_ID	CUSTOMER_ID	AMOUNT
20-01-2012	001	2	3000
12-02-2012	002	2	2000
22-03-2012	003	3	4000
11-04-2012	004	4	5000

ID	NAME	AMOUNT	DATE
2	AROHI	3000	20-01-2012
2	AROHI	2000	12-02-2012
3	VINEET	4000	22-03-2012
4	AJEET	5000	11-04-2012

```
SQL> SELECT ID,NAME,AMOUNT,DATE
FROM CUSTOMER
RIGHT JOIN ORDER
ON CUSTOMER.ID = ORDER.CUSTOMER_ID;
```

SQL Join Expressions

Outer Join

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;

MovieStar NATURAL LEFT OUTER JOIN MovieExec;

MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

Exists Operator

The SQL EXISTS Operator

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns TRUE if the subquery returns one or more records.

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

- If `EXISTS (subquery)` returns at least 1 row, the result is TRUE.
- If `EXISTS (subquery)` returns no rows, the result is FALSE.
- If `NOT EXISTS (subquery)` returns at least 1 row, the result is FALSE.
- If `NOT EXISTS (subquery)` returns no rows, the result is TRUE.

Exists Operator

The SQL EXISTS Operator

The following SQL statement returns TRUE and lists the suppliers with a product price less than 20:

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20);
```

Exists Operator

Consider the following two relation "Customers" and "Orders".

Customers

customer_id	lname	fname	website
401	Singh	Dolly	abc.com
402	Chauhan	Anuj	def.com
403	Kumar	Niteesh	ghi.com
404	Gupta	Shubham	jkl.com
405	Walecha	Divya	abc.com
406	Jain	Sandeep	jkl.com
407	Mehta	Rajiv	abc.com
408	Mehra	Anand	abc.com

Orders

order_id	c_id	order_date
1	407	2017-03-03
2	405	2017-03-05
3	408	2017-01-18
4	404	2017-02-05

To fetch the first and last name of the customers who placed atleast one order.

```
SELECT fname, lname
FROM Customers
WHERE EXISTS
(SELECT * FROM Orders
 WHERE Customers.customer_id = Orders.c_id);
```

fname	lname
Shubham	Gupta
Divya	Walecha
Rajiv	Mehta
Anand	Mehra

View

- A view is a saved SQL query.
- A view can also be considered as a virtual table
- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table.
- The fields in a view are fields from one or more real tables in the database.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
- Provides data abstraction.

View

CREATE VIEW view_name AS SELECT column1, column2..... FROM table_name WHERE condition;

view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

View

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

View

CREATE VIEW DetailsView AS SELECT NAME, ADDRESS FROM StudentDetails WHERE S_ID < 5;

SELECT * FROM DetailsView;

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

View

Creating View from multiple tables

```
CREATE VIEW MarksView AS SELECT
StudentDetails.NAME,
StudentDetails.ADDRESS,
StudentMarks.MARKS FROM StudentDetails,
StudentMarks WHERE StudentDetails.ID =
StudentMarks.ID;
```

View

Views

What is a View?

A view is nothing more than a saved SQL query. A view can also be considered as a virtual table

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mike	3400	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

Id	Name	Salary	Gender	DeptName
1	John	5000	Male	HR
2	Mike	3400	Male	Payroll
3	Pam	6000	Female	IT
4	Todd	4800	Male	Admin
5	Sara	3200	Female	IT
6	Ben	4800	Male	HR

```
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

```
Create View vWEmployeesByDepartment
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

View

Advantages of views

Views can be used to reduce the complexity of the database schema

Views can be used as a mechanism to implement row and column level security.

Views can be used to present aggregated data and hide detailed data.

To modify a view - ALTER VIEW statement
To Drop a view - DROP VIEW vWName

View

```
CREATE VIEW <view-name> AS <view-definition>;
```

Example 6.45: Suppose we want to have a view that is a part of the

```
Movie(title, year, length, inColor, studioName, producerC#)
```

relation, specifically, the titles and years of the movies made by Paramount Studios. We can define this view by

```
1)  CREATE VIEW ParamountMovie AS
2)     SELECT title, year
3)     FROM Movie
4)     WHERE studioName = 'Paramount';
```

First, the name of the view is `ParamountMovie`, as we see from line (1). The attributes of the view are those listed in line (2), namely `title` and `year`. The definition of the view is the query of lines (2) through (4). □

View

Selection statement on view

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

View

It is also possible to write queries involving both views and base tables.

```
SELECT DISTINCT starName
FROM ParamountMovie, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

View

```
Movie(title, year, length, inColor, studioName, producerC#)
```

from which we get a producer's certificate number, and the relation

```
MovieExec(name, address, cert#, netWorth)
```

where we connect the certificate to the name. We may write:

```
CREATE VIEW MovieProd AS
    SELECT title, name
    FROM Movie, MovieExec
    WHERE producerC# = cert#;
```

View

We can query this view as if it were a stored relation. For instance, to find the producer of *Gone With the Wind*, ask:

```
SELECT name
FROM MovieProd
WHERE title = 'Gone With the Wind';
```

View

Modifying Views

- In limited circumstances it is possible to execute an insertion, deletion, or update to a view.
- For sufficiently simple views, called *updatable views, it is possible to translate the m*odification of the view into an equivalent modification on a base table.

View

Modifying Views

Is possible if :

- The WHERE clause must not involve R in a subquery.
- The list in the SELECT clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with NULL values or the proper default and have a tuple of the base relation that will yield the inserted tuple of the view.

View

Example 6.49: Suppose we try to insert into view ParamountMovie of Example 6.45 a tuple like:

```
INSERT INTO ParamountMovie
VALUES('Star Trek', 1979);
```

View ParamountMovie almost meets the SQL updatability conditions, since the view asks only for some components of some tuples of one base table:

```
Movie(title, year, length, inColor, studioName, producerC#)
```

The only problem is that since attribute studioName of Movie is not an attribute of the view, the tuple we insert into Movie would have NULL rather than 'Paramount' as its value for studioName. That tuple does not meet the condition that its studio be Paramount.

View

Thus, to make the view ParamountMovie updatable, we shall add attribute studioName to its SELECT clause, even though it is obvious to us that the studio name will be Paramount. The revised definition of view ParamountMovie is:

```
CREATE VIEW ParamountMovie AS
   SELECT studioName, title, year
   FROM Movie
   WHERE studioName = 'Paramount';
```

Then, we write the insertion into updatable view ParamountMovie as:

```
INSERT INTO ParamountMovie
VALUES('Paramount', 'Star Trek', 1979);
```

View

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
'Star Trek'	1979	0	NULL	'Paramount'	NULL

View

```
DELETE FROM ParamountMovie
WHERE title LIKE '%Trek%';
```

This deletion is translated into an equivalent deletion on the `Movie` base table: the only difference is that the condition defining the view `ParamountMovie` is added to the conditions of the `WHERE` clause.

```
DELETE FROM Movie
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

View

```
UPDATE ParamountMovie
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

is turned into the base-table update

```
UPDATE Movie
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
    studioName = 'Paramount';
```

□

```
DROP VIEW ParamountMovie;
```

Comparisons Involving NULL and Three-Valued Logic

- SQL has various rules for dealing with NULL values.

- NULL is used to represent a missing value, but that it usually has one of three different interpretations

  - **value unknown** (value exists but is not known, or it is not known whether or not the value exists),
  - **value not available** (value exists but is purposely withheld)
  - **value not applicable** (the attribute does not apply to this tuple or is undefined for this tuple).

Comparisons Involving NULL and Three-Valued Logic

- SQL has various rules for dealing with NULL values.

- NULL is used to represent a missing value, but that it usually has one of three different interpretations

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database. An example of the other case of unknown would be NULL for a person's home phone because it is not known whether or not the person has a home phone.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute LastCollegeDegree would be

Comparisons Involving NULL and Three-Valued Logic

- When a record with NULL in one of its attributes is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE).
- Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE.
- It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used.

Comparisons Involving NULL and Three-Valued Logic

In WHERE clauses, we must be prepared for the possibility that a component of some tuple we are examining will be NULL. There are two important rules to remember when we operate upon a NULL value.

1. When we operate on a NULL and any value, including another NULL, using an arithmetic operator like × or +, the result is NULL.

2. When we compare a NULL value and any value, including another NULL, using a comparison operator like = or >, the result is UNKNOWN. The value UNKNOWN is another truth-value, like TRUE and FALSE; we shall discuss how to manipulate truth-value UNKNOWN shortly.

Comparisons Involving NULL and Three-Valued Logic

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Figure 6.2: Truth table for three-valued logic

Comparisons Involving NULL and Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Comparisons Involving NULL and Three-Valued Logic

Pitfalls Regarding Nulls

It is tempting to assume that NULL in SQL can always be taken to mean "a value that we don't know but that surely exists." However, there are several ways that intuition is violated. For instance, suppose x is a component of some tuple, and the domain for that component is the integers. We might reason that $0 * x$ surely has the value 0, since no matter what integer x is, its product with 0 is 0. However, if x has the value NULL, rule (1) of Section 6.1.5 applies; the product of 0 and NULL is NULL. Similarly, we might reason that $x - x$ has the value 0, since whatever integer x is, its difference with itself is 0. However, again rule (1) applies and the result is NULL.

Comparisons Involving NULL and Three-Valued Logic

Example 6.10: Suppose we ask about our running-example relation

```
Movie(title, year, length, inColor, studioName, producerC#)
```

the following query:

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

Intuitively, we would expect to get a copy of the `Movie` relation, since each movie has a length that is either 120 or less or that is greater than 120.

However, suppose there are `Movie` tuples with `NULL` in the `length` component. Then both comparisons `length <= 120` and `length > 120` evaluate to `UNKNOWN`. The `OR` of two `UNKNOWN`'s is `UNKNOWN`, by Fig. 6.2. Thus, for any tuple with a `NULL` in the `length` component, the `WHERE` clause evaluates to `UNKNOWN`. Such a tuple is *not* returned as part of the answer to the query. As a result, the true meaning of the query is "find all the `Movie` tuples with non-`NULL` lengths."

SQL queries & Subqueries

Eid	Ename	Dept	Salary
1	Ram	HR	10000
2	Anu	MRKT	20000
3	Rahul	HR	30000
4	Raju	MRKT	40000
5	Vani	IT	50000

(1) Write a SQL query to display max sal from emp.

```
Select max (salary) from emp
```

(2) Write a SQL query to display employee name who is taking max salary. → need to write subquery. or nested query

```
Select Ename from emp where salary = (Select max (salary) from emp)
```

→ Outer query.

inner query

(3) Write a SQL query to display second highest salary from emp table.

```
Select max (salary) emp where salary <> (select max (salary) from emp)
```

50000 ( ) 50000

(4) Write a SQL query to display employee name who is taking second highest salary.

```
Select ename from emp where salary = ( select
max (salary) emp where salary <> (select max (salary)
from emp)
```

(5) Write a query to display all the dept names along with no. of emps working in that.

You can write only the dept ∵ group by has dept for any other attribute we need to use aggregate fn.

```
Select dept from emp group by dept.
Select dept, Count(*) from emp group by dept
```

→ dept

O/p HR 2
MRKT 2
IT 1

O/p HR
HR
MRKT
MRKT
IT

HR - 2
MRKT - 2
IT - 1

max
min
sum
Avg
count

(6) write a query to display all the dept names where no. of emps are less than 2.

along with groupby we need to use having.

* Select dept from emp group by dept having count(*) <2;

Select ename ~~where~~ from emp where dept In (*);

↓ above query.

↓

To find name of person working in dept where no. of emp less than 2.

Part-5

(7) write a query to display highest salary department wise and name of emp who is taking that salary.

Select dept, max(salary) from emp group by dept;

Select ename from emp where salary In

(select max(salary) from emp groupby dept)

Part-6

(8) Find detail of emp whose addr is either Delhi or Mumbai or pune.

Select * from emp where Address in

('Delhi', 'mumbai', 'Pune');

(9) find the name of emp who are working on a project.

Select Ename from Emp where Eid In (

(Select distinct (Eid) from project).

correlated nested query - Exist & not exists.

(10) Find the detail of emp who is working on atleast one project.

[not exists]

Select * from emp where Eid exists

[Select Eid from project where Emp.Eid = project.Eid)

Emp

Eid	Ename	Addr
1	Rani	[illegible]
2	Varun	Delhi
3	Nitin	Pune
4	Robin	Bangalore
5	Raju	mumbai

Project

Eid	Pid	Pname	loc.
1	P1	IOT	Bangalore
5	P2	Mobile	Delhi
3	P3	Cloud	chd
4	P4	Image	chennai

Correlated Subquery (Synchronized query)

→ It is a subquery that uses values from outer query.

→ Top to down approach.
(outer) (Inner)

Find all employee details who works in a department.

Emp

Eid	name	Address
1	A	Delhi
2	B	Pune.
3	A	Chd
4	B	Delhi
5	E	Pune
6	D	mumbai
7	E	Hyd.

Dept

Did	Dname	Eid.
D1	HR.	1
D2	IT	2
D3	mgr.	3
D4	Testing.	4.

```
Select * from emp where
    exists (select * from dept where
                dept.Eid = Emp.Eid)
```

Outer & Inner Query is related with Eid.

Each row of outer query (Emp table) is compared with all rows of Inner table (dept table).

Exists returns True or false.

If any one row with condition exists it returns true.

Ex

(1) first row of emp table compared to all rows of dept table.

(2) Second row of emp table compared to all rows of dept table, ...

~~P/B~~ Ex

1	A	Delhi

		Cells
D1	HR	1 = 1
D2	IT	2 = 1
D3	MART	3 = 1
D4	Testing	4 = 1

O/P

Emp

1	A	Delhi
2	B	Pune
3	A	Chd
4	B	Delhi

(Inner) (outer)
Nested Subquery. – Bottom up,
(outer) (Inner)
Correlated Subquery – Top down.

Joins – Cross product + condition.

Emp

Eid	name.
1	A
2	B
3	C
4	D
5	E

Dept

Deptno	name	Eid.
D1	IT	1
D2	HR	2
D3	MRKT	3.

Select all details of employee who works in any department.

Nested Subquery

(1) Select * from emp where Eid in
(Select eid from dept)

Correlated subquery

(1) Select * from emp where exists (Select eid from dept where emp.eid = dept.eid)

Join

(1) Select attributes from emp, dept where emp.eid = dept.eid.