Unit 2 Relational Algebra and Calculus, SQL Query and Triggers

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. There are two kinds of query languages − relational algebra and relational calculus.

Relational Algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

We will discuss all these operations in the following sections.

Select Operation (σ)

It selects tuples that satisfy the given predicate from a relation.

Notation − $\sigma_p(r)$

Where **σ** stands for selection predicate and **r** stands for relation. ***p*** is prepositional logic formula which may use connectors like **and, or,** and **not**. These terms may use relational operators like − =, ≠, ≥, <, >, ≤

For example −

```
σsubject = "database"(Books)
```

Output − Selects tuples from books where subject is 'database'.

```
σsubject = "database" and price = "450"(Books)
```

Output − Selects tuples from books where subject is 'database' and 'price' is 450.

```
σsubject = "database" and price = "450" or year > "2010"(Books)
```

Output − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

Project Operation (∏)

It projects column(s) that satisfy a given predicate.

Notation − $\prod_{A1, A2,\ldots An}$ (r)

Where A_1, A_2, ….A_n are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

For example −

```
∏subject, author (Books)
```

Selects and projects columns named as subject and author from the relation Books.

Union Operation (∪)

It performs binary union between two given relations and is defined as −

```
r ∪ s = { t | t ∈ r or t ∈ s}
```

Notation − r U s

Where **r** and **s** are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold −

- **r**, and **s** must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

```
∏ author (Books) ∪ ∏ author (Articles)
```

Output − Projects the names of the authors who have either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation − **r** − **s**

Finds all the tuples that are present in **r** but not in **s**.

```
∏ author (Books) − ∏ author (Articles)
```

Output − Provides the name of authors who have written books but not articles.

Cartesian Product (X)

Combines information of two different relations into one.

Notation − r X s

Where **r** and **s** are relations and their output will be defined as −

r X s = { q t | q ∈ r and t ∈ s}

```
σ_author = 'tutorialspoint'(Books X Articles)
```

Output − Yields a relation, which shows all the books and articles written by tutorialspoint.

Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter **rho** *ρ*.

Notation − ρ_x (E)

Where the result of expression **E** is saved with name of **x**.

Additional operations are −

- Set intersection
- Assignment
- Natural join

Joins

We understand the benefits of taking a Cartesian product of two relations, which gives us all the possible tuples that are paired together. But it might not be feasible for us in certain cases to take a Cartesian product where we encounter huge relations with thousands of tuples having a considerable large number of attributes.

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

We will briefly describe various join types in the following sections.

Theta (θ) Join

Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol **θ**.

Notation

$R1 \bowtie_{\theta} R2$

R1 and R2 are relations having attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that the attributes don't have anything in common, that is R1 ∩ R2 = Φ.

Theta join can use all kinds of comparison operators.

Student		
SID	**Name**	**Std**
101	Alex	10
102	Maria	11

Subjects	
Class	**Subject**
10	Math
10	English
11	Music
11	Sports

Student_Detail −

$STUDENT \bowtie_{Student.Std = Subject.Class} SUBJECT$

Student_detail

SID	Name	Std	Class	Subject
101	Alex	10	10	Math
101	Alex	10	10	English
102	Maria	11	11	Music
102	Maria	11	11	Sports

Equijoin

When Theta join uses only **equality** comparison operator, it is said to be equijoin. The above example corresponds to equijoin.

Natural Join (⋈)

Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does. We can perform a Natural Join only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain.

Natural join acts on those matching attributes where the values of attributes in both the relations are same.

Courses		
CID	**Course**	**Dept**
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HoD

Dept	Head
CS	Alex
ME	Maya
EE	Mira

Courses ⋈ HoD			
Dept	**CID**	**Course**	**Head**
CS	CS01	Database	Alex
ME	ME01	Mechanics	Maya
EE	EE01	Electronics	Mira

Outer Joins

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

Left Outer Join(R ⟕ S)

All the tuples from the Left relation, R, are included in the resulting relation. If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

Left	
A	**B**

100	Database
101	Mechanics
102	Electronics

Right	
A	**B**
100	Alex
102	Maya
104	Mira

Courses ⟕ HoD			
A	**B**	**C**	**D**
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya

Right Outer Join: (R ⟖ S)

All the tuples from the Right relation, S, are included in the resulting relation. If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

Courses ⟖ HoD

A	B	C	D
100	Database	100	Alex
102	Electronics	102	Maya
---	---	104	Mira

Full Outer Join: (R ⟗ S)

All the tuples from both participating relations are included in the resulting relation. If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

Courses ⟗ HoD			
A	**B**	**C**	**D**
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya
---	---	104	Mira

Relational Calculus

In contrast to Relational Algebra, Relational Calculus is a non-procedural query language, that is, it tells what to do but never explains how to do it.

Relational calculus exists in two forms −

Tuple Relational Calculus (TRC)

Filtering variable ranges over tuples

Notation − {T | Condition}

Returns all tuples T that satisfies a condition.

For example −

```
{ T.name |  Author(T) AND T.article = 'database' }
```

Output − Returns tuples with 'name' from Author who has written article on 'database'.

TRC can be quantified. We can use Existential (∃) and Universal Quantifiers (∀).

For example −

```
{ R| ∃T   ∈ Authors(T.article='database' AND R.name=T.name)}
```

Output − The above query will yield the same result as the previous one.

Domain Relational Calculus (DRC)

In DRC, the filtering variable uses the domain of attributes instead of entire tuple values (as done in TRC, mentioned above).

Notation −

$\{ a_1, a_2, a_3, ..., a_n \mid P (a_1, a_2, a_3, ... ,a_n)\}$

Where a1, a2 are attributes and **P** stands for formulae built by inner attributes.

For example −

```
{< article, page, subject > |  ∈ TutorialsPoint ∧ subject = 'database'}
```

Output − Yields Article, Page, and Subject from the relation TutorialsPoint, where subject is database.

Just like TRC, DRC can also be written using existential and universal quantifiers. DRC also involves relational operators.

The expression power of Tuple Relation Calculus and Domain Relation Calculus is equivalent to Relational Algebra.

SQL Queries

SQL SELECT Statement:

```
SELECT column1, column2....columnN
FROM table_name;
```

SQL DISTINCT Clause:

```
SELECT DISTINCT column1, column2....columnN
FROM table_name;
```

SQL WHERE Clause:

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION;
```

SQL AND/OR Clause:

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION-1 {AND|OR} CONDITION-2;
```

SQL IN Clause:

```
SELECT column1, column2....columnN
FROM table_name
WHERE column_name IN (val-1, val-2,...val-N);
```

SQL BETWEEN Clause:

```
SELECT column1, column2....columnN
FROM table_name
WHERE column_name BETWEEN val-1 AND val-2;
```

SQL LIKE Clause:

```
SELECT column1, column2....columnN
FROM table_name
WHERE column_name LIKE { PATTERN };
```

SQL ORDER BY Clause:

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION
ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY Clause:

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name;
```

SQL COUNT Clause:

```
SELECT COUNT(column_name)
FROM table_name
WHERE CONDITION;
```

SQL HAVING Clause:

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQL CREATE TABLE Statement:

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

SQL DROP TABLE Statement:

```
DROP TABLE table_name;
```

SQL CREATE INDEX Statement:

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQL DROP INDEX Statement:

```
ALTER TABLE table_name
DROP INDEX index_name;
```

SQL DESC Statement:

```
DESC table_name;
```

SQL TRUNCATE TABLE Statement:

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE Statement:

```
ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_ype};
```

SQL ALTER TABLE Statement (Rename):

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQL INSERT INTO Statement:

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQL UPDATE Statement:

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE CONDITION ];
```

SQL DELETE Statement:

```
DELETE FROM table_name
WHERE {CONDITION};
```

SQL CREATE DATABASE Statement:

```
CREATE DATABASE database_name;
```

SQL DROP DATABASE Statement:

```
DROP DATABASE database_name;
```

SQL USE Statement:

```
USE DATABASE database_name;
```

SQL COMMIT Statement:

```
COMMIT;
```

SQL ROLLBACK Statement:

```
ROLLBACK;
```

Nested queries

- A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.
- A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY.
- The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  6 | Komal    |  22 | MP        |  4500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+
```

Now, let us check the following subquery with a SELECT statement.

```
SQL> SELECT *

  FROM CUSTOMERS

  WHERE ID IN (SELECT ID

      FROM CUSTOMERS

      WHERE SALARY > 4500) ;
```

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

Example

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL> INSERT INTO CUSTOMERS_BKP
   SELECT * FROM CUSTOMERS
   WHERE ID IN (SELECT ID
   FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

Example

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> UPDATE CUSTOMERS

  SET SALARY = SALARY * 0.25

  WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP

    WHERE AGE >= 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |   125.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  2125.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

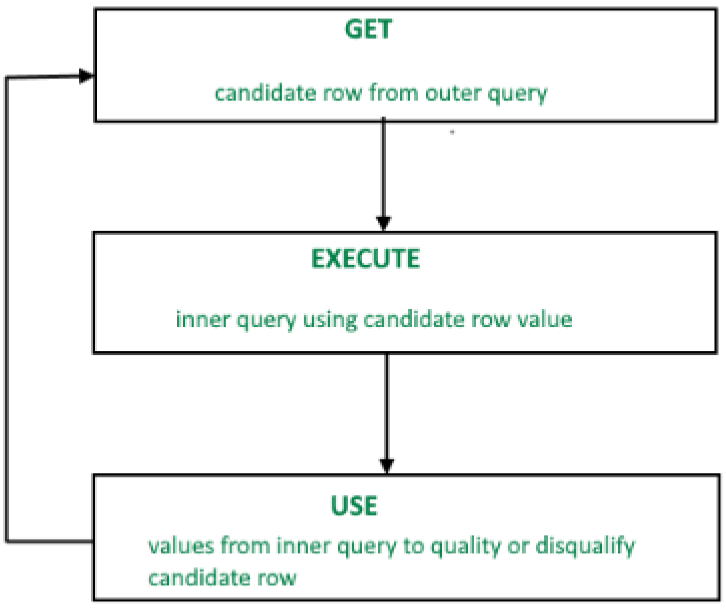
```
SQL> DELETE FROM CUSTOMERS

  WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP

    WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  3 | kaushik  |  23 | Kota    |  2000.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

SQL Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.


GET
candidate row from outer query
EXECUTE
inner query using candidate row value
USE
values from inner query to quality or disqualify candidate row

A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, r **DELETE** statement.

```
SELECT column1, column2, ....
FROM table1 outer
WHERE column1 operator
                (SELECT column1, column2
                 FROM table2
                 WHERE expr1 =
                         outer.expr2);
```

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

Nested Subqueries Versus Correlated Subqueries :

With a normal nested subquery, the inner **SELECT** query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

NOTE: You can also use the **ANY** and **ALL** operator in a correlated subquery.

EXAMPLE of Correlated Subqueries: Find all the employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
 FROM employees outer
 WHERE salary >
               (SELECT AVG (salary)
                FROM employees
                WHERE department_id = outer.department_id);
```

Other use of correlation is in **UPDATE** and **DELETE**

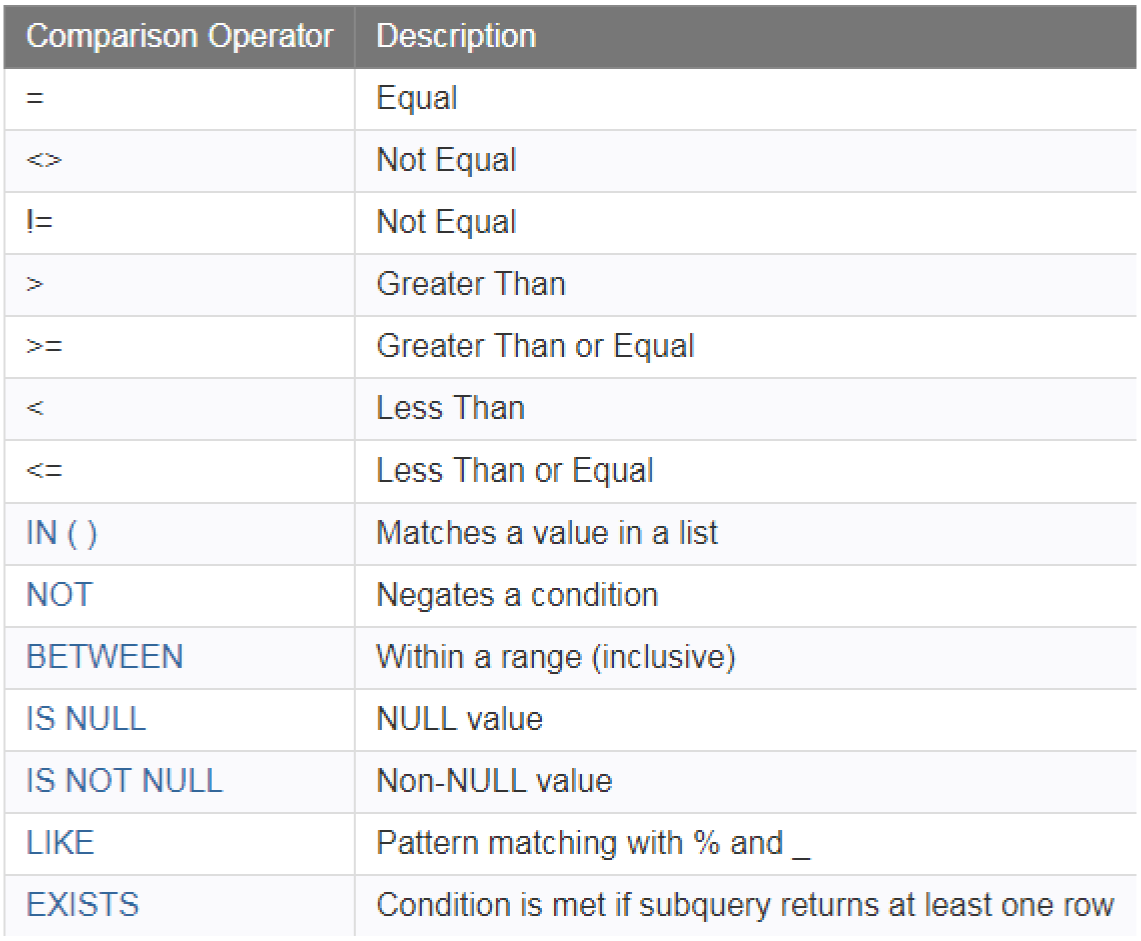
CORRELATED UPDATE:

```
UPDATE table1 alias1
 SET column = (SELECT expression
              FROM table2 alias2
              WHERE alias1.column =
                    alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

Set Comparison Operators

Comparison operators are used in the WHERE clause to determine which records to select. Here is a list of the comparison operators that you can use in SQL:

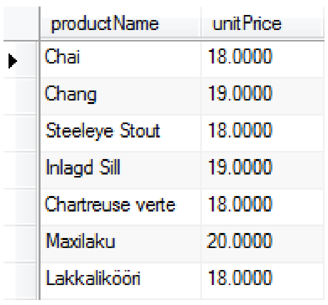


Comparison Operator	Description
=	Equal
<>	Not Equal
!=	Not Equal
>	Greater Than
>=	Greater Than or Equal
<	Less Than
<=	Less Than or Equal
IN ( )	Matches a value in a list
NOT	Negates a condition
BETWEEN	Within a range (inclusive)
IS NULL	NULL value
IS NOT NULL	Non-NULL value
LIKE	Pattern matching with % and _
EXISTS	Condition is met if subquery returns at least one row

SQL IN operator

For example, to select product whose unit price is $18, $19 or $20, you can perform the following query:

```
SELECT     productName, unitPrice

FROM  products

WHERE    unitPrice IN (18 , 19, 20)
```



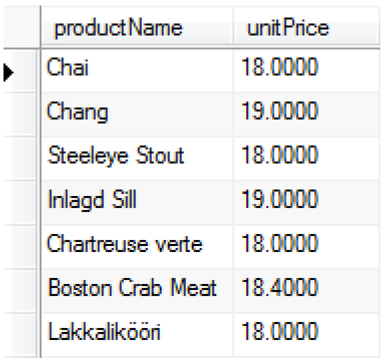
productName	unitPrice
Chai	18.0000
Chang	19.0000
Steeleye Stout	18.0000
Inlagd Sill	19.0000
Chartreuse verte	18.0000
Maxilaku	20.0000
Lakkalikööri	18.0000

SQL BETWEEN operator

The following query selects product whose unit price is from $18 to $19:

```
SELECT     productName, unitPrice
```

```
FROM      products

WHERE    unitPrice BETWEEN 18 AND 19;
```

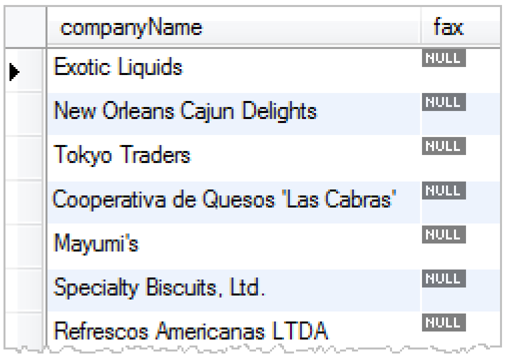


productName	unitPrice
Chai	18.0000
Chang	19.0000
Steeleye Stout	18.0000
Inlagd Sill	19.0000
Chartreuse verte	18.0000
Boston Crab Meat	18.4000
Lakkalikööri	18.0000

Is Null

You can use IS NULL to check if the supplier does not have a fax so that you can communicate with them via an alternative communication channel. The following query accomplishes this:

```
SELECT     companyName, fax
FROM       suppliers
WHERE      fax IS NULL;
```



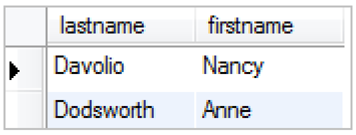
companyName	fax
Exotic Liquids	NULL
New Orleans Cajun Delights	NULL
Tokyo Traders	NULL
Cooperativa de Quesos 'Las Cabras'	NULL
Mayumi's	NULL
Specialty Biscuits, Ltd.	NULL
Refrescos Americanas LTDA	NULL

SQL LIKE operator examples

SQL LIKE operator with percentage wildcard (%) examples

Suppose you want to find employee whose last name starts with the letter D, you can use the following query.

```
SELECT     lastname, firstname

FROM      employees

WHERE    lastname LIKE 'D%'
```



lastname	firstname
Davolio	Nancy
Dodsworth	Anne

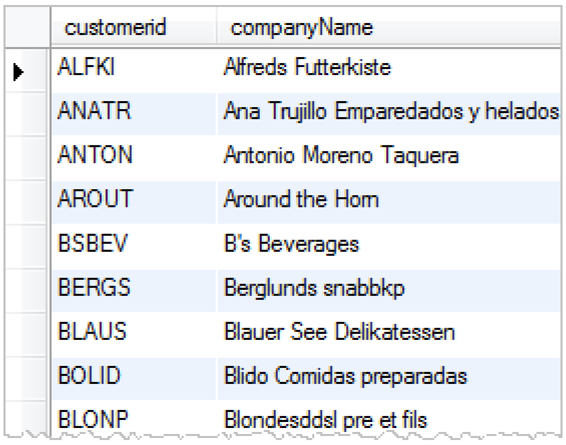
SQL EXISTS Operator examples

The EXISTS operator checks if a subquery returns any rows. The following illustrates the syntax of the EXISTS operator:

```
WHERE EXISTS (subquery)
```

You can use the EXISTS operator to find a customer who has ordered products. For each customer in the customers table, you check if there is at least one order exists in the orders table.

```
SELECT    customerid, companyName

FROM    customers

WHERE    EXISTS

(  SELECT    orderid       FROM          orders       WHERE          orders.customerid = customers.customerid);
```



customerid	companyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquera
AROUT	Around the Horn
BSBEV	B's Beverages
BERGS	Berglunds snabbkp
BLAUS	Blauer See Delikatessen
BOLID	Blido Comidas preparadas
BLONP	Blondesddsl pre et fils

Introduction to SQL aggregate functions

An aggregate function allows you to perform a calculation on a set of values to return a single scalar value. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

The following are the most commonly used SQL aggregate functions:

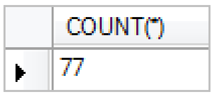
- AVG – calculates the average of a set of values.

- 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.

COUNT function example

To get the number of products in the products table, you use the COUNT function as follows:

```
SELECT  COUNT (*)

FROM     products;
```

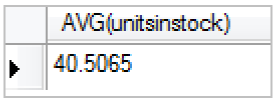


COUNT(*)
77

AVG function example

To calculate the average units in stock of the products, you use the AVG function as follows:

```
SELECT AVG (unitsinstock)
FROM    products;
```

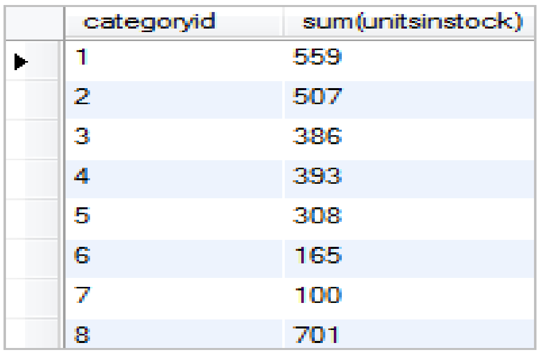


AVG(unitsinstock)
40.5065

SUM function example

To calculate the sum of units in stock by product category, you use the SUM function with the GROUP BY clause as the following query:

```
SELECT categoryid, SUM(unitsinstock)
FROM    products
GROUP BY categoryid;
```

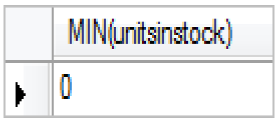


categoryid	sum(unitsinstock)
1	559
2	507
3	386
4	393
5	308
6	165
7	100
8	701

MIN function example

To get the minimum units in stock of products in the products table, you use the MIN function as follows:

```
SELECT MIN (unitsinstock)
FROM products;
```

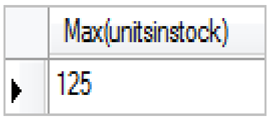


	MIN(unitsinstock)
▶	0

MAX function example

To get the maximum units in stock of products in the products table, you use the MAX function as shown in the following query:

```
SELECT MAX (unitsinstock)
FROM  products;
```



	Max(unitsinstock)
▶	125

Embedded SQL

Embedded SQL is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL. Embedded SQL statements are SQL statements written in line with the program source code , of the host language. The embedded SQL statements are parsed by an embedded SQL preprocessor and replaced by host-language calls to a code library. The output from the preprocessor is then compiled by the host compiler. This allows programmers to embed SQL statements in programs written in any number of languages such as C/C++, COBOL and Fortran. This differs from SQL-derived programming languages that don't go through discrete preprocessors, such as PL/SQL and T-SQL.

The SQL standards committee defined the embedded SQL standard in two steps: a formalism called **Module Language** was defined, then the embedded SQL standard was derived from Module Language.[1] The SQL standard defines embedding of SQL as *embedded SQL* and the language in which SQL queries are embedded is referred to as the *host language*. A popular host language is C. Host language C and embedded SQL, for example, is called Pro*C in Oracle and Sybase database management systems and ECPG in the PostgreSQL database management system. SQL may also be embedded in languages like PHP etc.

Dynamic SQL

Although static SQL works well in many situations, there is a class of applications in which the data access cannot be determined in advance. For example, suppose a spreadsheet allows a user to enter a query, which the spreadsheet then sends to the DBMS to retrieve data. The contents of this query obviously cannot be known to the programmer when the spreadsheet program is written.

To solve this problem, the spreadsheet uses a form of embedded SQL called dynamic SQL. Unlike static SQL statements, which are hard-coded in the program, dynamic SQL statements can be built at run time and placed in a string host variable. They are then sent to the DBMS for processing. Because the DBMS must generate an access plan at run time for dynamic SQL statements, dynamic SQL is generally slower than static SQL. When a program containing dynamic SQL statements is compiled, the dynamic SQL statements are not stripped from the program, as in static SQL. Instead, they are replaced by a function call that passes the statement to the DBMS; static SQL statements in the same program are treated normally.

Difference between Static and Dynamic SQL

Static or Embedded SQL are SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application. **Dynamic** SQL is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries. **Dynamic** SQL is a programming technique that enables you to build SQL statements dynamically at runtime. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation.

Below mentioned are the basic differences
between **Static** *or* **Embedded** and **Dynamic** *or***Interactive** SQL:

STATIC (EMBEDDED) SQL	**DYNAMIC (INTERACTIVE) SQL**
In Static SQL, how database will be accessed is	In Dynamic SQL, how database will be accessed

predetermined in the embedded SQL statement.	is determined at run time.
It is more swift and efficient.	It is less swift and efficient.
SQL statements are compiled at compile time.	SQL statements are compiled at run time.
Parsing, Validation, Optimization and Generation of application plan are done at compile time.	Parsing, Validation, Optimization and Generation of application plan are done at run time.
It is generally used for situations where data is distributed uniformly.	It is generally used for situations where data is distributed non-uniformly.
EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are not used.	EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are used.
It is less flexible.	It is more flexible.

Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events –

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
```

```
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW
```

```
WHEN (NEW.ID > 0)

DECLARE

   sal_diff number;

BEGIN

   sal_diff := :NEW.salary  - :OLD.salary;

   dbms_output.put_line('Old salary: ' || :OLD.salary);

   dbms_output.put_line('New salary: ' || :NEW.salary);

   dbms_output.put_line('Salary difference: ' || sal_diff);

END;

/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Trigger created.
```

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers

SET salary = salary + 500

WHERE id = 2;
```

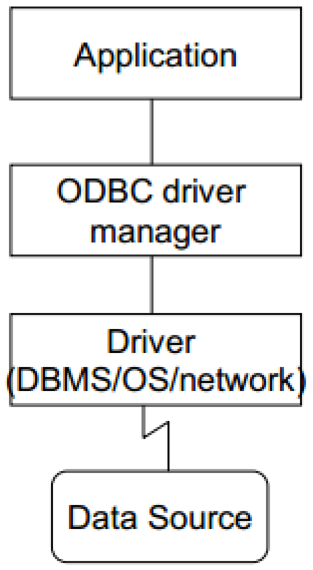
When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

Database Connectivity- ODBC, JDBC

ODBC is (Open Database Connectivity): is a standard or open application programming interface (API) for accessing a database. By using ODBC statements in a program, you can access files in a number of different databases, including Access, dBase, DB2, Excel, and Text. It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases. ODBC handles the SQL request and converts it into a request the individual database system understands.

ODBC Architecture



JDBC

JDBC is: Java Database Connectivity is a Java API for connecting programs written in Java to the data in relational databases. It consists of a set of classes and interfaces written in the Java programming language. It provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API. The standard defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.

JDBC: establishes a connection with a database sends SQL statements processes the results.

JDBC vs. ODBC

ODBC is used between applications. JDBC is used by Java programmers to connect to databases. With a small "bridge" program, you can use the JDBC interface to access ODBC-accessible databases. JDBC allows SQL-based database access for EJB persistence and for direct manipulation from CORBA, DJB or other server objects.

JDBC API

The JDBC API supports both two-tier and three-tier models for database access. Two-tier model -- a Java applet or application interacts directly with the database. Three-tier model -- introduces a middle-level server for execution of business logic: the middle tier to maintain control over data access. The user can employ an easy-to-use higher-level API which is translated by the middle tier into the appropriate low-level calls.

JDBC Architecture

The JDBC Steps:-

1. Importing Packages
2. Registering the JDBC Drivers
3. Opening a Connection to a Database
4. Creating a Statement Object

5. Executing a Query and Returning a Result Set Object
6. Processing the Result Set
7. Closing the Result Set and Statement Objects
8. Closing the Connection

1: Importing Packages

```
import java.sql.*;

import java.math.*;

 import java.io.*;

import oracle.jdbc.driver.*;
```

2: Registering JDBC Drivers

```
class LecExample_1a

{

public static void main (String args [])

   throws SQLException {
```

// Load Oracle driver

```
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
```

3: Opening connection to a Database

//Prompt user for username and password

```
String user;

String password;

user = readEntry("username: ");

password = readEntry("password: ");
```

// Connect to the local database

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:thin:@aardvark:1526:teach

", user, password);
```

4: Creating a Statement Object

```
 // Query the hotels table for resort = 'palma nova’

 //   Please notice the essential trim

 PreparedStatement pstmt = conn.prepareStatement ("SELECT hotelname, rating
    FROM hotels WHERE trim(resort) = ?");
```

```
pstmt.setString(1, "palma nova");
```

5: Executing a Query, Returning a Result Set Object &

6. Processing the Result Set

```
ResultSet rset = pstmt.executeQuery ();
```

// **Print query results while (rset.next ())**

```
System.out.println (rset.getString (1)+" "+ rset.getString(2));
```

7. Closing the Result Set and Statement Objects

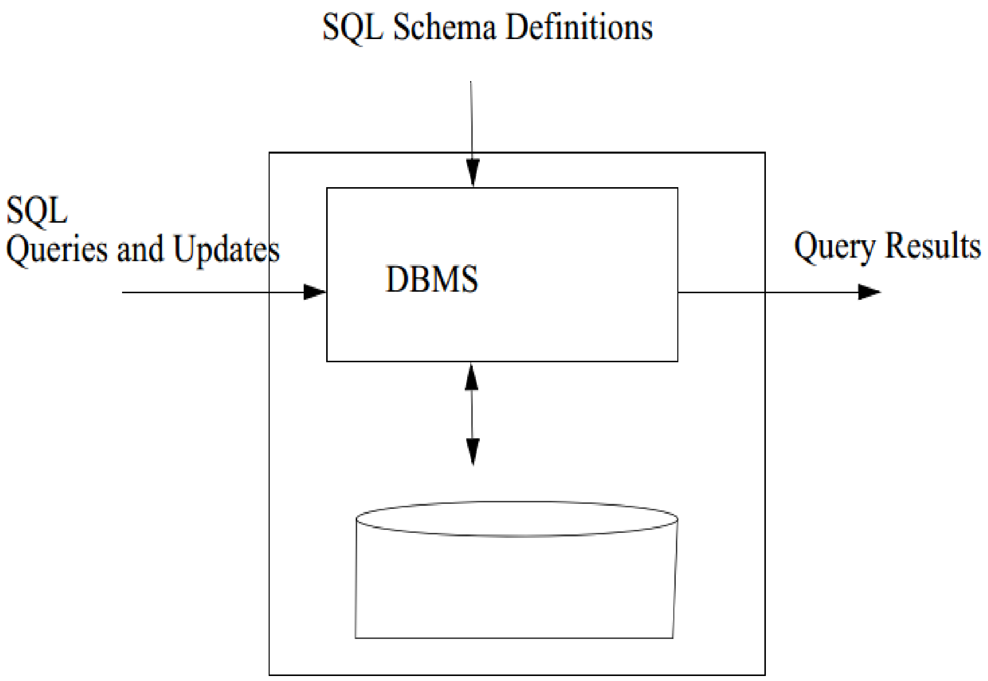
8. Closing the Connection

```
//close the result set, statement, and the connection

    rset.close ();

    pstmt.close ();

    conn.close ();

  }
```

Active Databases

Conventional Database Systems

-General principles of Conventional Database Systems



Conventional (Passive) Databases

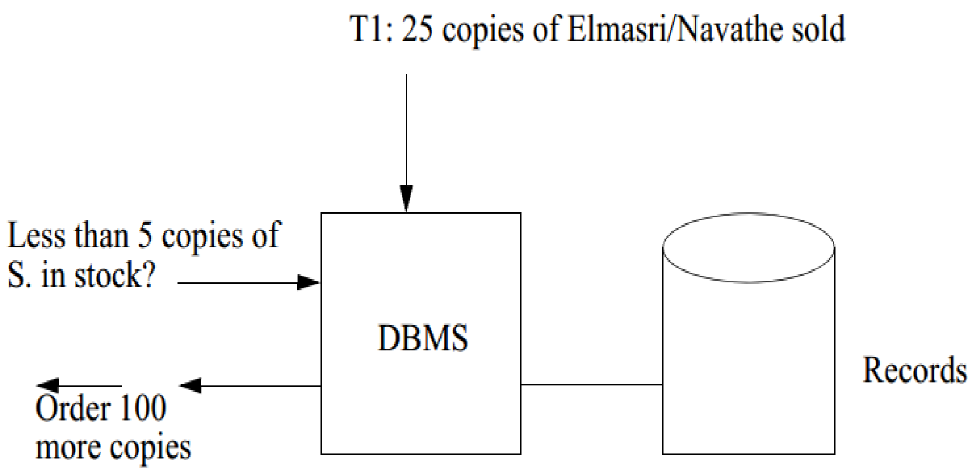
• Data model, usually relational
• Transaction model - Passive update principle

- Client controls DBMS updates

Example of real world problem not well suited for passive update principle:

• Inventory control - reordering items when quantity in stock falls below threshold.
• Travel waiting list - book ticket as soon as right kind is available
• Stock market - Buy/sell stocks when price below/above threshold

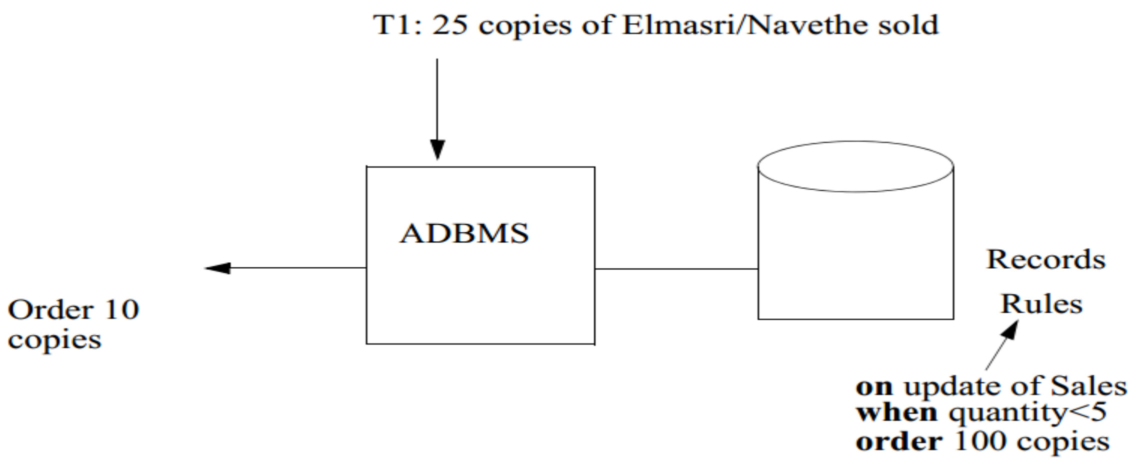
Conventional Databases- Passive DBMS


T1: 25 copies of Elmasri/Navathe sold
Less than 5 copies of S. in stock?
DBMS
Order 100 more copies
Records

Periodical polling of database by application
- Frequent polling => expensive
- Infrequent polling => might miss the right time to react
• The polling has to be done for all items in stock and can be expensive.
• Problem is that DBMS does not know that application is polling.

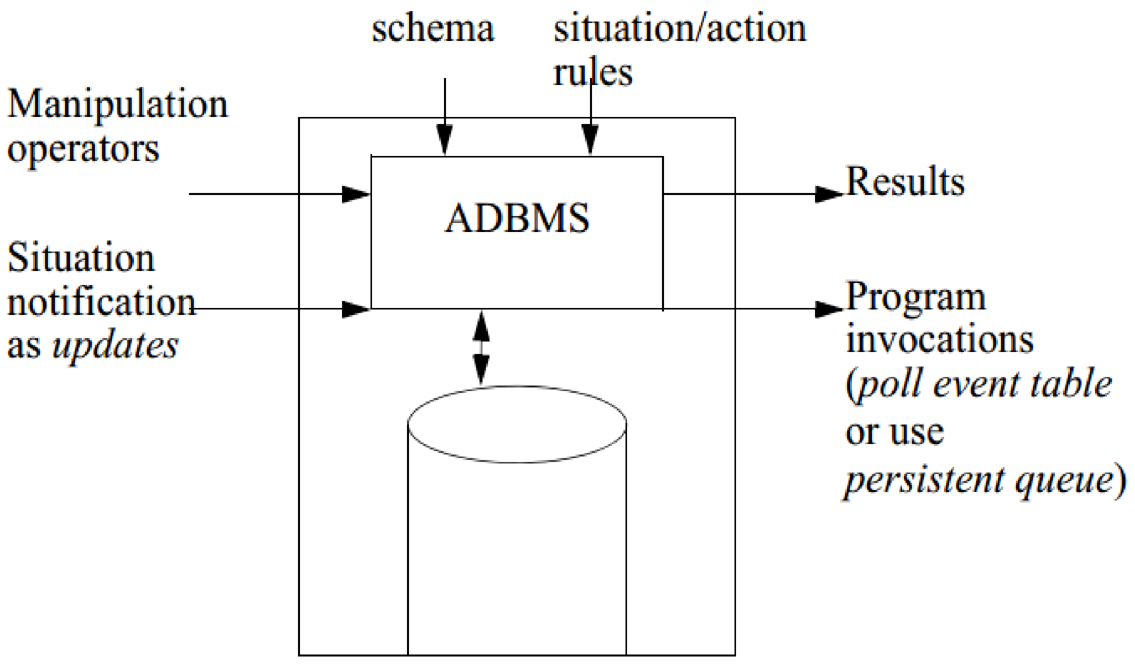
Active Databases- Active DBMS

Recognize predefined situations in database
• Trigger predefined actions when situations occur
Actions are usually database updates, not calls to external programs to e.g. order items.



Active Databases (General idea)

• ADBMS provides: Regular DBMS primitives + definition of application-defined situations + triggering of application-defined reactions


schema
situation/action rules
Manipulation operators
ADBMS
Results
Situation notification as *updates*
Program invocations (*poll event table* or use *persistent queue*)