UNIVERSITY OF KARACHI

DEPARTMENT OF COMPUTER SCIENCE

PRACTICAL / LAB

HANDOUT

DATA BASE SYSTEMS

PREPARED & COMPILED

BY

NAZISH ALI / S.M KHALID JAMAL

University Of Karachi

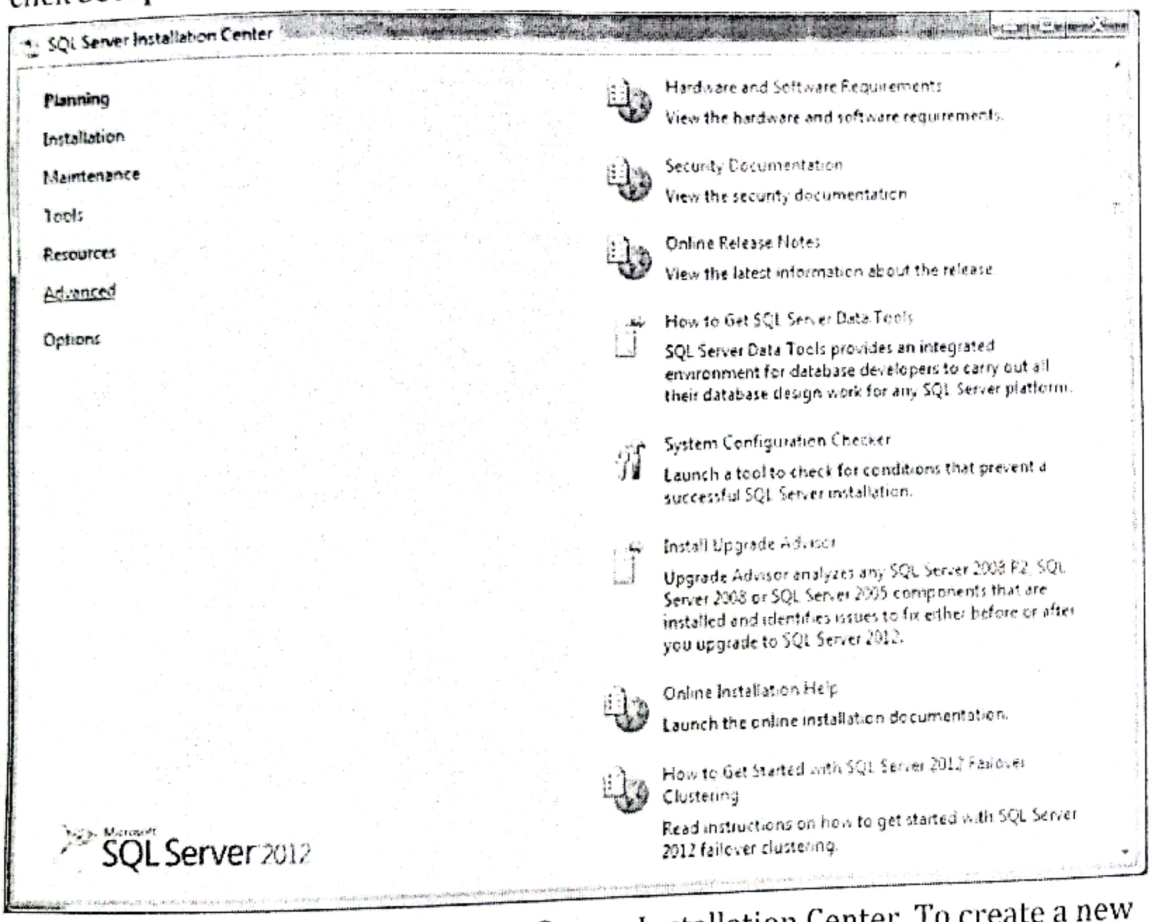
Department Of Computer Science

HAND OUT #0

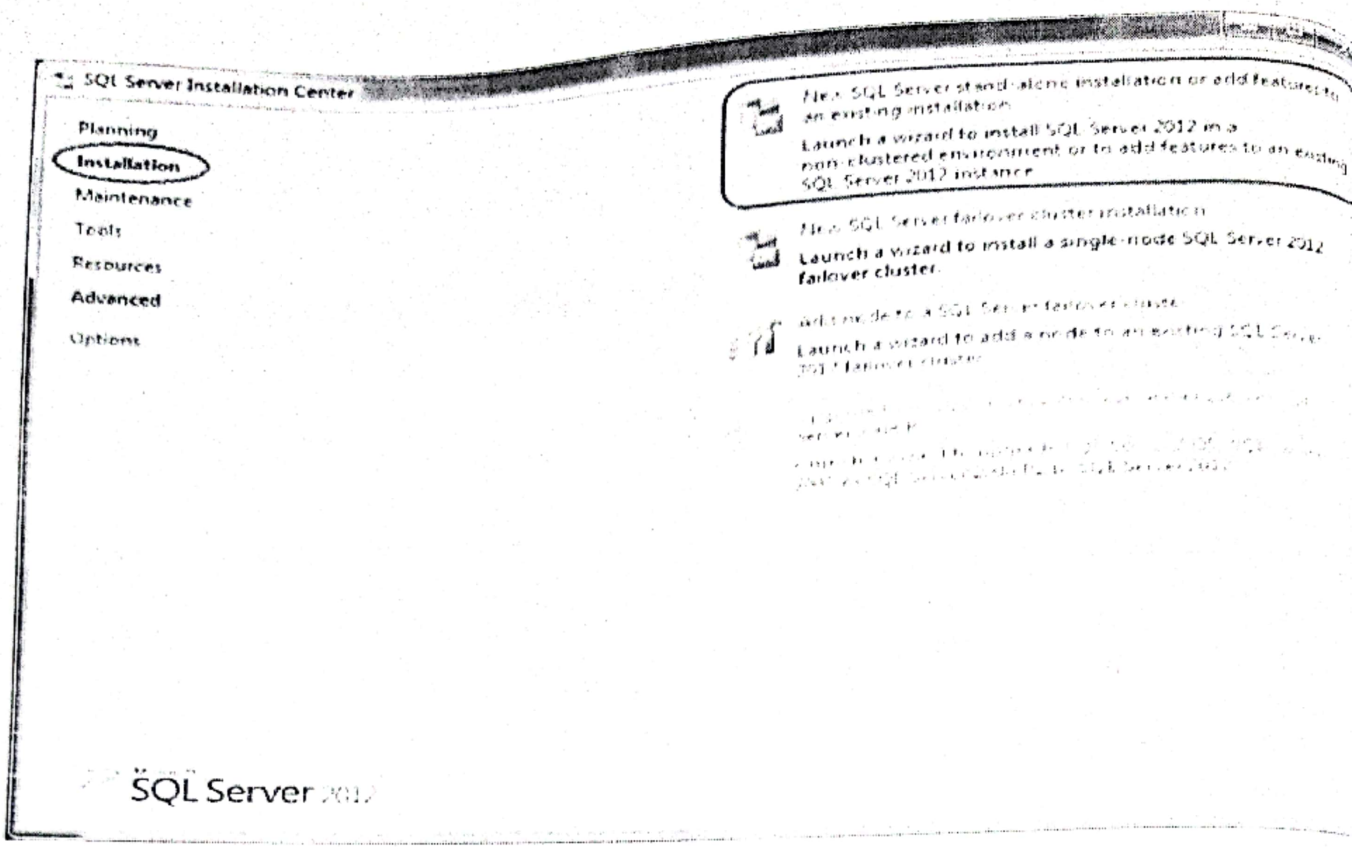
INSTALLATION GUIDE

Installing SQL Server 2012

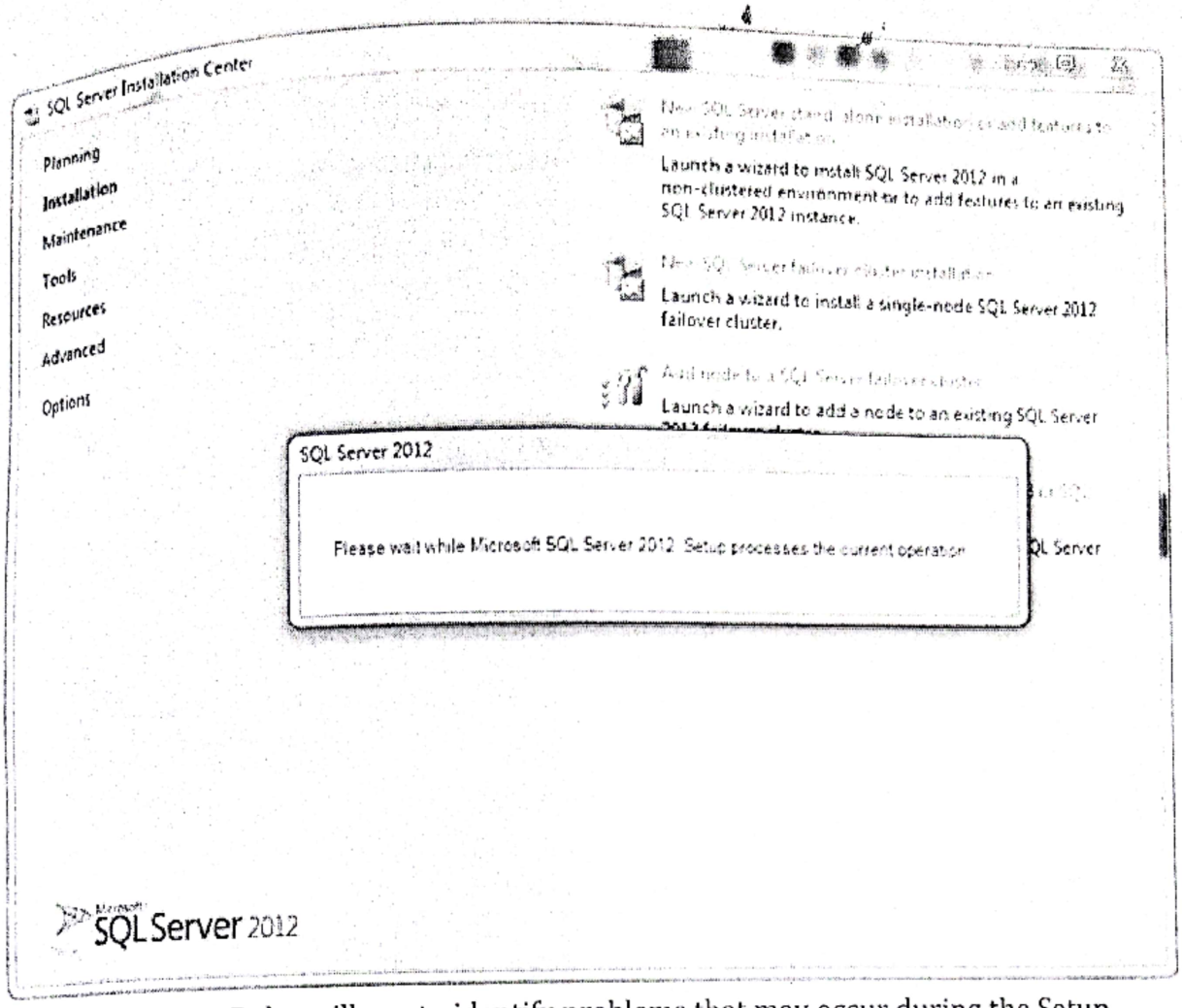
Insert the SQL Server installation media. From the root folder, double-click **Setup.exe**. To install from a network share, locate the root folder on the share, and then double-click **Setup.exe**:



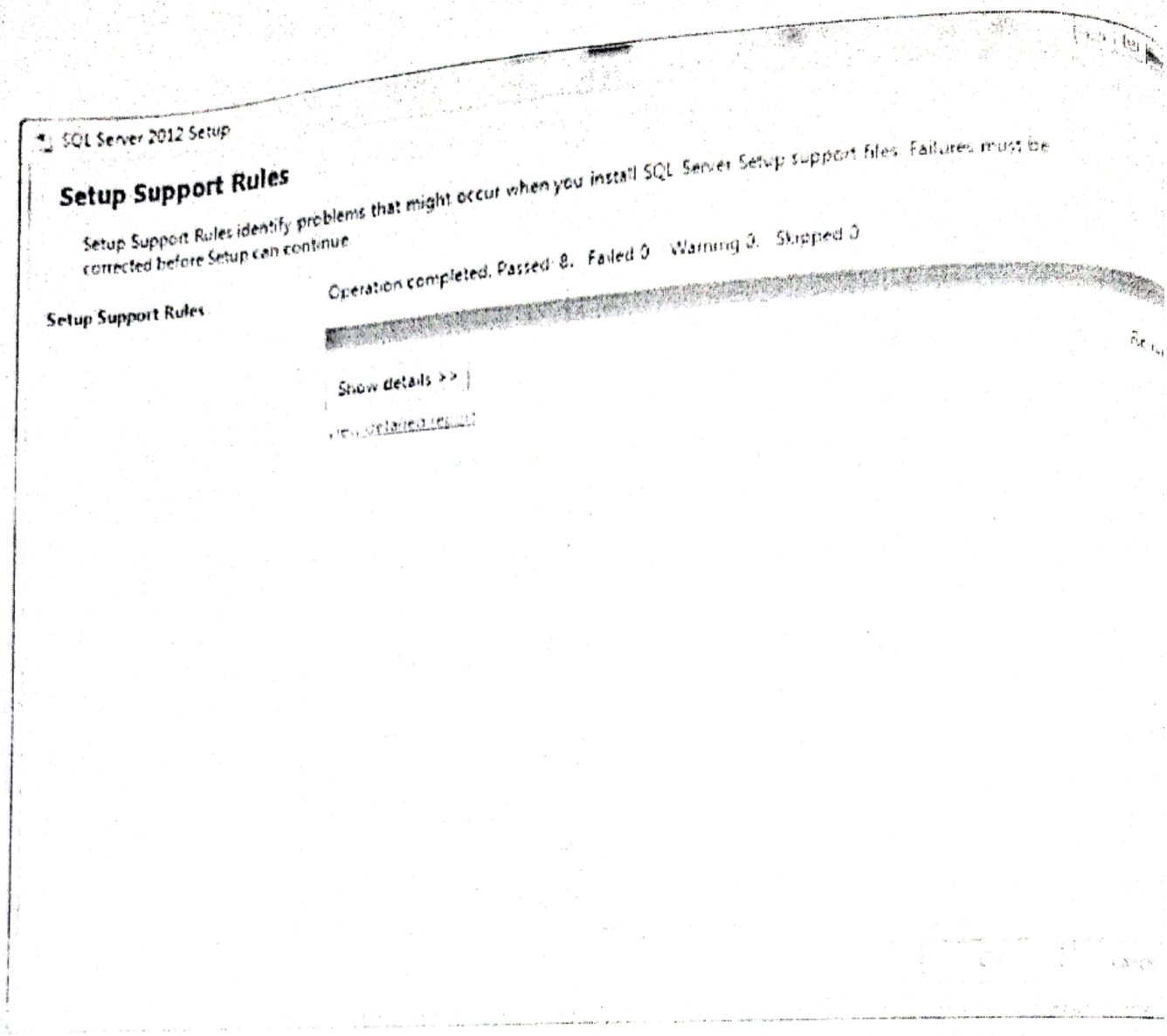
The Installation Wizard runs the SQL Server Installation Center. To create a new installation of SQL Server, select the Installation option on the left side, and then click **New SQL Server stand-alone installation or add features to an existing installation**:



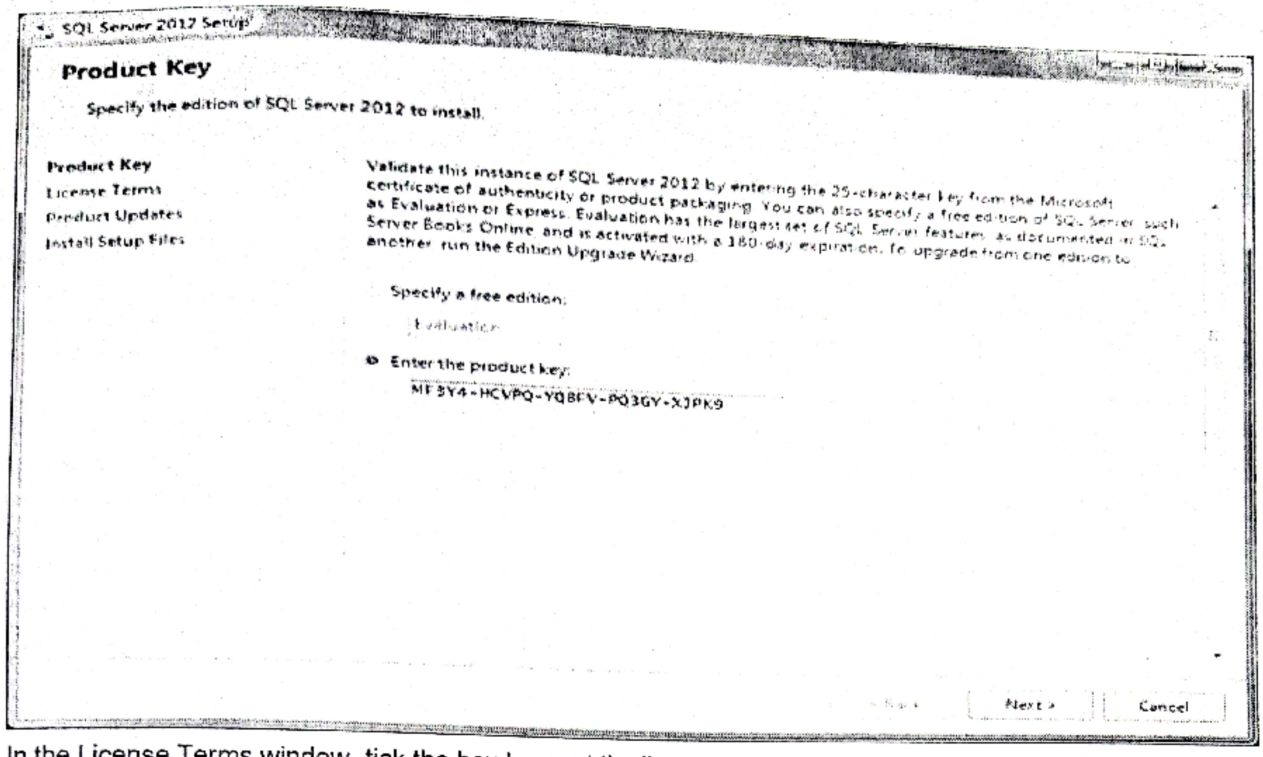
Setup is now preparing to launch Setup Support Rules window:



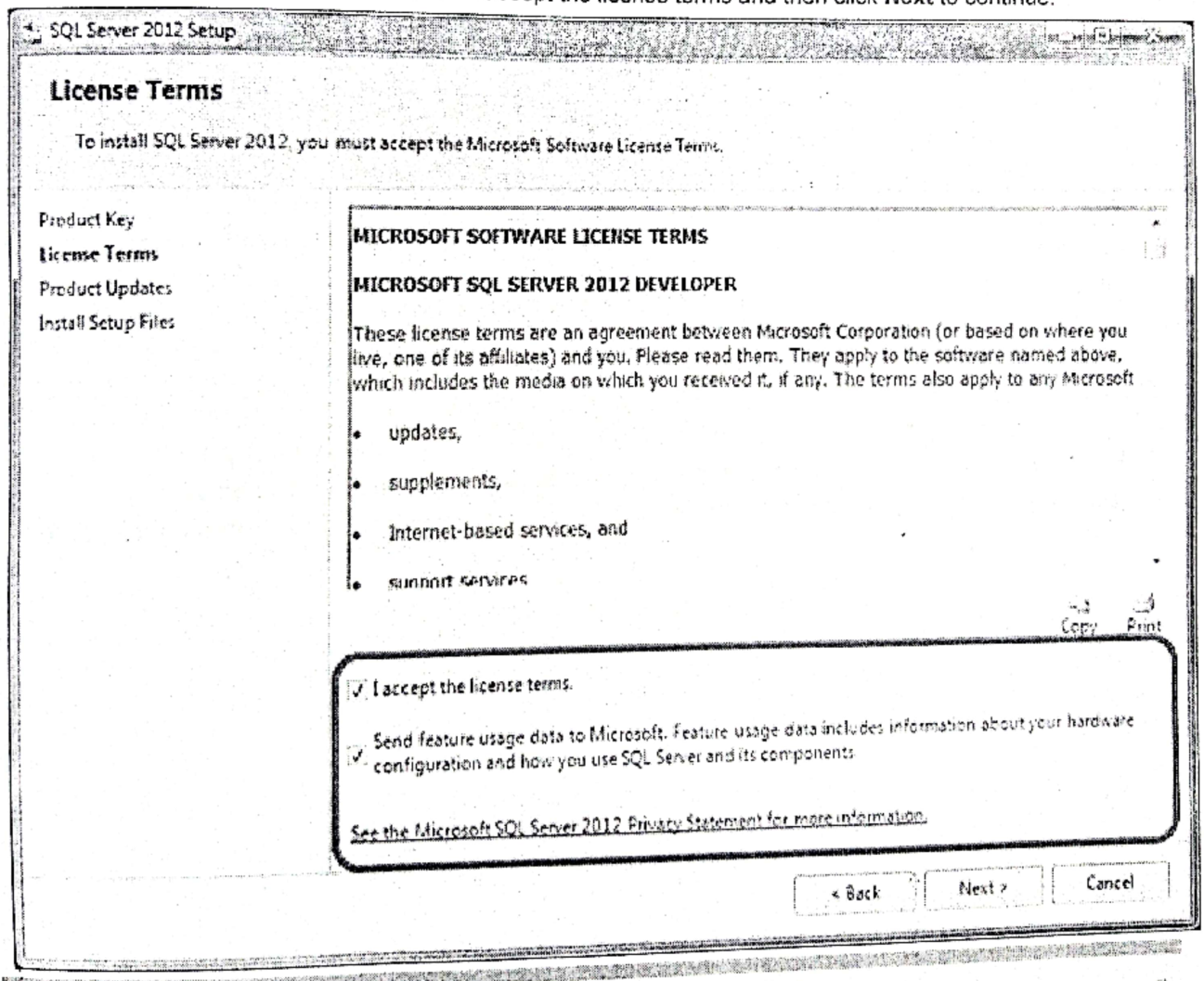
Now Setup Support Rules will run to identify problems that may occur during the Setup Support Files installation:



Once this step finishes click **OK** to proceed to Product Key window, In the Product Key window, enter the Product license key (if required), and click **Next** to continue:

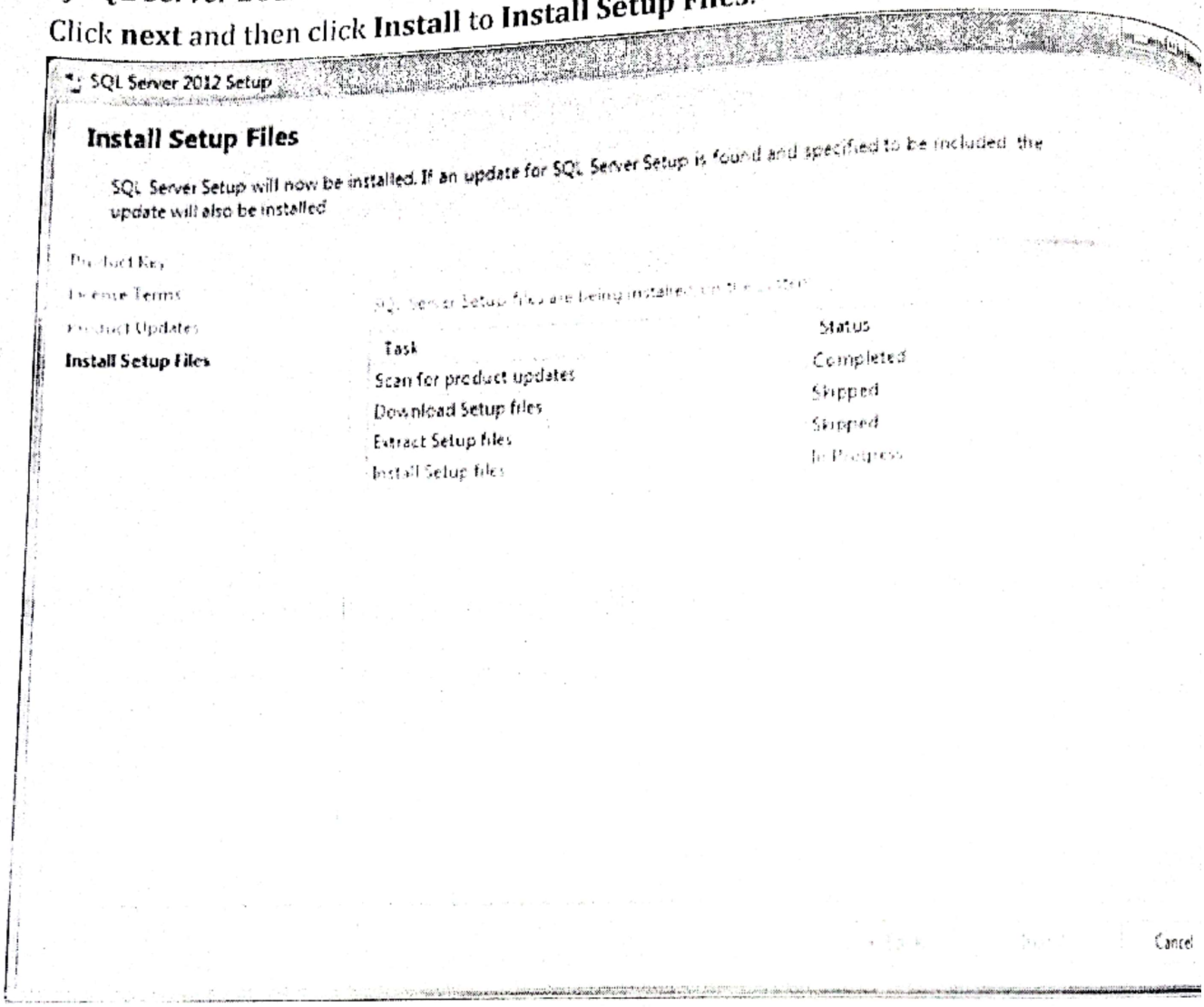


In the License Terms window, tick the box I accept the license terms and then click **Next** to continue:

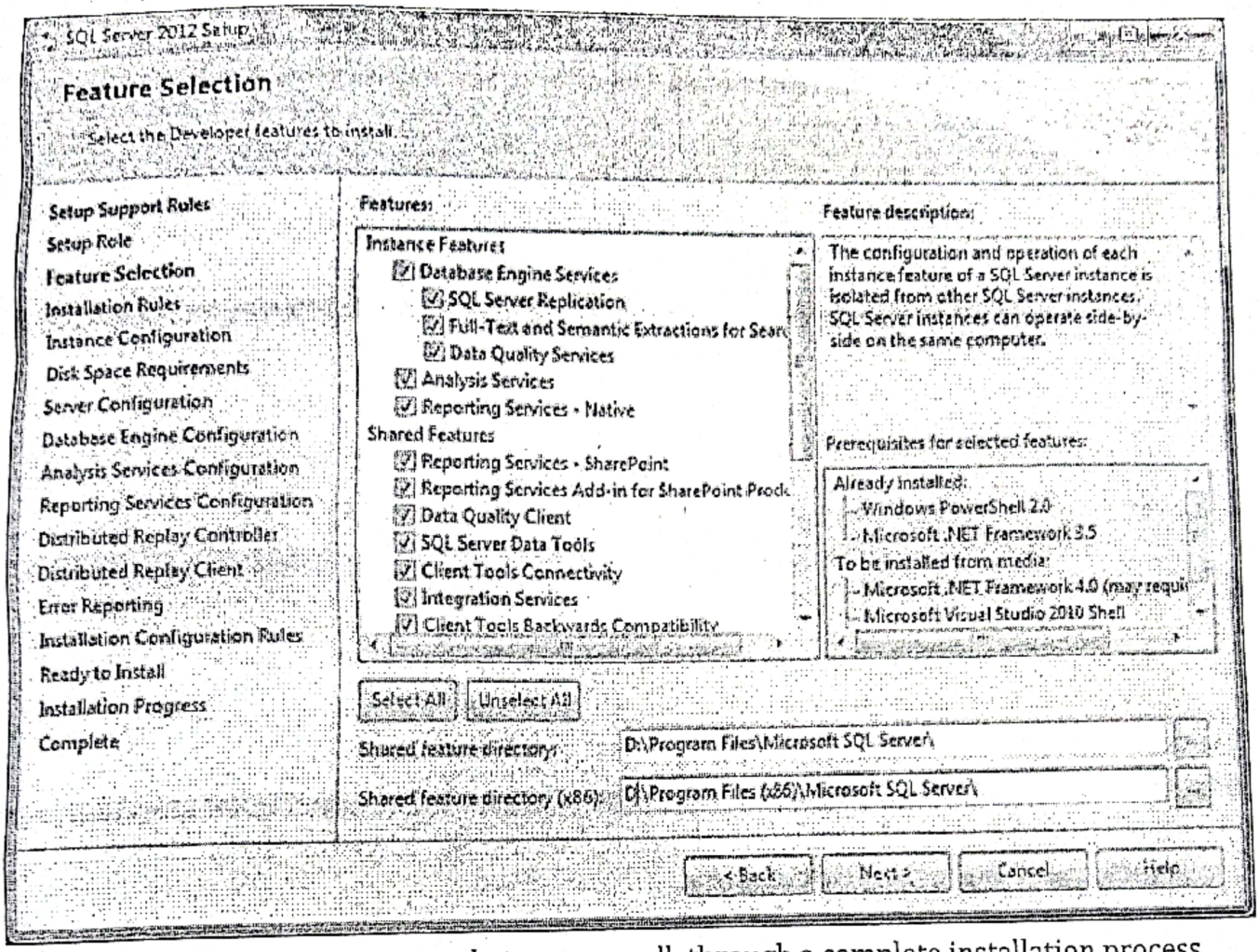


Note: You must accept the license agreement before you can continue the installation of SQL Server 2012. Send feature usage data to Microsoft option is optional.

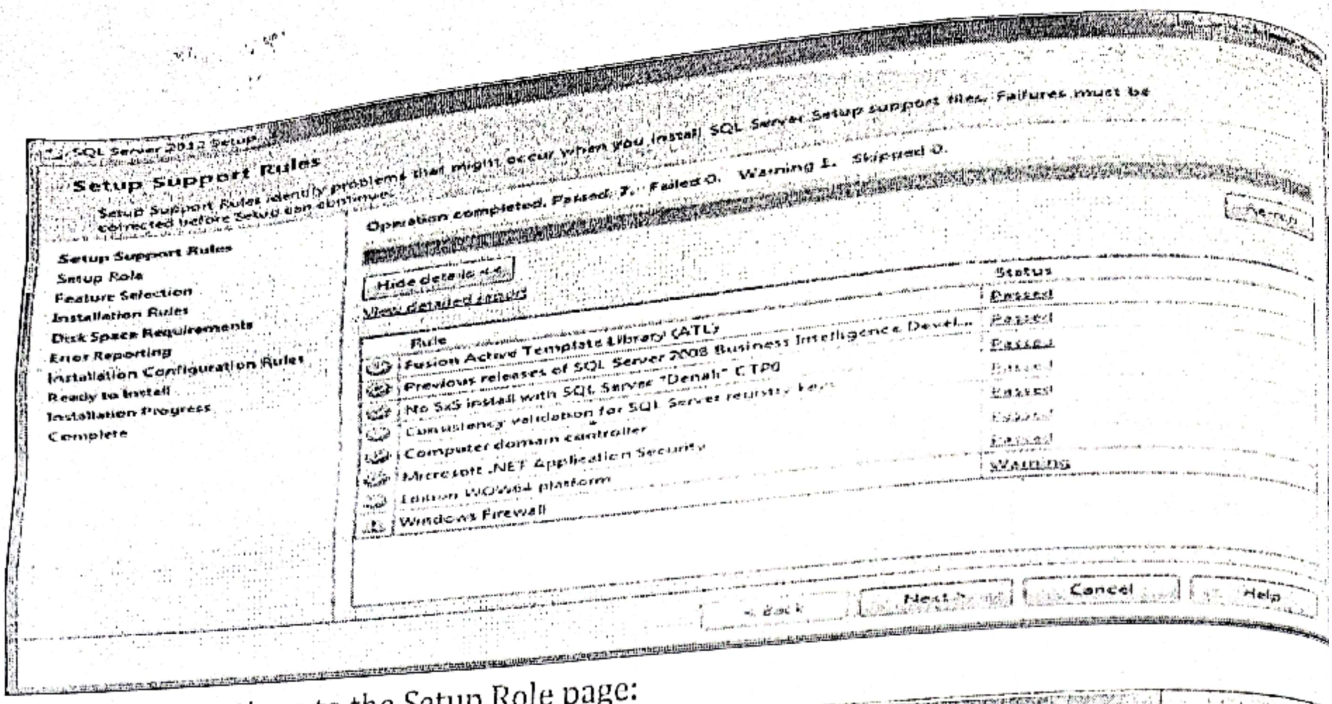
Click **next** and then click **Install** to **Install Setup Files:**



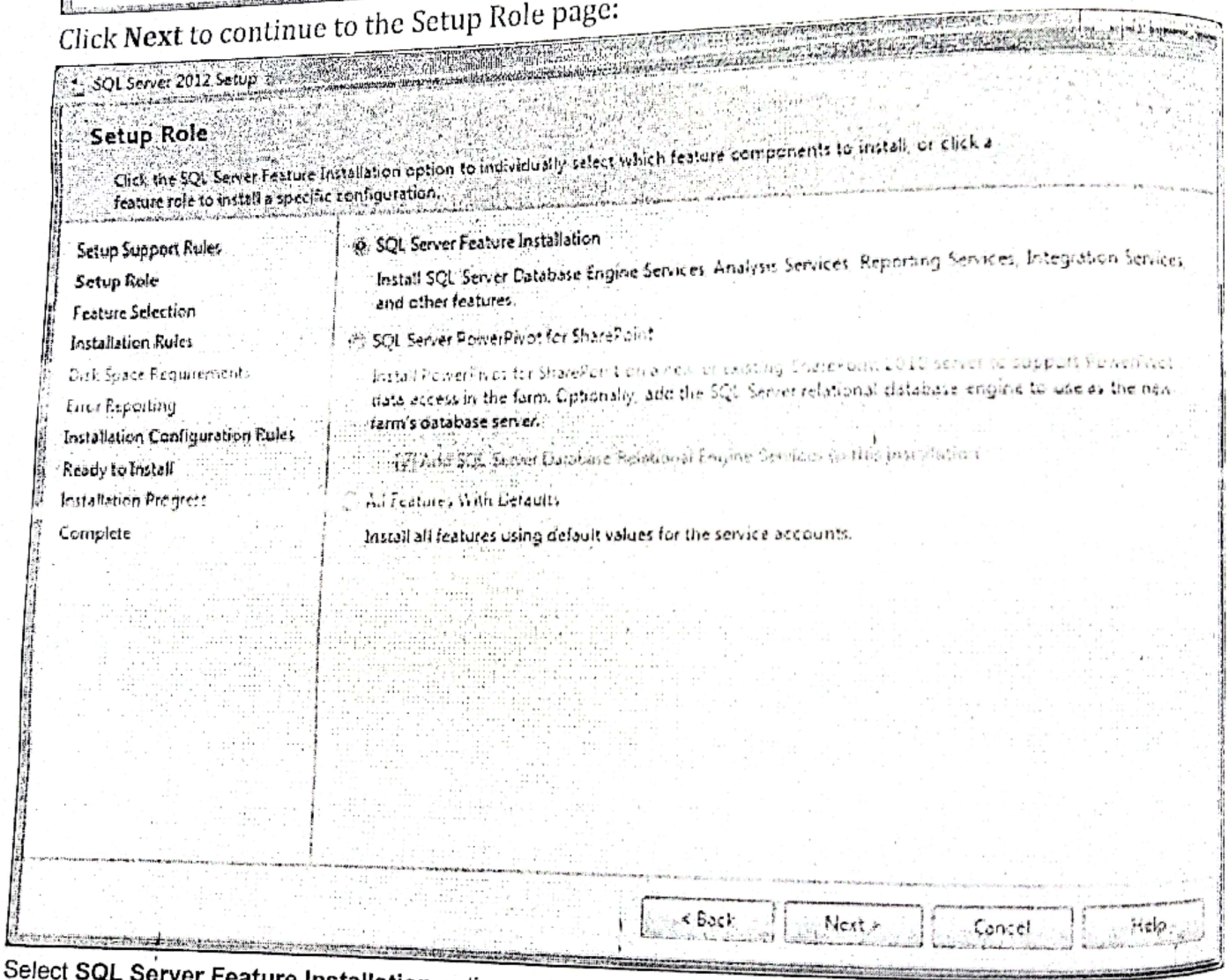
These files are necessary to perform the actual installation. Following the installation of th setup support files, you will be presented with another compatibility check. Following dialog appears once you successfully pass these checks. You can click the **Show Details** button under the green progress bar if you want to see the individual checks listed



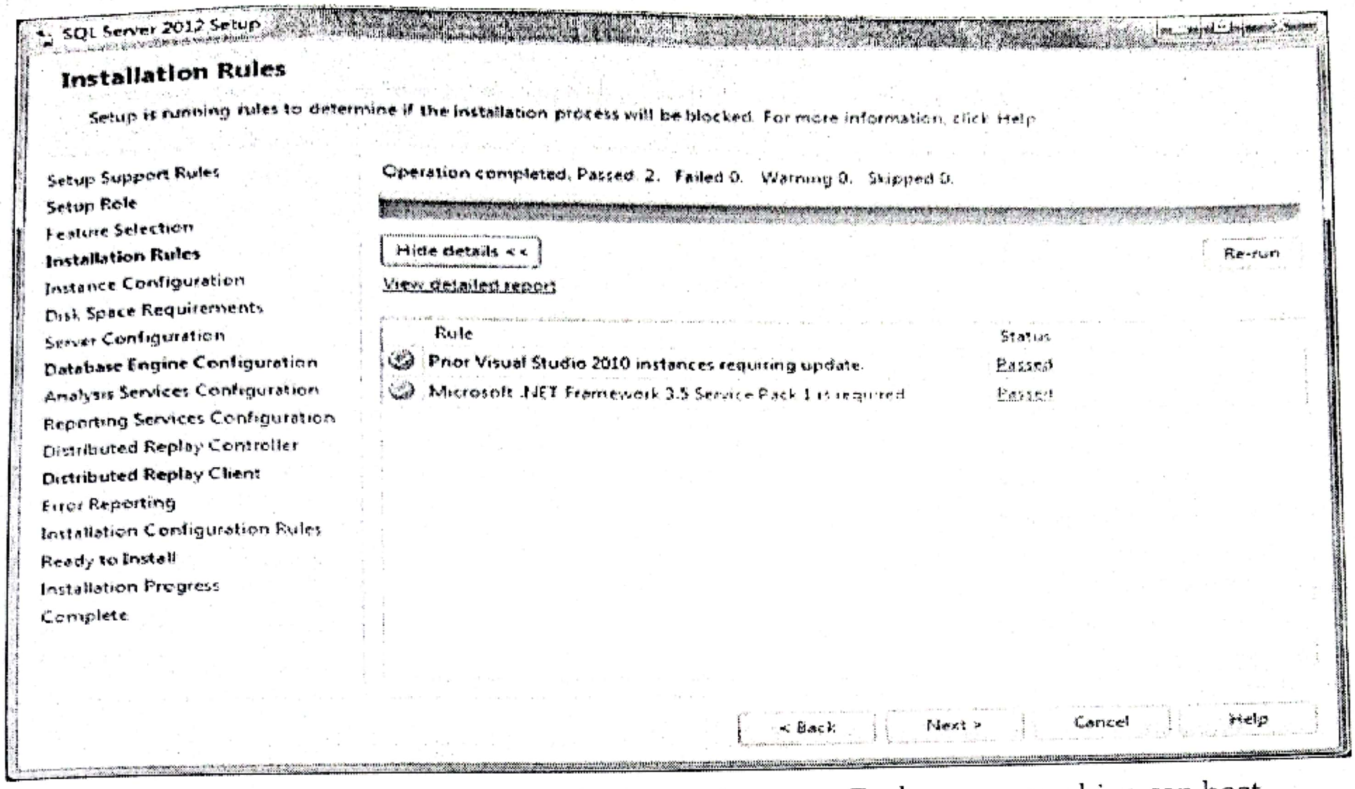
I'm going to select all features so that we can walk through a complete installation process. This will install the Database Engine Services, Analysis Services, Reporting Services, and a number of shared features including SQL Server Books Online. You can also specify the shared feature directory where share features components will be installed. Click **Next** to continue to the Installation Rules page, Setup verifies the system state of your computer before Setup continues:



Click **Next** to continue to the Setup Role page:

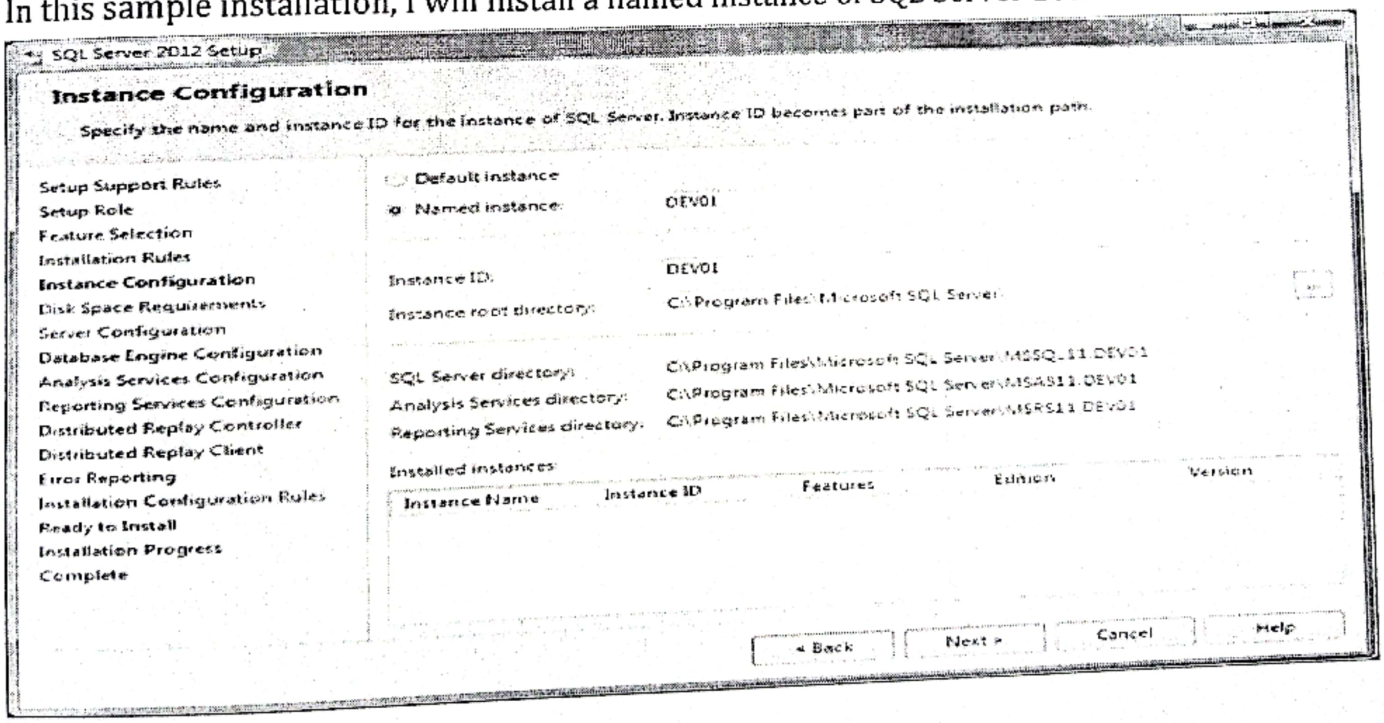


Select **SQL Server Feature Installation** option and then **Next** to continue to Feature Selection page:



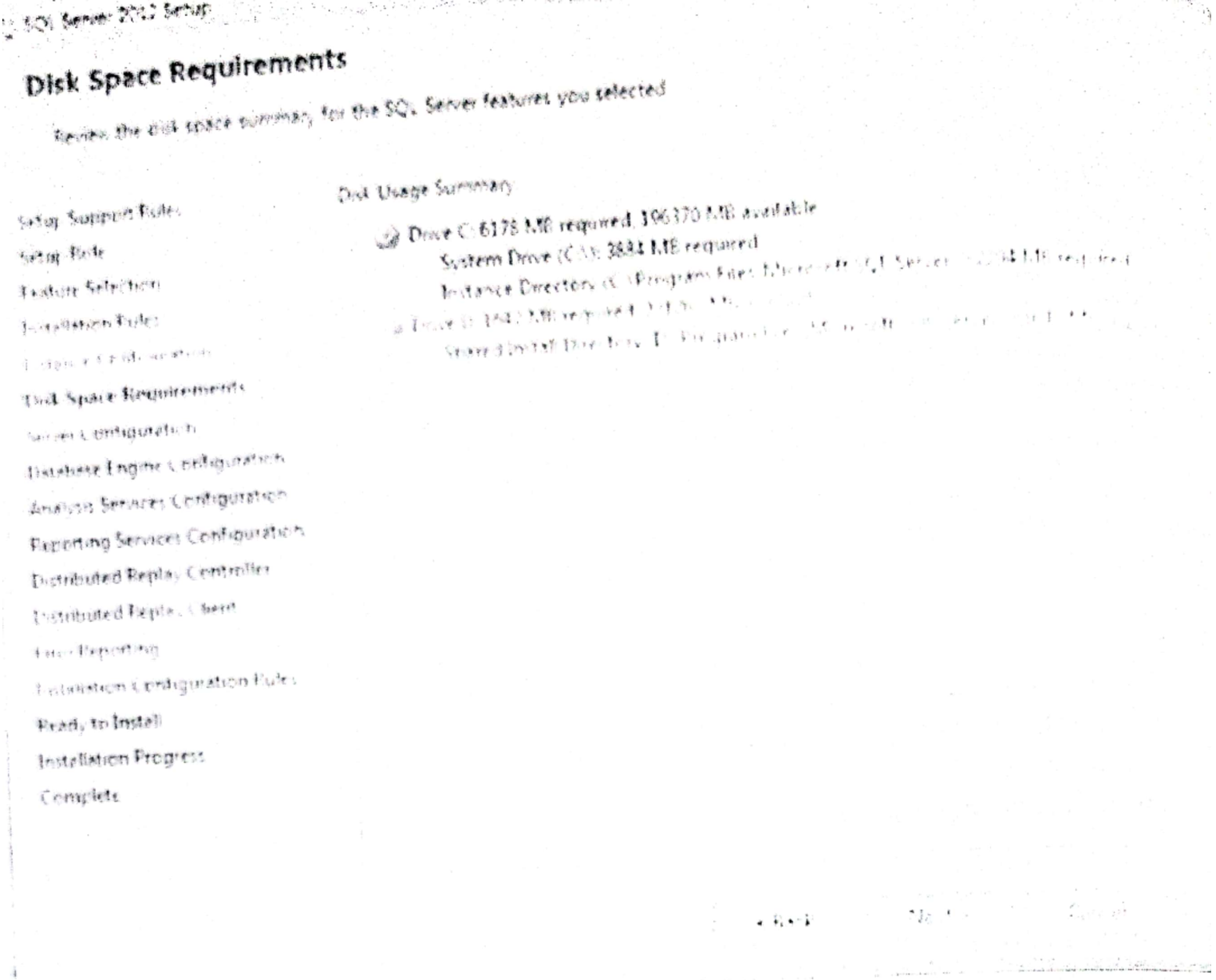
Click **Next** to continue to the Instance Configuration page. Each server machine can host one default instance of SQL Server, which resolves to the server name, and multiple named instances, which resolves to the pattern ServerName\InstanceName.

In this sample installation, I will install a named instance of SQL Server 2012 called DEV01:

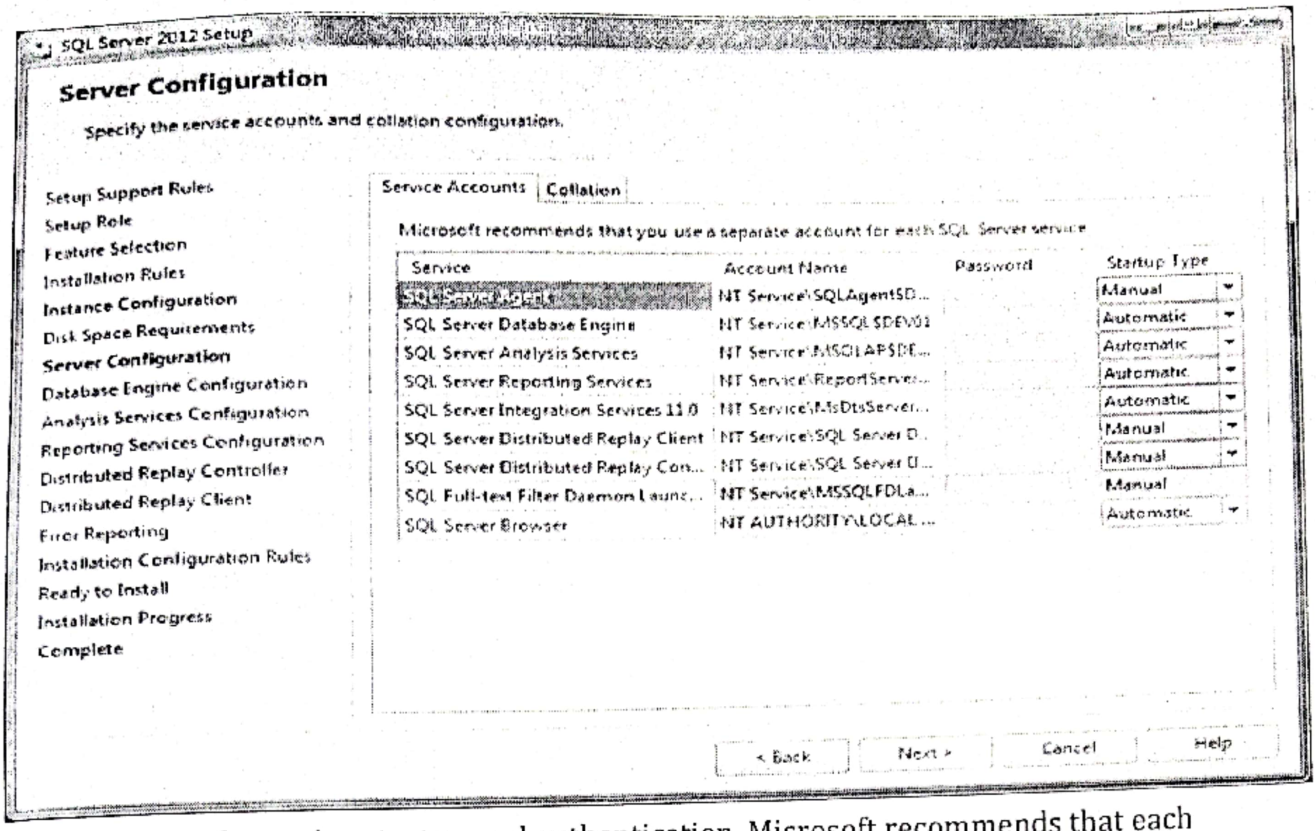


Note: This screen also report on other instances installed on this machine.

Click **Next** to proceed to the Disk Space Requirements page:



This is just a information page that does not require you to make any choices. Click N go to Server Configuration page:



Here you specify service startup and authentication. Microsoft recommends that each service account have separate user accounts as a security best practice as shown in the following figure:

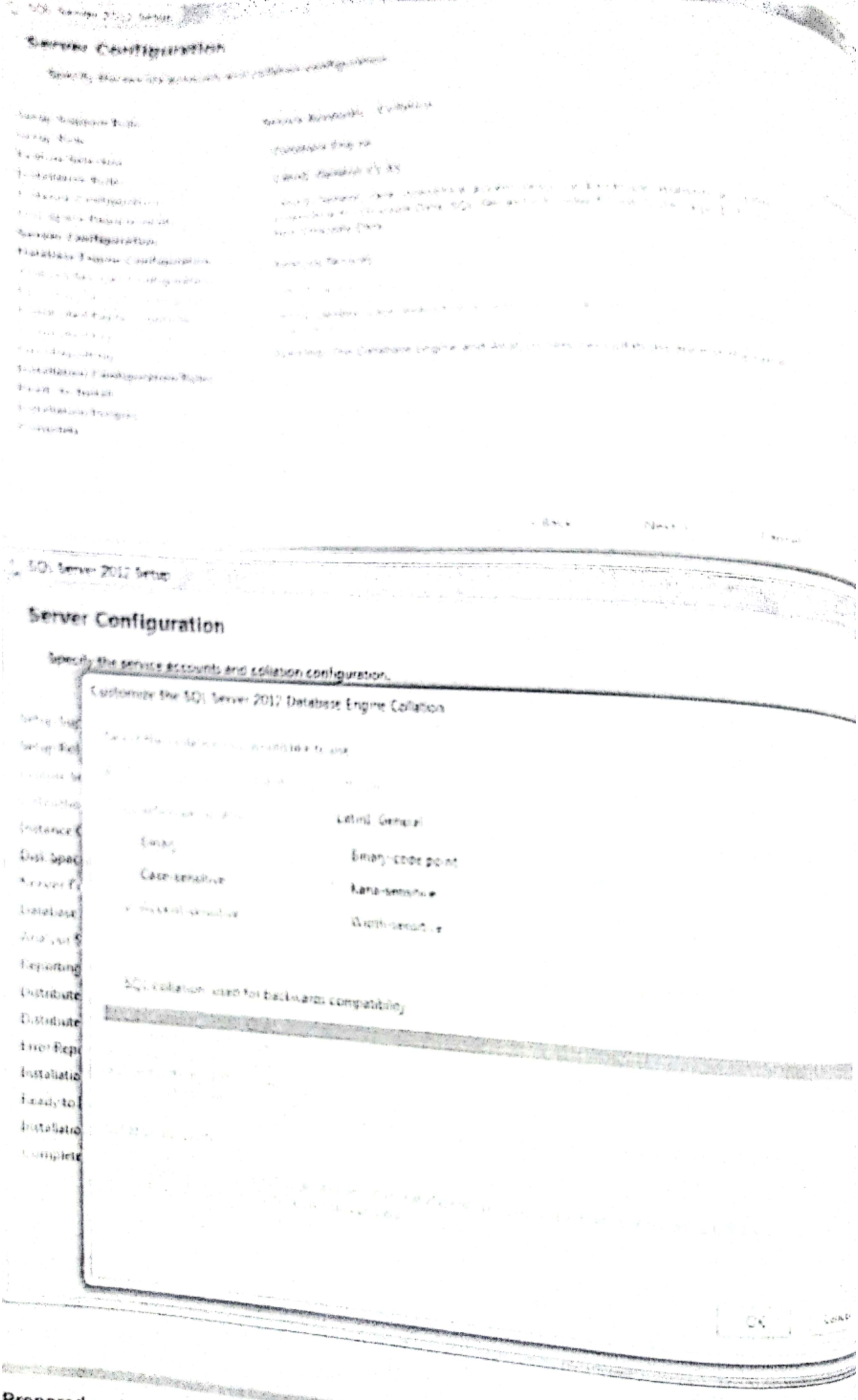
svc_SQL	User	SQL Server service account
svc_SQLBrowser	User	SQL Browser service account
svc_SQLReportServer	User	SQL Report Server service account
svc_SQLServerAgent	User	SQL Server Agent service account
svc_SQLServerOLAP	User	SQL Server OLAP service account

The SQL Server 2012 Books Online makes the following security recommendations:

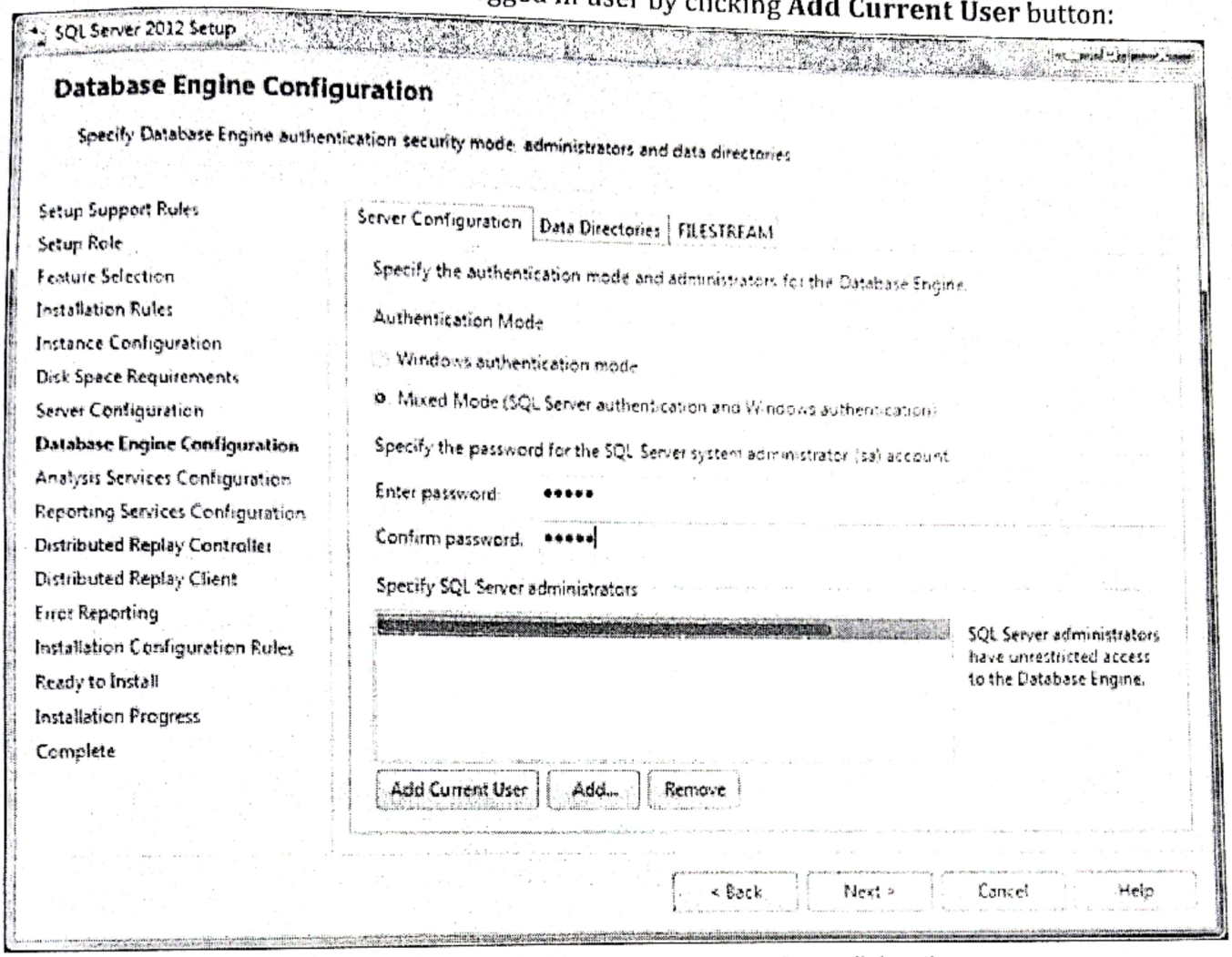
- Run separate SQL Server services under separate Windows accounts.
- Run SQL Server services with the lowest possible privileges.
- Associate SQL Server services with Windows accounts.
- Require Windows Authentication for connections to the SQL Server.

For the purposes of this article, we will authenticate all services using LocalSystem account.

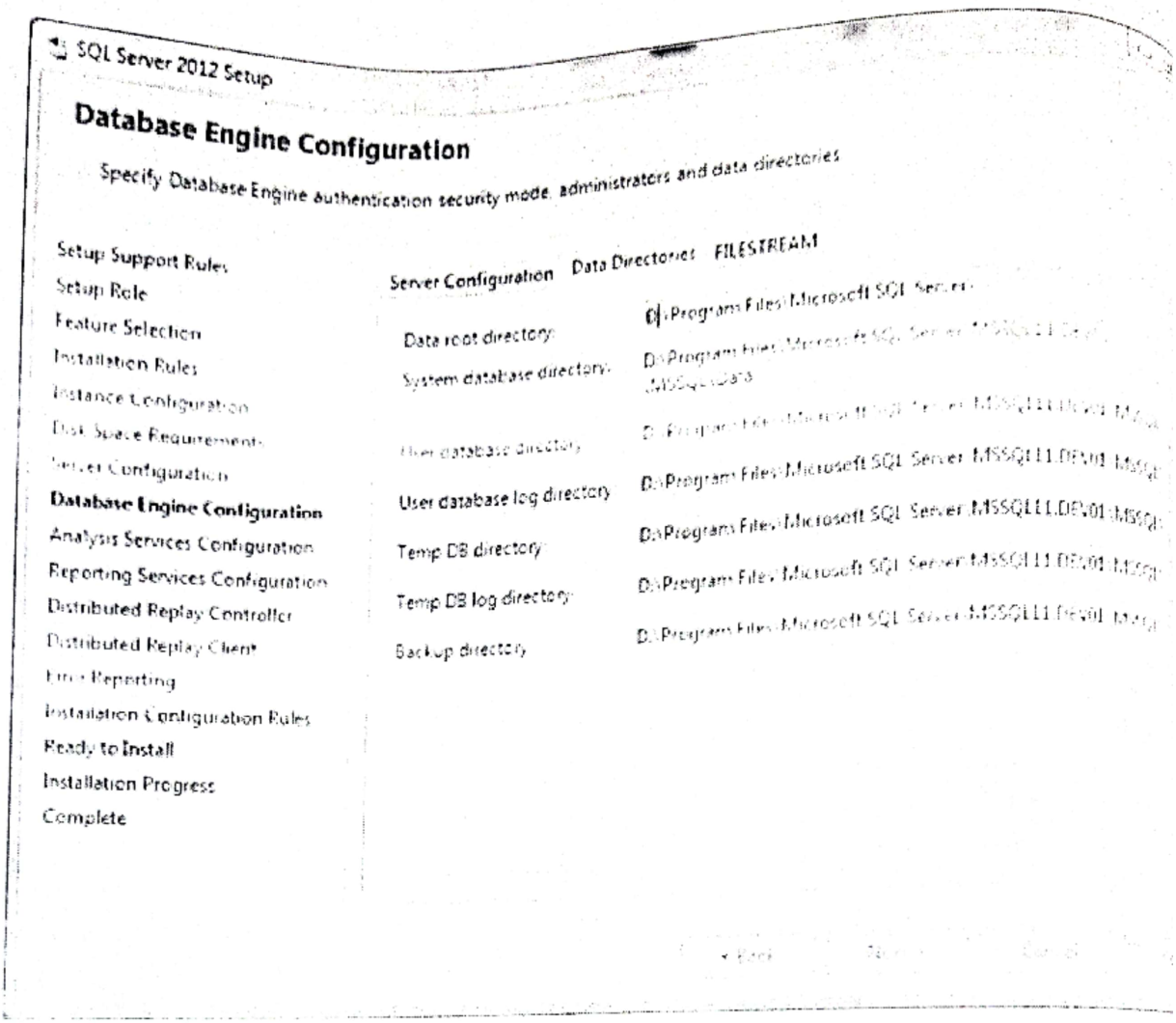
You may have noticed the Collation tab here in this page. Click **Collation** tab and then click Customize button to specify the collation for your Database Engine and Analysis Services instance that best matches your application need:


Server Configuration
Server Configuration
Specify the service accounts and collation configuration.
Customize the SQL Server 2012 Database Engine Collation
Latin1_General
Binary-code point
Case-sensitive
Kana-sensitive
Width-sensitive

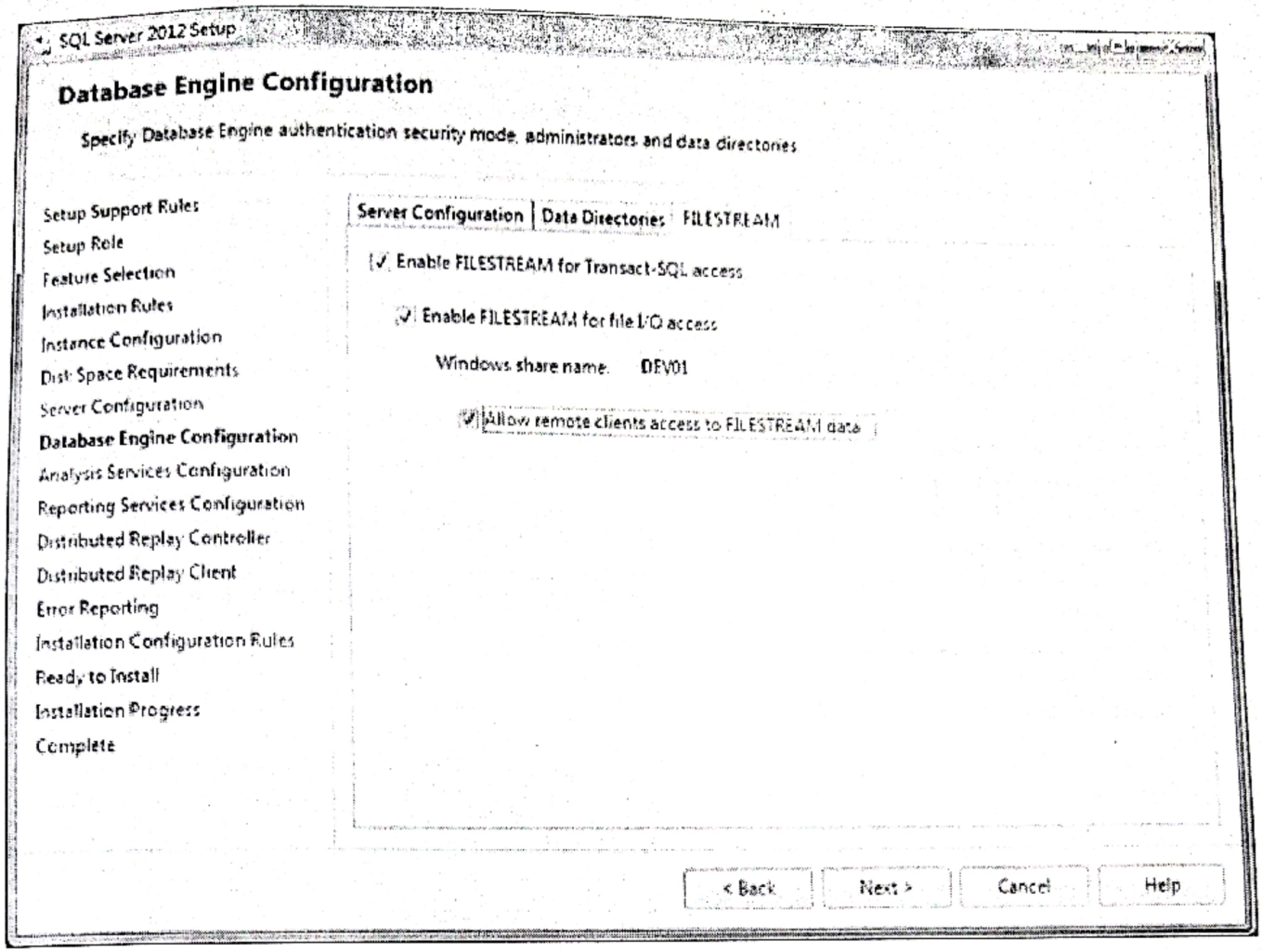
Click **Next** to continue to Database Engine Configuration page. The SQL Server 2012 authentication mode was configured for "Windows Authentication Mode" per Microsoft security best practice. For the purposes of this article, I will be choosing "Mixed Mode Authentication". You will also need to specify the SQL Server administrators to be used; in this example I will use the current logged in user by clicking **Add Current User** button:



Click Data Dictionary tab and specify default database, log, backup, and tempdb locations:



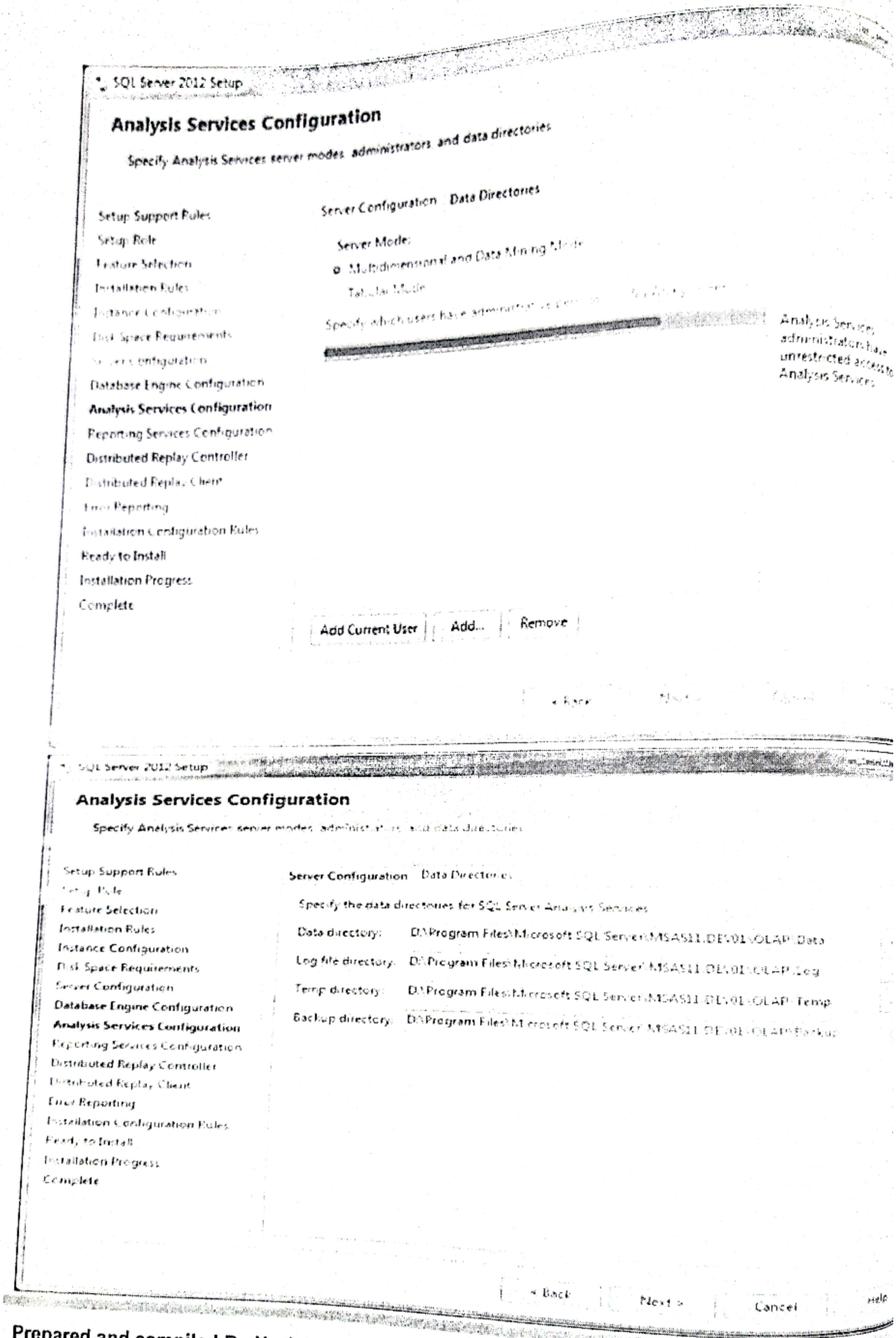
Click FILESTREAM tab and enable this feature as below:



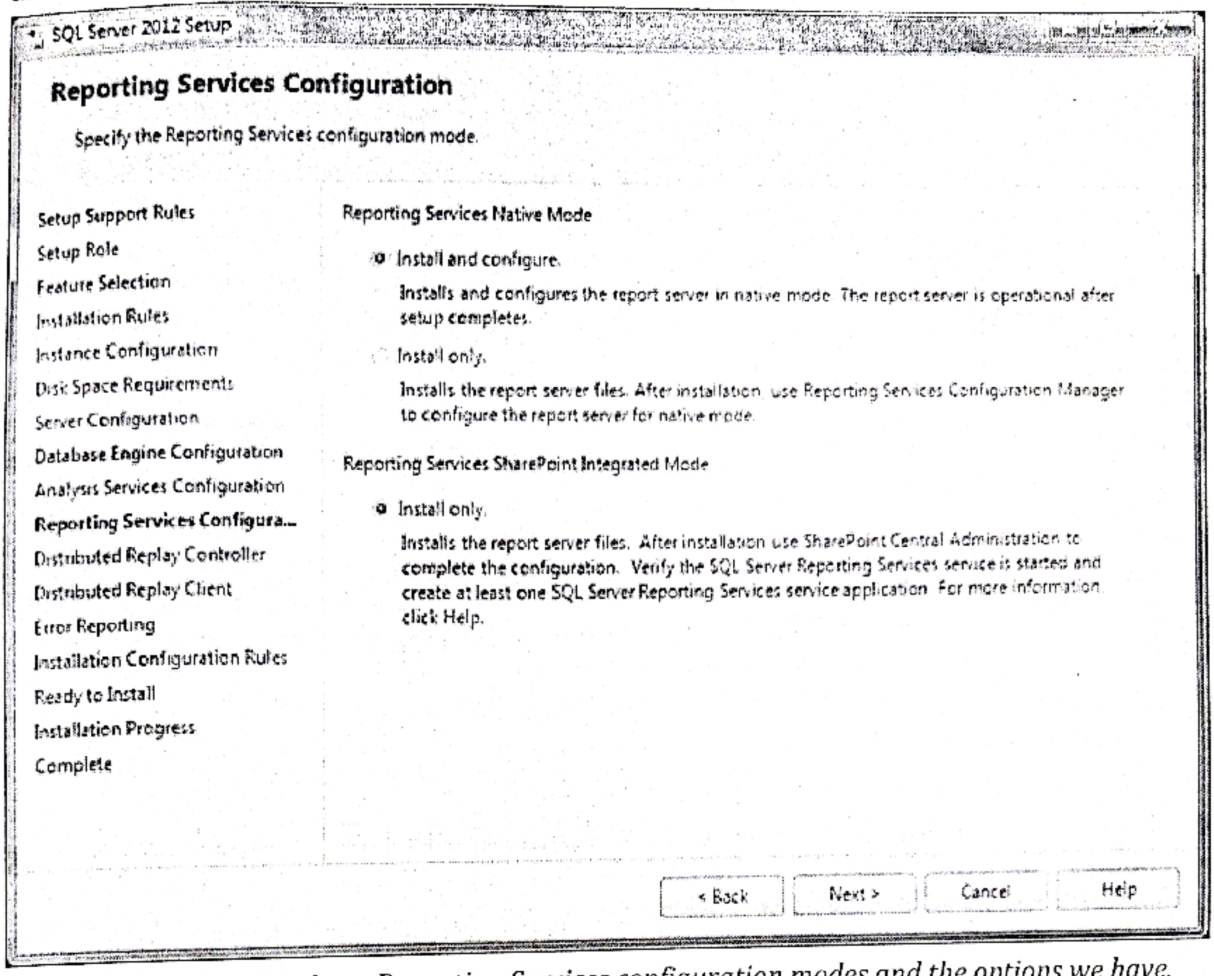
FILESTREAM feature is quiet useful when binaries or other data that does not fit neatly in a table structure.

Click **Next** to proceed to Analysis Services Configuration page, this is similar to Database Engine Configuration page.

Specify users or accounts that will have administrator permissions for Analysis Services in Account Provisioning page and specify non-default installation directories in Data Directories page:

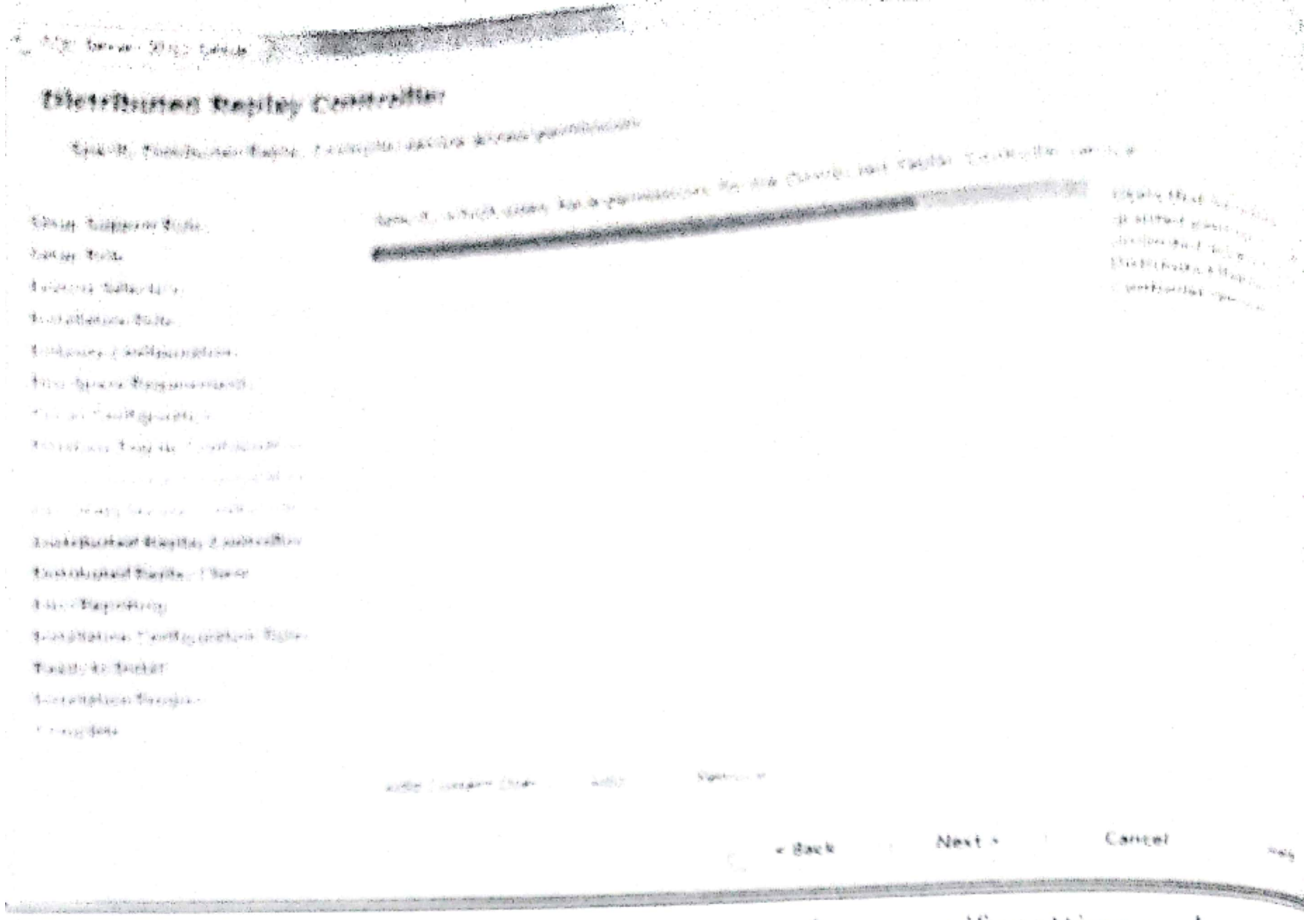

SQL Server 2012 Setup
Analysis Services Configuration
Specify Analysis Services server modes, administrators, and data directories
Setup Support Rules
Setup Role
Feature Selection
Installation Rules
Instance Configuration
Disk Space Requirements
Server Configuration
Database Engine Configuration
Analysis Services Configuration
Reporting Services Configuration
Distributed Replay Controller
Distributed Replay Client
Error Reporting
Installation Configuration Rules
Ready to Install
Installation Progress
Complete
Server Configuration
Data Directories
Server Mode:
Multidimensional and Data Mining Mode
Tabular Mode
Add Current User
Add...
Remove
< Back
Next >
Cancel
Help
Data directory:
Log file directory:
Temp directory:
Backup directory:

Click Next to continue to Reporting Services Configuration page, specify the kind of Reporting Services installation to create. For the purpose of this article, I will use Install and Configure option:

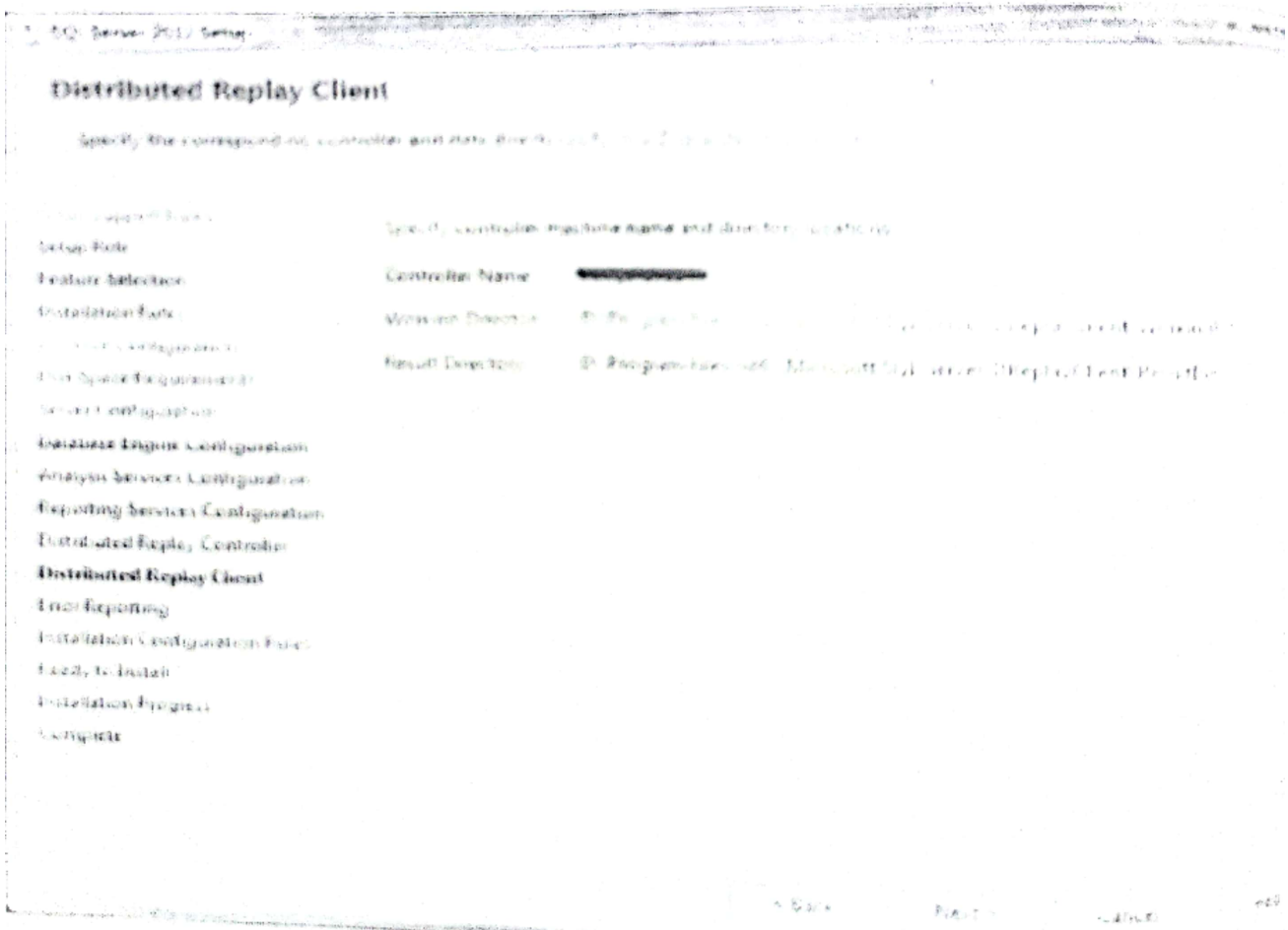


For more information about Reporting Services configuration modes and the options we have, see Reporting Services Configuration Options (SSRS).

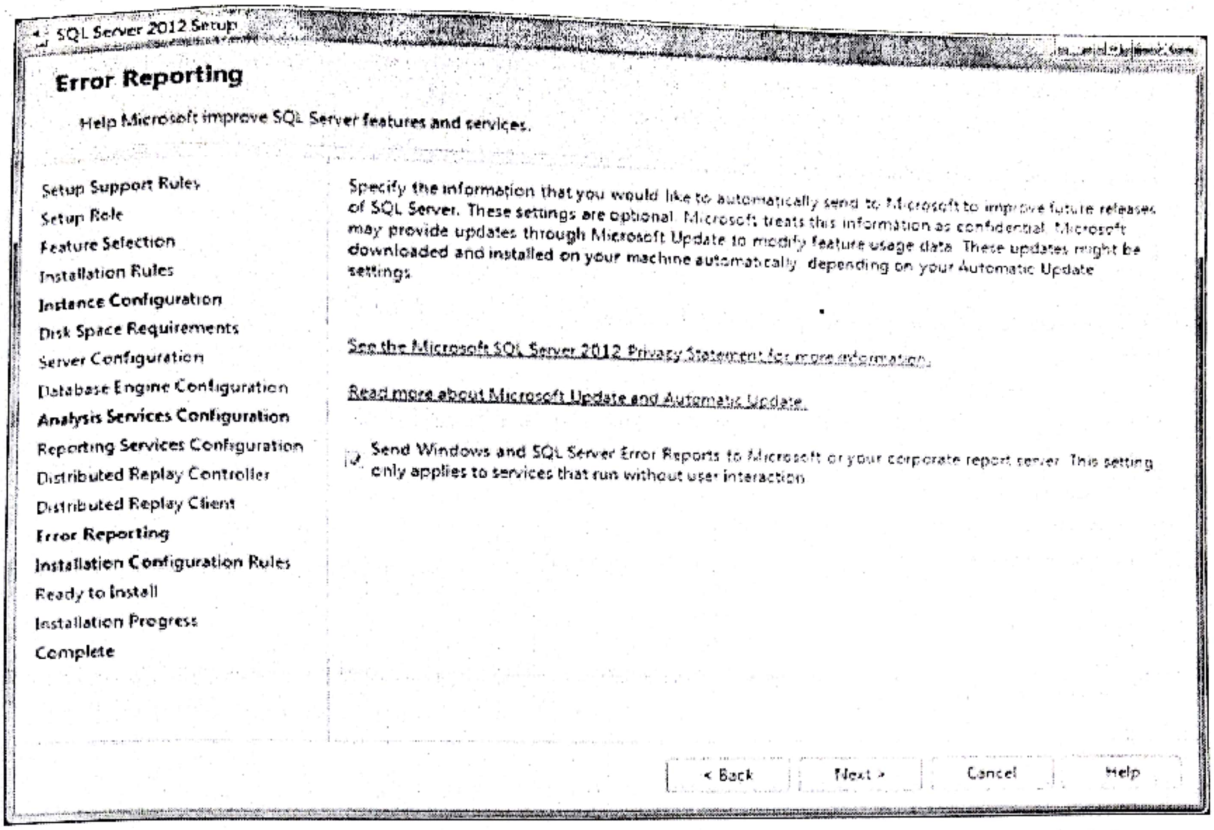
Click **Next** to proceed to Distributed Replay Controller page, specify the users you want to grant administrative permissions to for the Distributed Replay controller service and then click Next to continue:



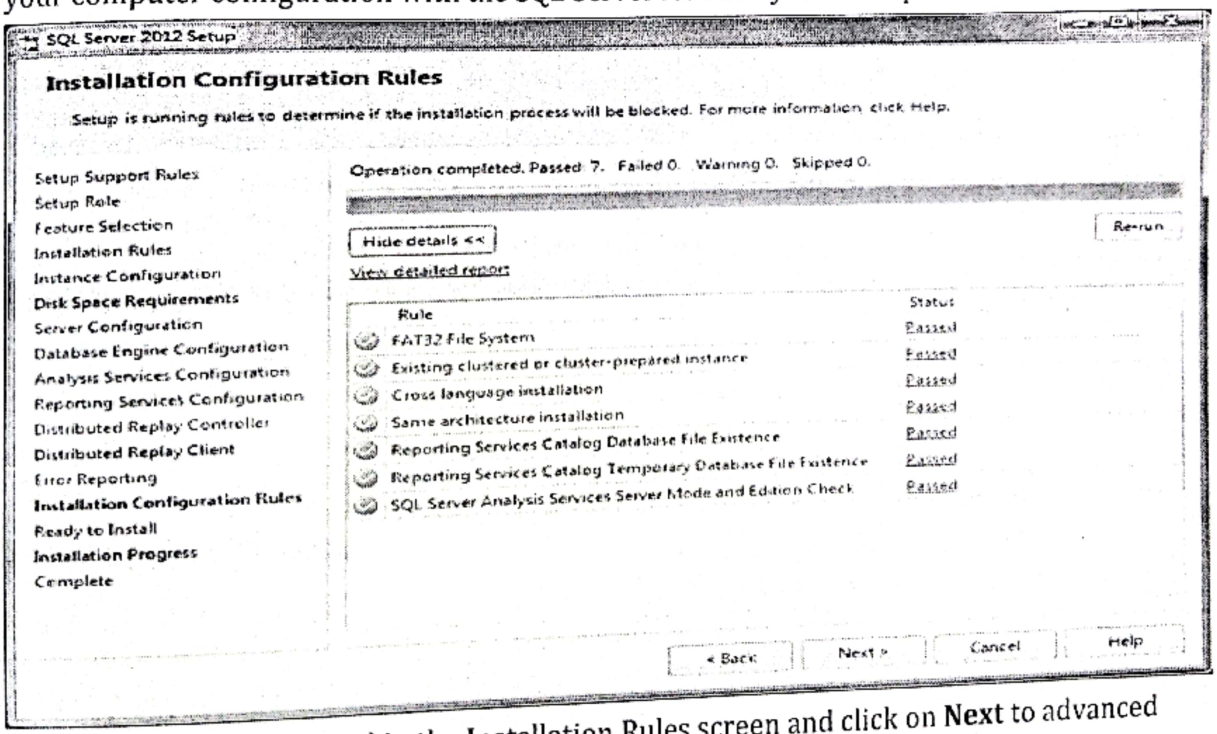
Distributed Replay Client Configuration page appears, here specify setting as shown in f igure:



Click **Next** to advance to Error Reporting page, tick the check box if you want to send Windows and SQL Server error reports to Microsoft:

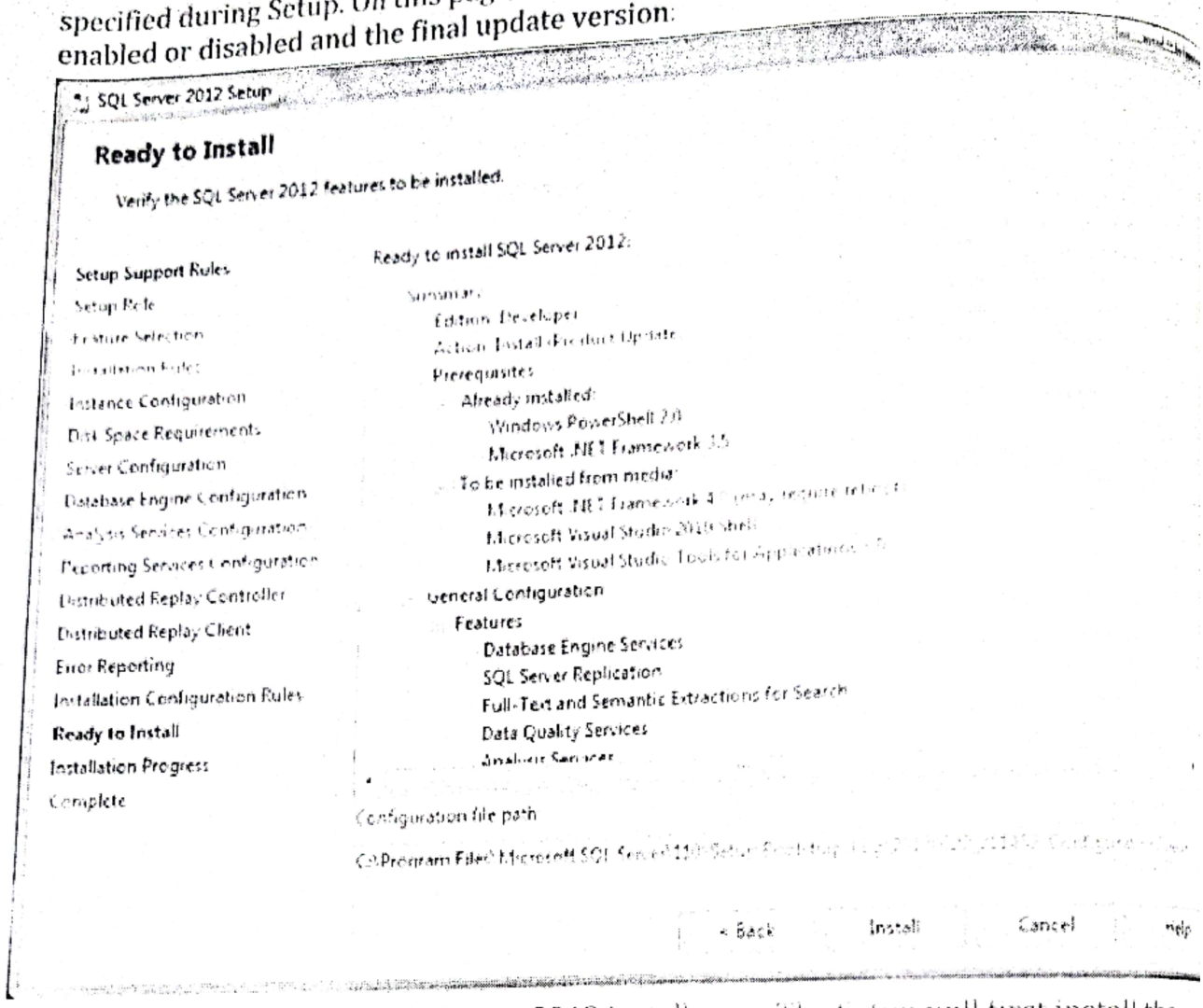


Click **Next**, Now System Configuration Checker will run some more rules that will validate your computer configuration with the SQL Server features you have specified:

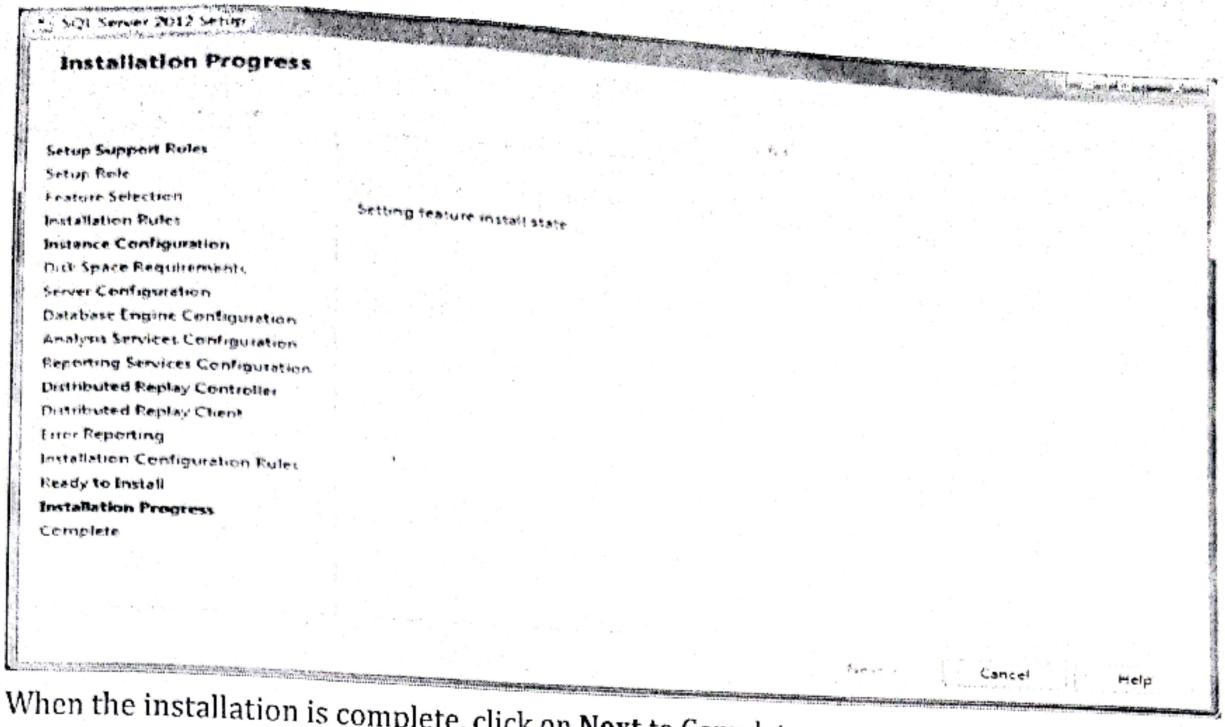


Correct any errors reported in the Installation Rules screen and click on **Next** to advanced to Ready to Install page. This page shows a tree view of installation options that were

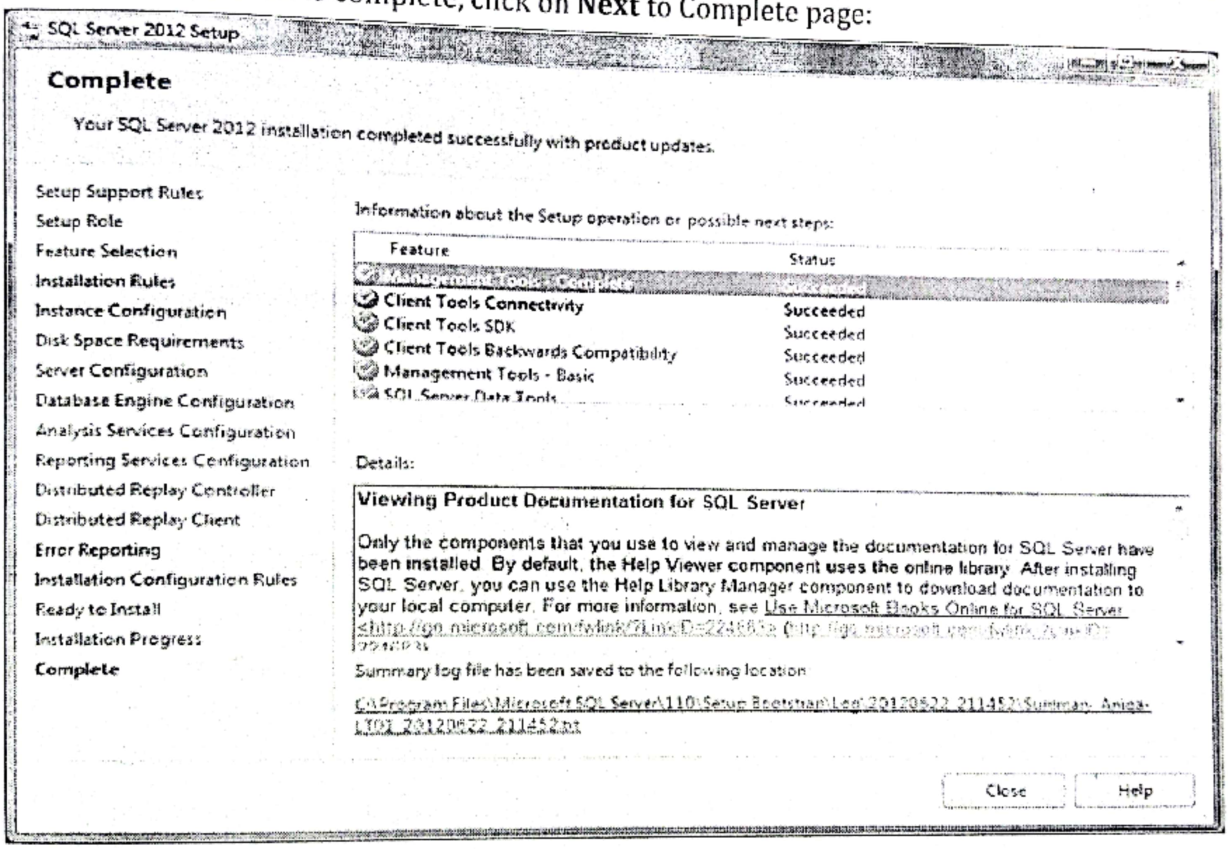
specified during Setup. On this page, Setup indicates whether the Product Update feature is enabled or disabled and the final update version:



Click **Install** button to start SQL Server 2012 installation. The Setup will first install the required prerequisites for the selected features followed by the feature installation. The Installation Progress page provides status so that you can monitor installation progress as Setup continues:



When the installation is complete, click on **Next** to Complete page:



Click on the link to review the installation log if appropriate, then click on **Close.**

This finalizes the installation process for SQL Server 2012.

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #1

rse: Database Systems

Semester: Spring 2018

uctor: NAZISH ALI

:

Maintain discipline during the lab.
Listen and follow the instructions as they are given.
Just raise hand if you have any problem.

INTRODUCTION

What is a Database?

A database is an organized collection of data. It is the collection of schemas, tables, queries, reports, views and other objects.

Real life example

we are using it every day in different situations like Food storage, Medical store, Cloths cupboard, etc. and are related to database because database is like a container.

What is a Relational Database?

A **Relational Database management System** (RDBMS) is a database management system based on relational model introduced by E.F Codd. In relational model, data is represented in terms of tuples (rows).

RDBMS is used to manage Relational database.

Relational database is a collection of organized set of tables from which data can be accessed easily. Relational Database is most commonly used database. It consists of number of tables and each table has its own primary key.

What is Table?

In Relational database, a **table** is a collection of data elements organized in terms
and columns. A table is also considered as convenient representation of **relations**
table can have duplicate tuples while a true **relation** cannot have duplicate tuples.
the simplest form of data storage. Below is an example of Employee table.

ID	Name	Age	Salary
1	Adam	34	13000
2	Alex	28	15000
3	Stuart	20	18000
4	Ross	42	19020

What is a Record?

A single entry in a table is called a **Record** or **Row**. A **Record** in a table represents set of
data. For example, the above **Employee** table has 4 records. Following is an example of si
record.

1	**Adam**	**34**	**13000**

What is Field?

A table consists of several records (row), each record can be broken into several small
entities known as **Fields**. The above **Employee** table consist of four
fields, **ID**, **Name**, **Age** and **Salary**.

What is a Column?

In **Relational** table, a column is a set of value of a particular type. The term **Attribu**
also used to represent a column. For example, in Employee table, Name is a column
represent names of employee.

Name
Adam
Alex
Stuart
Ross

What is SQL?

- SQL stands for Structured Query Language

- SQL lets you access and manipulate databases
- SQL is a standard language for accessing and manipulating databases.
- SQL is the standard language for Relational Database System.
- Used to Create, Transfer, Retrieve information from RDBMS.
- SQL is defined by ISO & ANSI.
- Simple language
- Very powerful.

What is a SQL Server?

SQL Server is a Microsoft product used to manage and store information. Technically, SQL Server is a "relational database management system" (RDMS). Broken apart, this term means two things. First, that data stored inside SQL Server will be housed in a "relational database", and second, that SQL Server is an entire "management system", not just a database. SQL itself stands for Structured Query Language. This is the language used to manage and administer the database server.

Why SQL?

SQL is widely popular because it offers the following advantages –

- Allows users to access data in the relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in a database and manipulate that data.
- Allows to embed within other languages using SQL modules, libraries & pre-compilers.
- Allows users to create and drop databases and tables.
- Allows users to create view, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures and views.

A Brief History of SQL

- **1970** – Dr. Edgar F. "Ted" Codd of IBM is known as the father of relational databases. He described a relational model for databases.
- **1974** – Structured Query Language appeared.
- **1978** – IBM worked to develop Codd's ideas and released a product named System/R.

- **1986** – IBM developed the first prototype of relational database and standar by ANSI. The first relational database was released by Relational Software w later came to be known as Oracle.

SQL Process

When you are executing an SQL command for any RDBMS, the system determines way to carry out your request and SQL engine figures out how to interpret the tas

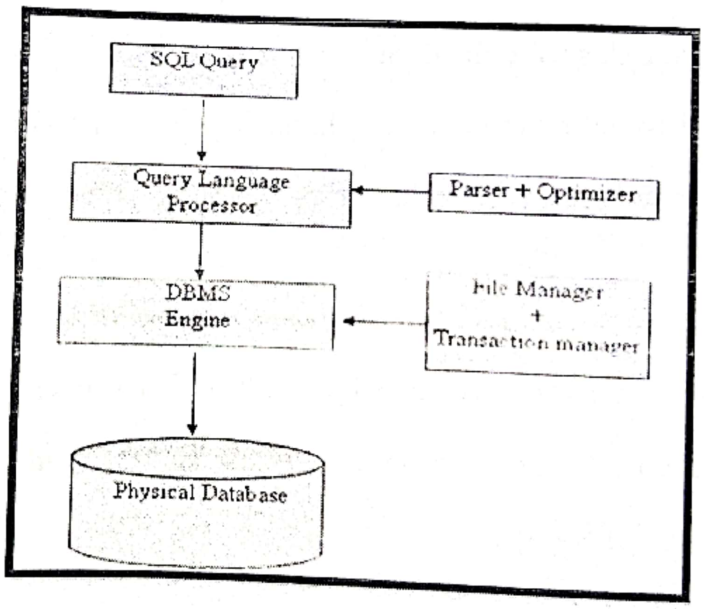
There are various components included in this process.

These components are

- Query Dispatcher
- Optimization Engines
- Classic Query Engine
- SQL Query Engine, etc.

A classic query engine handles all the non-SQL queries, but a SQL query eng handle logical files.

Following is a simple diagram showing the SQL Architecture –



What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases

- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

SQL Commands

Data Query Language (DQL)

- SELECT - Used to retrieve certain records from one or more tables.

Data Manipulation Language (DML)

- INSERT - Used to create a record.
- UPDATE - Used to change certain records.
- DELETE - Used to delete certain records.

Data Definition Language (DDL)

- CREATE - Used to create a new table, a view of a table, or other object in database.
- ALTER - Used to modify an existing database object, such as a table.
- DROP - Used to delete an entire table, a view of a table or other object in the database.

Data Control Language (DCL)

- GRANT - Used to give a privilege to someone.
- REVOKE - Used to take back privileges granted to someone.

TCL – Transaction Control Language.

- These commands are used for managing changes affecting the data.
- COMMIT, ROLLBACK, SAVEPOINT commands are used.

DOMAIN TYPES IN SQL:

The SQL standard supports a variety of built in domain types, including.

- **Char (n)** - A fixed length character length string with user specified length.
- **Varchar (n)** - A variable character length string with user specified maximum length n.
- **Int-** An integer.
- **Small integer**- A small integer.
- **Numeric (p, d)**-A Fixed point number with user defined precision.

- **Real, double precision-** Floating point and double precision floating point numb with machine dependent precision.
- **Float (n)** - A floating point number, with precision of at least n digits.
- **Date-** A calendar date containing a (four digit) year, month and day of the month
- **Time-** The time of day, in hours, minutes and seconds Eg. Time '09:30:00'.
- **Number-** Number is used to store numbers (fixed or floating point)

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

Parameters or Arguments

Expressions

The columns or calculations that you wish to retrieve. Use * if you wish to select al columns.

Tables

The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

WHERE conditions

Optional. The conditions that must be met for the records to be selected. If no conditions are provided, then all records will be selected.

ORDER BY expression

Optional. The expression used to sort the records in the result set. If more than one expression is provided, the values should be comma separated.

ASC

Optional. ASC sorts the result set in ascending order by *expression*. This is the default behavior, if no modifier is provider.

DESC

Optional. DESC sorts the result set in descending order by *expression*.

Introduction to the Query: The SELECT Statement

BACKGROUND

To fully use the power of a relational database as described briefly.

"Introduction to SQL," you need to communicate with it. The ultimate communication would be to turn to your computer and say, in a clear, distinct voice, "Show me all the left-handed, brown-eyed bean counters who have worked for this company for at least

10 years." A few of you may already be doing so (talking to your computer, not listing bean counters).

General Rules of Syntax

As you will find, syntax in SQL is quite flexible, although there are rules to follow as in any programming language. A simple query illustrates the basic syntax of an SQL select statement. Pay close attention to the case, spacing, and logical separation of the components of each query by SQL keywords

SQL: SELECT Statement

Description

The SQL SELECT statement is used to retrieve records from one or more tables in your SQL database

- The SELECT statement retrieves data from a database.
- The data is returned in a table-like structure called a result-set.
- SELECT is the most frequently used action on a database.

Syntax:

SELECT <COLUMN NAMES>

The commands, see also statements basic **SELECT** statement couldn't be simple However, **SELECT** does not work alone. If you typed just **SELECT** into your system, you might get the following response:

INPUT:

SQL> **SELECT**;

OUTPUT:

SELECT

*

ERROR at line 1:

ORA-00936: missing expression

The asterisk under the offending line indicates where SQL server thinks the offe occurred. The error message tells you that something is missing. That something is the

FROM clause:

SYNTAX:

FROM <TABLE>

Together, the statements **SELECT** and **FROM** begin to unlock the power behind your database.

The Building Blocks of Data Retrieval:

SELECT and FROM

As your experience with SQL grows, you will notice that you are typing the words **SELEC** and **FROM** more than any other words in the SQL vocabulary. They aren't as glamorous **CREATE** or as ruthless as **DROP**, but they are indispensable to any conversation you hop to have with the computer concerning data retrieval. And isn't data retrieval the reaso that you entered mountains of information into your very expensive database in the firs place?

This discussion starts with **SELECT** because most of your statements will also start with

Keywords clauses at this point you may be wondering what the

Difference is between a keyword, a statement, and a clause. SQL keywords

Refer to individual SQL elements, such as **SELECT** and **FROM**. A clause is a part of an SQL statement; for example, **SELECT** column1, column2, ... is a clause.

SQL clauses combine to form a complete SQL statement. For example, you can combine a **SELECT** clause and a **FROM** clause to write an SQL statement.

Each implementation of SQL has a unique way of indicating errors.

SELECT * FROM customer;

The asterisk (*) in **select** * tells the database to return all the columns associated with the given table described in the **FROM** clause. The database determines the order in which to return the columns.

Terminating an SQL Statement

In some implementations of SQL, the semicolon at the end of the statement tells the interpreter that you are finished writing the query. For example, Oracle's SQL*PLUS won't execute the query until it finds a semicolon (or a slash). On the other hand, some implementations of SQL do not use the semicolon as a terminator. For example,Microsoft Query and Borland's ISQL don't require a terminator, because your query is typed in an edit box and executed when you push a button.

```
1. SELECT FirstName, LastName, City
2.   FROM Customer
```

Results: 91 records

FirstName	LastName	City
Maria	Anders	Berlin
Ana	Trujillo	México D.F.
Antonio	Moreno	México D.F.
Thomas	Hardy	London

Christina Berglund Luleå

SQL WHERE Clause

- To limit the number of rows use the WHERE clause.
- The WHERE clause filters for rows that meet certain criteria.
- WHERE is followed by a condition that returns either true or false.
- WHERE is used with SELECT, UPDATE, and DELETE.

SQL WHERE Clause Examples

Problem: **List the customers in Sweden**

```
1. SELECT Id, FirstName, LastName, City, Country, Phone
2.  FROM Customer
3. WHERE Country = 'Sweden'
```

Results: 2 records

Id	FirstName	LastName	City	Country	Phone
5	Christina	Berglund	Luleå	Sweden	0921-12 34 65
24	Maria	Larsson	Bräcke	Sweden	0695-34 67 21

SUMMARY:

Database is an organized collection of data A database consists of one or more tables. A tabl is identified by its name. A table is made up of columns and rows. Columns contain the column name and data type. Rows contain the records or data for the columns.

A *Database Management System* (DBMS), is a software program that enables the creation and management of databases.
SQL stands for Structured Query Language. SQL is the standard language for querying relational databases.is used to communicate with a database.
SQL is a special-purpose programming language designed for managing data in a relationa database, and is used by a huge number of apps and organizations.

Structured Query Language (SQL) was created to shield the database programmer from understanding the specifics of how data is physically stored in each database management system and also to provide a universal foundation for updating, creating and extracting data from database systems that support an SQL interface.

SQL or Structured Query Language is a language; language that communicates with a relational database thus providing ways of manipulating and creating databases.
MySQL and Microsoft's SQL Server both are relational database management systems that use SQL as their standard relational database language.

DDL or Data Definition Language commands in SQL approaches databases from a logical perspective rather than a physical perspective. **CREATE, ALTER, DROP**

The DML statements are used to add new rows to a table, update or modify data in existing rows, or remove existing rows from a table.
WHERE is followed by a condition that returns either true or false. WHERE is used with **SELECT, UPDATE, and DELETE.**

- INSERT does just what it sounds like — it inserts new rows into a database. This is the primary tool used to build the actual records within a database. Here's an example: if a new student joins a school, that school might update its students table by inserting a new record.
- UPDATE makes changes to existing records. For example, if the database administrator accidentally mistyped our new student's name when adding her to the database using INSERT, the situation could be remedied with a quick UPDATE of the student's 'name' field.
- DELETE permanently removes records from the database. It should be used carefully, as over-ambitious DELETE commands can wipe out portions of a database you actually want to keep.

REVIEW QUESTIONS:

1. Why we use SQL server?
2. What is database and RDMS?
3. What are various DDL commands in SQL?
4. What is the purposes of DML statements in SQL?
5. What are various DCL commands in SQL? Give brief description of their purposes.
6. What is the difference between SQL and MySQL or SQL Server?

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #2

Course: Database Systems
Instructor: NAZISH ALI

Semester: Spring 2018

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.

Creating and Maintaining Tables

Objectives

Today you learn about creating databases. **CREATE DATABASE, CREATE TABLE, ALTER TABLE, DROP TABLE,** and **DROP DATABASE** statements, which are collectively known as data definition statements.

(In contrast, the **SELECT, UPDATE, INSERT,** and **DELETE** statements are described in next handout as data manipulation statements.) By the end of the day, you will understand and be able to do the following:

- l Create key fields
- l Create a database with its associated tables
- l Create, alter, and drop a table
- l Add data to the database
- l Modify the data in a database
- l Drop databases

You now know much of the SQL vocabulary and have examined the SQL query in some detail, beginning with its basic syntax. "Introduction to the Query: The **SELECT** Statement," you learned how to select data from the database.

"Manipulating Data," you learned how to insert, update, and delete data from t database in next handout. You probably have been wondering just where the Databases come from.

For simplicity's sake, we have been ignoring the process of creating databases ar tables. We have assumed that these data objects existed currently on your syste Today you finally create these objects.

The syntax of the **CREATE** statements can range from the extremely simple to t complex, depending on the options your database management system (DBM supports and how detailed you want to be when building a database.

The CREATE DATABASE Statement

The first data management step in any database project is to create the databas This task can range from the elementary to the complicated, depending on you needs and the database management system you have chosen.

Many modern systems (including Personal Oracle7) include graphical tools tha enable you to completely build the database with the click of a mouse button.

This time-saving feature is certainly helpful, but you should understand the SQ statements that execute in response to the mouse clicks. Through persona experience, we have learned the importance of creating a good SQL install script.

This script file contains the necessary SQL code to completely rebuild a database o databases; the script often includes database objects such as indexes, stored procedures, and triggers.

You will see the value of this script during development as you continually make changes to the underlying database and on occasion want to completely rebuild the database with all the latest changes.

Using the graphical tools each time you need to perform a rebuild can become extremely time-consuming.

In addition, the syntax for the typical **CREATE DATABASE** statement looks like this:

SYNTAX:

CREATE DATABASE database_name

Because the syntax varies so widely from system to system, we will not expand on the **CREATE DATABASE** statement's syntax. Many systems do not even support an SQL **CREATE DATABASE** command. However, all the popular, more powerful,

relational database management systems (RDBMSs) do provide it. Instead of focusing on its syntax, we will spend some time discussing the options to consider when creating a database.

CREATE DATABASE Options

The syntax for the **CREATE DATABASE** statement can vary widely. Many SQL texts skip over the **CREATE DATABASE** statement and move directly on to the **CREATE TABLE** statement.

Because you must create a database before you can build a table, this section focuses on some of the concepts a developer must consider when building a database. The first consideration is your level of permission.

If you are using a relational database management system (RDBMS) that supports user permissions, you must make sure that either you have system administrator-level permission settings or the system administrator has granted you **CREATE DATABASE** permission. Refer to your RDBMS documentation for more information. Most RDBMSs also allow you to specify a default database size, usually in terms of hard disk space (such as megabytes).

You will need to understand how your database system stores and locates data on the disk to accurately estimate the size you need. The responsibility for managing this space falls primarily to system administrators, and possibly at your location a database administrator will build you a test database.

Don't let the **CREATE DATABASE** statement intimidate you. At its simplest, you can create a database named **test with** the following statement:

The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a new SQL database.

CREATE DATABASE

The following SQL statement creates a database called "testDB"

Example

```
CREATE DATABASE testDB;
```

Again, be sure to consult your database management system's Documentation to learn the specifics of building a database, as the **CREATE DATABASE** statement can and does vary for the different implementations. Each implementation also has some unique options.

Database Design

Designing a database properly is extremely important to the success of yo application. The introductory material on Day 1, "Introduction to SQL," touched the topics of relational database theory and database normalization.

Normalization is the process of breaking your data into separate components reduce the repetition of data. Each level of normalization reduces the repetition data. Normalizing your data can be an extremely complex process, and numero database design tools enable you to plan this process in a logical fashion.

Many factors can influence the design of your database, including the following:

- l Security
- l Disk space available
- l Speed of database searches and retrievals
- l Speed of database updates
- l Speed of multiple-table joins to retrieve data
- l RDBMS support for temporary tables

Disk space is always an important factor. Although you may not think that disk space is a major concern in an age of multi gigabyte storage, remember that the bigger your database is, the longer it takes to retrieve records.

If you have done a poor job of designing your table structure, chances are that you have needlessly repeated much of your data. Often the opposite problem can occur.

You may have sought to completely normalize your tables' design with the database and in doing so created many tables. Although you may have approached database-design nirvana, any query operations done against this database may take a very long time to execute.

Databases designed in this manner are sometimes difficult to maintain because the table structure might obscure the designer's intent.

This problem underlines the importance of always documenting your code or design so that others can come in after you (or work with you) and have some idea of what you were thinking at the time you created your database structure.

In database designer's terms, this documentation is known as a data dictionary.

Creating a Data Dictionary

A data dictionary is the database designer's most important form of documentation. It performs the following functions:

- l Describes the purpose of the database and who will be using it.
- l Documents the specifics behind the database itself: what device it was created on, the database's default size, or the size of the log file (used to store database operations information in some RDBMSs).
- l Contains SQL source code for any database install or uninstall scripts, including documentation on the use of import/export tools, such as those introduced
- l Provides a detailed description of each table within the database and explains its purpose in business process terminology.
- l Documents the internal structure of each table, including all fields and their data types with comments, all indexes, and all views. (See Day 10, "Creating Views and Indexes.")
- l Contains SQL source code for all stored procedures and triggers.
- l Describes database constraints such as the use of unique values or **NOT NULL** values. The documentation should also mention whether these constraints are enforced at the RDBMS level or whether the database programmer is expected to check for these constraints within the source code.

Most of the major RDBMS packages come with either the data dictionary installed or scripts to install it.

Creating Key Fields

Along with documenting your database design, the most important design goal you should have is to create your table structure so that each table has a primary key and a foreign key. The primary key should meet the following goals:

- l Each record is unique within a table (no other record within the table has all of its columns equal to any other).
- l For a record to be unique, all the columns are necessary; that is, data in one column should not be repeated anywhere else in the table.

Regarding the second goal, the column that has completely unique data throughout the table is known as the *primary key field*.

A *foreign key field* is a field that links one table to another table's primary or foreign key. The following example should clarify this situation. Assume you have three tables: **BILLS**, **BANK_ACCOUNTS**, and **COMPANY**.
Table 9.1 shows the Format of these three tables.

Table 9.1. Table structure for the PAYMENTS database.

Bills Bank_Accounts Company
NAME, CHAR (30) ACCOUNT_ID, NUMBER NAME, CHAR (30)
AMOUNT, NUMBER TYPE, CHAR (30) ADDRESS, CHAR (50)
ACCOUNT_ID, NUMBER BALANCE, NUMBER CITY, CHAR (20)
BANK, CHAR (30) STATE, CHAR (2)

Take a moment to examine these tables. Which fields do you think are the prim keys?

Which are the foreign keys?

The primary key in the **BILLS** table is the **NAME** field. This field should not b duplicated because you have only one bill with this amount. (In reality, you woul probably have a check number or a date to make this record truly unique, bu assume for now that the

NAME field works.) The **ACCOUNT_ID** field in the **BANK_ACCOUNTS** table is th primary key for that table. The **NAME** field is the primary key for the **COMPAN** table. The foreign keys in this example are probably easy to spot. The **ACCOUNT_I** field in the **BILLS** table joins the **BILLS** table with the **BANK_ACCOUNTS** table.

The **NAME** field in the **BILLS** table joins the **BILLS** table with the **COMPANY** table. I this were a full-fledged database design, you would have many more tables and data breakdowns.

For instance, the **BANK** field in the **BANK_ACCOUNTS** table could point to a **BANK** table containing bank information such as addresses and phone numbers. The **COMPANY** table could be linked with another table (or database for that matter) containing information about the company and its products.

EXAMPLE

Let's take a moment to examine an incorrect database design using the same information contained in the **BILLS**, **BANK_ACCOUNTS**, and **COMPANY** tables.

A mistake many beginning users make is not breaking down their data into as many logical groups as possible. For instance, one poorly designed **BILLS** table might look like this:

Column Names Comments

NAME, CHAR (30) Name of company that bill is owed to

AMOUNT, NUMBER Amount of bill in dollars
ACCOUNT_ID, NUMBER Bank account number of bill (linked to **BANK_ACCOUNTS** table)

ADDRESS, CHAR (30) Address of company that bill is owed to
CITY, CHAR (15) City of company that bill is owed to
STATE, CHAR (2) State of company that bill is owed to

The results may look correct, but take a moment to really look at the data here. If over several months you wrote several bills to the company in the **NAME** field, each time a new record was added for a bill, the company's **ADDRESS**, **CITY**, and **STATE** information would be duplicated. Now multiply that duplication over several hundred or thousand records and then multiply that figure by 10, 20, or 30 tables. You can begin to see the importance of a properly normalized database.

Before you actually fill these tables with data, you will need to know how to create a table.

The CREATE TABLE Statement

The process of creating a table is far more standardized than the **CREATE DATABASE** Statement. Here's the basic syntax for the **CREATE TABLE** statement:

SYNTAX:

CREATE TABLE table_name
(field1 datatype [NOT NULL],
field2 datatype [NOT NULL],
field3 datatype [NOT NULL]...)
A simple example of a **CREATE TABLE** statement follows.

```
CREATE TABLE BILLS (
2 NAME CHAR(30),
3 AMOUNT NUMBER,
4 ACCOUNT_ID NUMBER);
```

Table created.

ANALYSIS:

This statement creates a table named **BILLS**. Within the **BILLS** table are three fields:

NAME, AMOUNT, and **ACCOUNT_ID**. The **NAME** field has a data type of charact and can store strings up to 30 characters long. The **AMOUNT** and **ACCOUNT_I** fields can contain number values only.
The following section examines components of the **CREATE TABLE** command.

The Table Name

When creating a table using Personal Oracle7, several constraints apply whe naming the table. First, the table name can be no more than 30 characters lon Because Oracle is case insensitive, you can use either uppercase or lowercase for th individual characters. However, the first character of the name must be a lett between A and Z.
The remaining characters can be letters or the symbols _, #, $, and @. Of course, th table name must be unique within its schema. The name also cannot be one of th Oracle or SQL reserved words (such as **SELECT**).

> You can have duplicate table names as long as the owner or schema Is different. Table names in the same schema must be unique.

The Field Name

The same constraints that apply to the table name also apply to the field name However, a field name can be duplicated within the database. The restriction is tha the field name must be unique within its table. For instance, assume that you have two tables in your database: **TABLE1**and **TABLE2**. Both of these tables could have fields called ID. You cannot, however, have two fields within **TABLE1** called **ID**, even if they are of different data types.

The Field's Data Type

If you have ever programmed in any language, you are familiar with the concept of data types, or the type of data that is to be stored in a specific field. For instance, a character data type constitutes a field that stores only character string data. Table 9.2 shows the data types supported by Personal Oracle7.

Table 9.2. Data types supported by Personal Oracle7.

Data Type Comments

CHAR Alphanumeric data with a length between 1 and 255 characters. Spaces are padded to the right of the value to supplement the total allocated length of the column. **DATE** Included as part of the date are century, year, month, day, hour, minute, and second.

LONG Variable-length alphanumeric strings up to 2 gigabytes. (See the following note.)

LONG RAW Binary data up to 2 gigabytes. (See the following note.)
NUMBER Numeric **0**, positive or negative fixed or floating-point data.
RAW Binary data up to 255 bytes. **ROWID** Hexadecimal string representing the unique address of a row in a table.
(See the following note.)
VARCHAR2 Alphanumeric data that is variable length; this field must be between 1 and 2,000 characters long.

NOTE: The **LONG** data type is often called a **MEMO** data type in other database management systems. It is primarily used to store large amounts of text for retrieval at some later time.
The **LONG RAW** data type is often called a binary large object (**BLOB**) in other database management systems. It is typically used to store graphics, sound, or video data. Although relational database management systems were not originally designed to serve this type of data, many multimedia systems today store their data in **LONG RAW**, or **BLOB**, fields.

The **ROWID** field type is used to give each record within your table a unique, non-duplicating value. Many other database systems support this concept with a **COUNTER** field (Microsoft Access) or an **IDENTITY** field (SQL Server)

Check your implementation for supported data types as they may vary.

The NULL Value

SQL also enables you to identify what can be stored within a column. A **NULL** value is almost an oxymoron, because having a field with a value of **NULL** means that the field actually has no value stored in it. When building a table, most database systems enable you to denote a column with the

NOT NULL keywords.

NOT NULL means the column cannot contain any **NULL** values for any records in the table. Conversely, **NOT NULL** means that every record must have an actual value in this column. The following example illustrates the use of the **NOT NULL** keywords.

```
CREATE TABLE BILLS (
2 NAME CHAR (30) NOT NULL,
3 AMOUNT NUMBER,
4 ACCOUNT_ID NOT NULL);
```

ANALYSIS:

In this table you want to save the name of the company you owe the money to, along with the bill's amount. If the **NAME** field and/or the **ACCOUNT ID** were not stored, the record would be meaningless. You would end up with a record with a bill, but you would have no idea whom you should pay.
The first statement in the next example inserts a valid record containing data for a bill to be sent to Joe's Computer Service for $25.

```
INSERT INTO BILLS VALUES ("Joe's Computer Service", 25, 1);
1 row inserted.

INSERT INTO BILLS VALUES ("", 25000, 1);
1 row inserted.
```

ANALYSIS:

Notice that the second record in the preceding example does not contain a NAME value. (You might think that a missing payee is to your advantage because the bill amount is $25,000, but we won't consider that.) If the table had been created with a **NOT NULL** value for the **NAME** field, the second insert would have raised an error. A good rule of thumb is that the primary key field and all foreign key fields should never contain **NULL** values.

Unique Fields

One of your design goals should be to have one unique column within each table. This column or field is a primary key field. Some database management systems allow you to set a field as unique. Other database management systems, such as Oracle and SQL Server, allow you to create a unique index on a field.

This feature keeps you from inserting duplicate key field values into the database. You should notice several things when choosing a key field. As we mentioned, Oracle provides a **ROWID** field that is incremented for each row that is added, which makes this field by default always a unique key. **ROWID** fields make excellent key fields for several reasons.

First, it is much faster to join on an integer value than on an 80-character string. Such joins result in smaller database sizes over time if you store an integer value in

every primary and foreign key as opposed to a long **CHAR** value. Another advantage is that you can use **ROWID** fields to see how a table is organized.

Also, using **CHAR** values leaves you open to a number of data entry problems. For instance, what would happen if one person entered **111 First Street**, another entered **111 1st Street**, and yet another entered **111 First St.**? With today's graphical user environments, the correct string could be entered into a list box.

When a user makes a selection from the list box, the code would convert this string to a unique ID and save this ID to the database. Now you can create the tables you used earlier today. You will use these tables for the rest of today, so you will want to fill them with some data.
Use the **INSERT** command covered yesterday to load the tables with the data in Tables 9.3, 9.4, and 9.5.

```
create database PAYMENTS;
```
Statement processed.

```
create table BILLS (
2 NAME CHAR(30) NOT NULL,
3 AMOUNT NUMBER,
4 ACCOUNT_ID NUMBER NOT NULL);
```
Table created.

```
create table BANK_ACCOUNTS (
2 ACCOUNT_ID NUMBER NOT NULL,
3 TYPE CHAR(30),
4 BALANCE NUMBER,
5 BANK CHAR(30));
```
Table created.

```
create table COMPANY (
2 NAME CHAR(30) NOT NULL,
3 ADDRESS CHAR(50),
4 CITY CHAR(30),
5 STATE CHAR(2));
```
Table created.

Table 9.3. Sample data for the BILLS table.

Name	Amount	Account_ID
Phone Company	125	1
Power Company	75	1
Record Club	25	2
Software Company	250	1
Cable TV Company	35	3

Table 9.4. Sample data for the BANK_ACCOUNTS table.

Account_ID Type Balance Band
1 Checking 500 First Federal
2 Money Market 1200 First Investor's
3 Checking 90 Credit Union

Table 9.5. Sample data for the COMPANY table.
Name Address City State
Phone Company 111 1st Street Atlanta GA
Power Company 222 2nd Street Jacksonville FL
Record Club 333 3rd Avenue Los Angeles CA
Software Company 444 4th Drive San Francisco CA
Cable TV Company 555 5th Drive Austin TX

Table Storage and Sizing

Most major RDBMSs have default settings for table sizes and table locations. If you do not specify table size and location, then the table will take the defaults. The defaults may be very undesirable, especially for large tables. The default sizes and locations will vary among the implementations. Here is an example of a **CREATE TABLE** statement with a storage clause .

```
CREATE TABLE TABLENAME
2 (COLUMN1 CHAR NOT NULL,
3 COLUMN2 NUMBER,
4 COLUMN3 DATE)
5 TABLESPACE TABLESPACE NAME
6 STORAGE
7 INITIAL SIZE,
8 NEXT SIZE,
9 MINEXTENTS value,
10 MAXEXTENTS value,
11 PCTINCREASE value);
```

Table created.

ANALYSIS:

In Oracle you can specify a tablespace in which you want the table to reside. A decision is usually made according to the space available, often by the database

administrator (DBA). **INITIAL SIZE** is the size for the initial extent of the table (the initial allocated space). **NEXT SIZE** is the value for any additional extents the table may take through growth. **MINEXTENTS** and **MAXEXTENTS** identify the minimum and maximum extents allowed for the table, and **PCTINCREASE** identifies the percentage the next extent will be
Increased each time the table grows, or takes another extent.

Creating a Table from an Existing Table

The most common way to create a table is with the **CREATE TABLE** command. However, some database management systems provide an alternative method of creating tables, using the format and data of an existing table. This method is useful when you want to select the data out of a table for temporary modification. It can also be useful when you have to create a table similar to the existing table and fill it with similar data. (You won't have to reenter all this information.) The syntax for Oracle follows.

SYNTAX:

CREATE TABLE NEW_TABLE(FIELD1, FIELD2, FIELD3)
AS (SELECT FIELD1, FIELD2, FIELD3
FROM OLD_TABLE <WHERE...>

This syntax allows you to create a new table with the same data types as those of the fields that are selected from the old table. It also allows you to rename the fields in the new table by giving them new names.

CREATE TABLE NEW_BILLS (NAME, AMOUNT, ACCOUNT_ID)
2 **AS (SELECT * FROM BILLS WHERE AMOUNT < 50);**
Table created.

ANALYSIS:

The preceding statement creates a new table (**NEW_BILLS**) with all the records from the **BILLS** table that have an **AMOUNT** less than **50**.
Some database systems also allow you to use the following syntax:

SYNTAX:

INSERT NEW_TABLE
SELECT <field1, field2... | *> from OLD_TABLE
<WHERE...>

The preceding syntax would create a new table with the exact field structure and data found in the old table. Using SQL Server's Transact-SQL language in the following

Example: illustrates this technique.

```
INSERT NEW_BILLS
1> select * from BILLS where AMOUNT < 50
2> go
```

(The **GO** statement in SQL Server processes the SQL statements in the command buffer. It is equivalent to the semicolon (;) used in Oracle7.)

The ALTER TABLE Statement

Many times your database design does not account for everything it should. Also, requirements for applications and databases are always subject to change. The

ALTER TABLE statement enables the database administrator or designer to change the structure of a table after it has been created.

The **ALTER TABLE** command enables you to do two things: l Add a column to an existing table l Modify a column that already exists
The syntax for the **ALTER TABLE** statement is as follows:

SYNTAX:

```
ALTER TABLE table_name
<ADD column_name data_type; |
MODIFY column_name data_type;>
```

The following command changes the **NAME** field of the **BILLS** table to hold 40 characters:

```
ALTER TABLE BILLS
2 MODIFY NAME CHAR(40);
Table altered.
```

You can increase or decrease the length of columns; however, you cannot decrease a column's length if the current size of one of its values is
Greater than the value you want to assign to the column length.

Here's a statement to add a new column to the **NEW_BILLS** table:

```
ALTER TABLE NEW_BILLS
2 ADD COMMENTS CHAR(80);
Table altered.
```

ANALYSIS:

This statement would add a new column named **COMMENTS** capable of holding 80 characters. The field would be added to the right of all the existing fields. Several restrictions apply to using the **ALTER TABLE** statement. You cannot use it to add or delete fields from a database.

It can change a column from **NOT NULL** to **NULL**, but not necessarily the other way around. A column specification can be changed from **NULL** to **NOT NULL** only if the column does not contain any **NULL** values. To change a column from **NOT NULL** to **NULL**, use the following syntax:

SYNTAX:

ALTER TABLE table_name MODIFY (column_name data_type NULL);

To change a column from **NULL** to **NOT NULL**, you might have to take several steps:

1. Determine whether the column has any **NULL** values.

2. Deal with any **NULL** values that you find. (Delete those records, update the column's value, and so on.)

3. Issue the **ALTER TABLE** command.

Some database management systems allow the use of the **MODIFY** clause; others do not. Still others have added other clauses to the **ALTER TABLE** statement. In Oracle, you can even alter the table's storage parameters. Check the documentation of the system you are using to determine the implementation of the **ALTER TABLE** statement.

The DROP TABLE Statement

SQL provides a command to completely remove a table from a database. The **DROP TABLE** command deletes a table along with all its associated views and indexes. After this command has been issued, there is no turning back.

The most common use of the **DROP TABLE** statement is when you have created a table for temporary use. When you have completed all operations on the table that you planned to do, issue the **DROP TABLE** statement with the following syntax:

SYNTAX:

DROP TABLE table_name;

Here's how to drop the **NEW_BILLS** table:

DROP TABLE NEW_BILLS;
Table dropped.

ANALYSIS:

Notice the absence of system prompts. This command did not ask **Are you sure? (Y/N)**. After the **DROP TABLE** command is issued, the table is permanently dele

WARNING: If you issue
DROP TABLE NEW_BILLS;

You could be dropping the incorrect table. When dropping tables, you should alw use the owner or schema name. The recommended syntax is

DROP TABLE OWNER.NEW_BILLS;

We are stressing this syntax because we once had to repair a production database from which the wrong table had been dropped. The table was not properly identified with the schema name. Restoring the database was an eight-hour job, ar we had to work until well past midnight.

The DROP DATABASE Statement

Some database management systems also provide the **DROP DATABASE** stateme which is identical in usage to the **DROP TABLE** statement. The syntax for th statement is as follows:

DROP DATABASE database_name

Don't drop the **BILLS** database now because you will use it for the rest of today.

The various relational database implementations require you to take diff-ere steps to drop a database. After the database is dropped, you will need to clean u the operating system files that compose the database.

SUMMARY:

In this handout, we are knowing the SQL syntax for this procedure enables you to apply your knowledge to other database systems.

Database is nothing but an organized form of data for easy access, storing, retrieval and managing of data. A table is a set of data that are organized in a model with

Columns and Rows. Columns can be categorized as vertical, and Rows are horizontal. A table has specified number of column called fields but can have any number of rows which is called record.

The CREATE DATABASE statement is used to create a new SQL database.
The DROP DATABASE statement is used to drop an existing SQL database.The CREATE TABLE statement is used to create a new table in a database.The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

The ALTER TABLE state*ment is used to add, delete, or modify columns in an existing table.The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

Drop Column statement is used to delete or alter/modify column
DROP command removes a table from the database and operation cannot be rolled back.

The column that has completely unique data throughout the table is known as the *primary key field.*

A *foreign key field* is a field that links one table to another table's primary or foreign key.

REVIEW QUESTIONS:

1. Why we create Database?
2. What is the difference between drop and delete?
3. Which SQL statement is used to add, modify or drop columns in a database table?
4. What happens if you omit the WHERE clause in a delete statement?
5. What are tables and Fields?

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #3

e: Database Systems

uctor: WAQAS ALI

laintain discipline during the lab.
isten and follow the instructions as they are given.
ot raise hand if you have any problem

Manipulating Data

Objectives

Today we discuss data manipulation. By the end of the day, you should

- How to manipulate data using the INSERT, UPDATE, and DELETE s
- The importance of using the WHERE clause when you are manipula
- The basics of importing and exporting data from foreign data sourc

Introduction to Data Manipulation

Statements

Up to this point you have learned how to retrieve data from a data selection criterion imaginable. After this data is retrieved, you can use it program or edit it. Week 1 focused on retrieving data. However, you m how to enter data into the database in the first place. You may also be do with data that has been edited. Today we discuss three SQL statemen to manipulate the data within a database's table. The three statements a

- The INSERT statement
- The UPDATE statement
- The DELETE statement

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #3

se: Database Systems

Semester: Spring 2018

uctor: NAZISH ALI

:

Maintain discipline during the lab.
isten and follow the instructions as they are given.
ust raise hand if you have any problem

Manipulating Data

Objectives

Today we discuss data manipulation. By the end of the day, you should understand:

- l How to manipulate data using the **INSERT**, **UPDATE**, and **DELETE** commands
- l The importance of using the **WHERE** clause when you are manipulating data
- l The basics of importing and exporting data from foreign data sources

Introduction to Data Manipulation

Statements

Up to this point you have learned how to retrieve data from a database using every selection criterion imaginable. After this data is retrieved, you can use it in an application program or edit it. Week 1 focused on retrieving data. However, you may have wondered how to enter data into the database in the first place. You may also be wondering what to do with data that has been edited. Today we discuss three SQL statements that enable you to manipulate the data within a database's table. The three statements are as follows:

- l The **INSERT** statement
- l The **UPDATE** statement
- l The **DELETE** statement

You may have used a PC-based product such as Access, dBASE IV, or FoxPro to enter
data in the past. These products come packaged with excellent tools to enter, edit,
delete records from databases. One reason that SQL provides data manipulation statem
is that it is primarily used within application programs that enable the user to edit the
using the application's own tools.

The SQL programmer needs to be able to return the data to the database using SQL
addition, most large-scale database systems are not designed with the database designe
programmer in mind. Because these systems are designed to be used in high volu
multiuser environments, the primary design emphasis is placed on the query optimizer;
data retrieval engines.

Most commercial relational database systems also provide tools for importing
exporting data. This data is traditionally stored in a delimited text file format. Ofte
format file is stored that contains information about the table being imported. Tools s
as Oracle's SQL*Loader, SQL Server's bcp (bulk copy), and Microsoft Access Import/Exp
are covered at the end of the day.

The INSERT Statement

The **INSERT** statement enables you to enter data into the database. It can be broken dow
into two statements:

INSERT...VALUES

and

INSERT...SELECT

The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.
The first way specifies both the column names and the values to be inserted:
INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)
VALUES (*value1*, *value2*, *value3*, ...);

If you are adding values for all the columns of the table, you do not need to specify the
column names in the SQL query. However, make sure the order of the values is in the sam
order as the columns in the table. The INSERT INTO syntax would be as follows:

INSERT INTO *table_name*
VALUES (*value1, value2, value3, ...*);

The basic format of the **INSERT...VALUES** statement adds a record to a table using the columns you give it and the corresponding values you instruct it to add. You must follow three rules when inserting data into a table with the **INSERT...VALUES** statement:

- l The values used must be the same data type as the fields they are being added to.
- l The data's size must be within the column's size. For instance, you cannot add an 80-character string to a 40-character column.
- l The data's location in the **VALUES** list must correspond to the location in the column list of the column it is being added to. (That is, the first value must be entered into the first column, the second value into the second column, and so on.

ustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
9	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
)	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
[	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland

Below is a selection from the "Customers" table in the Northwind sample database:

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

Example

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

stomerID	CustomerName	ContactName	Address	City	PostalCode	Country
	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

The selection from the "Customers" table will now look like this:

Did you notice that we did not insert any number into the CustomerID field?
The CustomerID column is an auto-increment field and will be generated automatically when a new record is inserted into the table.

Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.
The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

Example

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Co
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	US
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Fin
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Po
92	Cardinal	null	null	Stavanger	null	No

The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

DELETE FROM *table_name*
WHERE *condition*;

The first thing you will probably notice about the **DELETE** command is that it doesn't have a prompt. Users are accustomed to being prompted for assurance when, for instance, a directory or file is deleted at the operating system level. **Are you sure? (Y/N)** is a common question asked before the operation is performed. Using SQL, when you instruct the DBMS to delete a group of records from a table, it obeys your command without

Asking. That is, when you tell SQL to delete a group of records, it will really do it! You will learn about transaction control. Transactions are database operations that enable programmers to either **COMMIT** or **ROLLBACK** changes to the database.

These operations are very useful in online transaction-processing applications in which you want to execute a batch of modifications to the database in one logical execution. Data integrity problems will occur if operations are performed while other users are modifying the data at the same time. For now, assume that no transactions are being undertaken.

Some implementations, for example, Oracle, automatically issue a

COMMIT command when you exit SQL. Depending on the use of the **DELETE** statements **WHERE** clause, SQL can do the following:

- l Delete single rows
- l Delete multiple rows
- l Delete all rows
- l Delete no rows

Here are several points to remember when using the **DELETE** statement:

- l The **DELETE** statement cannot delete an individual field's values (use **UPDATE** instead). The **DELETE** statement deletes entire records from a single table.

ustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) that should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

Below is a selection from the "Customers" table in the Northwind sample database:

Delete All Records Using Where Clause:

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste';
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Co
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Me
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Me
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sw

The "Customers" table will now look like this:

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM *table_name*;

or:-

DELETE * FROM *table_name*;

The previous example used all three data manipulation commands--**INSERT**, **UPDATE**, and **DELETE**--to perform a set of operations on a table. The **DELETE** statement is the easiest of the three to use.

What is a NULL Value?

"Creating and Maintaining Tables," you learn how to create tables using the **SQL CREATE TABLE** statement. For now, all you need to know is that when a column is created, it can

have several different limitations placed upon it. One of these limitations is that the column should (or should not) be allowed to contain **NULL** values.

A **NULL** value means that the value is empty. It is neither a zero, in the case of an integer, nor a space, in the case of a string. Instead, no data at all exists for that record's column. If a column is defined as **NOT NULL** (that column is not allowed to

Contain a **NULL** value), you must insert a value for that column when using the **INSERT** statement. The **INSERT** is canceled if this rule is broken, and you should receive a descriptive error message concerning your error.

You could insert spaces for a null column, but these spaces will be treated as a value. **NULL** simply means nothing is there.

A field with a NULL value is a field with no value.
If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

It is very important to understand that a NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

SELECT *column_names*
FROM *table_name*
WHERE *column_name* IS NULL;

EXAMPLE:

Assume you have a **COLLECTION** table that lists all the important stuff you have collected. You can display the table's contents by writing

SELECT * FROM COLLECTION;

Which would yield this:

ITEM WORTH REMARKS

NBA ALL STAR CARDS 300 SOME STILL IN BIKE SPOKES

MALIBU BARBIE 150 TAN NEEDS WORK

STAR WARS GLASS 5.5 HANDLE CHIPPED

LOCK OF SPOUSES HAIR 1 HASN'T NOTICED BALD SPOT YET

If you wanted to add a new record to this table, you would write

```
INSERT INTO COLLECTION
2 (ITEM, WORTH, REMARKS)
3 VALUES ('SUPERMANS CAPE', 250.00, 'TUGGED ON IT');
```

1 row created.

You can execute a simple SELECT statement to verify the insertion:

```
SELECT * FROM COLLECTION;
```

ITEM WORTH REMARKS

NBA ALL STAR CARDS 300 SOME STILL IN BIKE SPOKES

MALIBU BARBIE 150 TAN NEEDS WORK

STAR WARS GLASS 5.5 HANDLE CHIPPED

LOCK OF SPOUSES HAIR 1 HASN'T NOTICED BALD SPOT YET

SUPERMANS CAPE 250 TUGGED ON IT

ANALYSIS:

The **INSERT** statement does not require column names. If the column names are n entered, SQL lines up the values with their corresponding column numbers. In other word SQL inserts the first value into the first column, the second value into the second colum and so on.

Example

The following statement inserts the values from above Example into the table:

```
INSERT INTO COLLECTION VALUES
```

2 ('STRING', 1000.00,'SOME DAY IT WILL BE VALUABLE');

1 row created.

ANALYSIS:

By issuing the same **SELECT** statement as you did in above Example, you can verify that the Insertion worked as expected:

SELECT * FROM COLLECTION;

ITEM WORTH REMARKS

---------- --------- ----------

NBA ALL STAR CARDS 300 SOME STILL IN BIKE SPOKES

MALIBU BARBIE 150 TAN NEEDS WORK

STAR WARS GLASS 5.5 HANDLE CHIPPED

LOCK OF SPOUSES HAIR 1 HASN'T NOTICED BALD SPOT YET

SUPERMANS CAPE 250 TUGGED ON IT

STRING 1000 SOME DAY IT WILL BE VALUABLE

6 rows selected.

IS NOT NULL Syntax

SELECT *column_names*
FROM *table_name*
WHERE *column_name* IS NOT NULL;

The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE *table_name*
SET *column1 = value1, column2 = value2, ...*
WHERE *condition;*

Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

Below is a selection from the "Customers" table in the Northwind sample database:

UPDATE Table

CustomerID	CustomerName	ContactName	Address	City	PostalCode
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

Example

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Co
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Ge
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Me

Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The selection from the "Customers" table will now look like this:

UPDATE Multiple Records

It is the WHERE clause that determines how many records that will be updated.

The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

Example

```
UPDATE Customers
SET ContactName='Juan'
WHERE Country='Mexico';
```

The selection from the "Customers" table will now look like this:

ustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
	Ana Trujillo Emparedados y helados	Juan	Avda. de la Constitución 2222	México D.F.	05021	Mexico
	Antonio Moreno Taquería	Juan	Mataderos 2312	México D.F.	05023	Mexico
	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

Example

UPDATE Customers

CustomerID	CustomerName	ContactName	Address	City	PostalCode
1	Alfreds Futterkiste	Juan	Obere Str. 57	Frankfurt	12209
2	Ana Trujillo Emparedados y helados	Juan	Avda. de la Constitución 2222	México D.F.	05021
3	Antonio Moreno Taquería	Juan	Mataderos 2312	México D.F.	05023
4	Around the Horn	Juan	120 Hanover Sq.	London	WA1 1DP
5	Berglunds snabbköp	Juan	Berguvsvägen 8	Luleå	S-958 22

SET ContactName='Juan';

The selection from the "Customers" table will now look like this:

SUMMARY:

- **INSERT does just what it sounds like** — it inserts new rows into a database. This is the primary tool used to build the actual records within a database.

- **UPDATE makes changes to existing records.** For example, if the database administrator accidentally mistyped our new student's name when adding her to the database using INSERT, the situation could be remedied with a quick UPDATE of the student's 'name' field.
- **DELETE permanently removes records from the database.** It should be used carefully, as over-ambitious DELETE commands can wipe out portions of a database you actually want to keep.

Transactions are database operations that enable programmers to either **COMMIT** or **ROLLBACK** changes to the database

A **NULL** value means that the value is empty. It is neither a zero, in the case of an integer, nor a space, in the case of a string. Instead, no data at all exists for that record's column.

REVIEW QUESTIONS:

1. Which SQL statement is used to extract data from a database?
2. Which SQL statement is used to update data in a database?
3. Which SQL statement is used to delete data from a database
4. Why we use insert, update statement?
5. What do you mean by NULL values?

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #4

rse: Database Systems **Semester:** Spring 2018

ructor: NAZISH ALI

:

Maintain discipline during the lab.
Listen and follow the instructions as they are given.
ust raise hand if you have any problem.

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Syntax

```
CREATE TABLE table_name (
  column1 datatype constraint,
  column2 datatype constraint,
  column3 datatype constraint,
  ....
);
```

SQL Constraints

SQL constraints are used to specify rules for the data in a table.
Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely ident each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

SQL NOT NULL Constraint

By default, a column can hold NULL values.
The NOT NULL constraint enforces a column to NOT accept NULL values.
This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.
The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values:

Example

```
CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2)
);
```

The UNIQUE constraint ensures that all values in a column are different.
Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

SQL Server

```
CREATE TABLE CUSTOMERS(

  ID  INT          NOT NULL,
  NAME VARCHAR (20)    NOT NULL,
  AGE  INT          NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2)

);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server

```
CREATE TABLE CUSTOMERS(
  ID  INT          NOT NULL,
  NAME VARCHAR (20)    NOT NULL,
  AGE  INT
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),
CONSTRAINT UC_ CUSTOMERS UNIQUE (ID,LastName)

);
```

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE CUSTOMERS
ADD UNIQUE (ID);
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE CUSTOMERS
DROP CONSTRAINT UC_CUSTOMERS;
```

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.
Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only one primary key, which may consist of single or multiple fields.

SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "CUSTOMERS" table is created.

MySQL:

```
CREATE TABLE CUSTOMERS (
  ID   INT          NOT NULL PRIMARY KEY,
  NAME VARCHAR (20)    NOT NULL,
  AGE INT          NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE CUSTOMERS (
  ID   INT          NOT NULL,
  NAME VARCHAR (20)    NOT NULL,
  AGE INT          NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2)
);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constr on multiple columns,

Use the following SQL syntax:

MySQL / SQL Server

```
CREATE TABLE CUSTOMERS (
  ID   INT          NOT NULL,
  NAME VARCHAR (20)    NOT NULL,
  AGE INT          NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2),
  CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

In the example above there is only ONE PRIMARY KEY (PK_CUSTOMERS). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server

```
ALTER TABLE CUSTOMERS
ADD PRIMARY KEY (ID);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns

Use the following SQL syntax:

MySQL / SQL Server

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT PK_CUSTOMERS PRIMARY KEY (ID,LastName);
```

If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

MySQL:

```
ALTER TABLE CUSTOMERS
DROP PRIMARY KEY;
```

SQL Server

```
ALTER TABLE CUSTOMERS
DROP CONSTRAINT PK_CUSTOMERS;
```

SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.
A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Look at the following two tables.

"CUSTOMERS" table:

CUSTOMERSID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

"Orders" table:

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

"CUSTOMERSID" column in the "Orders" table points to the "CUSTOMERSID" column in the "CUSTOMERS" table.

The "CUSTOMERS ID" column in the "Persons" table is the PRIMARY KEY in the "CUSTOMERS" table.

The "CUSTOMERS ID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "CUSTOMERS ID" column when the "Orders" table is created:

MySQL:

```
CREATE TABLE Orders (
  OrderID int NOT NULL,
```

```
  OrderNumber int NOT NULL,
  PersonID int,
  PRIMARY KEY (OrderID),
  FOREIGN KEY (CUSTOMERS ID) REFERENCES Persons(CUSTOMERS ID)
);
```

SQL Server

```
CREATE TABLE Orders (
  OrderID int NOT NULL PRIMARY KEY,
  OrderNumber int NOT NULL,
  CUSTOMERS ID int FOREIGN KEY REFERENCES CUSTOMERS (CUSTOMERS ID)
);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server

```
CREATE TABLE Orders (
  OrderID int NOT NULL,
  OrderNumber int NOT NULL,
  CUSTOMERS ID int,
  PRIMARY KEY (OrderID),
  CONSTRAINT FK_ CUSTOMERS Order FOREIGN KEY (CUSTOMERS ID)
  REFERENCES CUSTOMERS (CUSTOMERS ID)
);
```

SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "CUSTOMERS ID" column when the "Orders" table is already created, use the following SQL:

MySQL / SQL Server

```
ALTER TABLE Orders
ADD FOREIGN KEY (CUSTOMERS ID) REFERENCES CUSTOMERS (CUSTOMERS ID);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server

```
ALTER TABLE Orders
ADD CONSTRAINT FK_ CUSTOMERS Order
FOREIGN KEY (CUSTOMERS ID) REFERENCES Persons (CUSTOMERS ID);
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

MySQL:

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_ CUSTOMERS Order;
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;
```

SQL CHECK Constraint

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

SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "CUSTOMERS" table is created. The CHECK constraint ensures that you cannot have any CUSTOMERS below 18 years:

MySQL:

```
CREATE TABLE CUSTOMERS (
  ID int NOT NULL,
  LastName varchar (255) NOT NULL,
  FirstName varchar (255),
  Age int,
  CHECK (Age>=18)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE CUSTOMERS (
  ID int NOT NULL,
  LastName varchar (255) NOT NULL,
  FirstName varchar (255),
  Age int CHECK (Age>=18)
);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int,
  City varchar(255),
  CONSTRAINT CHK_ CUSTOMERS CHECK (Age>=18 AND City='Sandnes')
);
```

SQL CHECK on ALTER TABLE

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE CUSTOMERS
ADD CHECK (Age>=18);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT CHK_ CUSTOMERS Age CHECK (Age>=18 AND City='Sandnes');
```

DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

SQL Server / Oracle / MS Access:

```
ALTER TABLE CUSTOMERS
DROP CONSTRAINT CHK_CUSTOMERS Age;
```

MySQL:

```
ALTER TABLE CUSTOMERS
DROP CHECK CHK_CUSTOMERS Age;
```

SQL DEFAULT Constraint

The DEFAULT constraint is used to provide a default value for a column.
The default value will be added to all new records IF no other value is specified.

SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

My SQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE CUSTOMERS (
  ID int NOT NULL,
  LastName varchar (255) NOT NULL,
  FirstName varchar (255),
  Age int,
  City varchar (255) DEFAULT 'Sandnes'
);
```

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE ():

```
CREATE TABLE Orders (
  ID int NOT NULL,
  OrderNumber int NOT NULL,
  OrderDate date DEFAULT GETDATE ()
);
```

SQL DEFAULT on ALTER TABLE

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

MySQL:

```
ALTER TABLE CUSTOMERS
ALTER City SET DEFAULT 'Sandnes';
```

SQL Server / MS Access:

```
ALTER TABLE CUSTOMERS
ALTER COLUMN City SET DEFAULT 'Sandnes';
```

DROP a DEFAULT Constraint

To drop a DEFAULT constraint, use the following SQL:

MySQL:

```
ALTER TABLE CUSTOMERS
ALTER City DROP DEFAULT;
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE CUSTOMERS
ALTER COLUMN City DROP DEFAULT;
```

SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.
Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

> Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX *index_name*
ON *table_name* (*column1*, *column2* ...);

CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

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

The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

CREATE INDEX Example

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx_lastname
ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx_pname
ON Persons (LastName, FirstName);

DROP INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

SQL Server:

DROP INDEX *table_name.index_name*;

MySQL:

ALTER TABLE *table_name*
DROP INDEX *index_name*;

AUTO INCREMENT Field

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

Syntax for MySQL

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the " CUSTOMERS " table:

```
CREATE TABLE CUSTOMERS (
  ID int IDENTITY(1,1) PRIMARY KEY,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int
);
```

SUMMARY

SQL constraints or just constraints are rules that you define which data values are valid while doing INSERT, UPDATE, and DELETE operations.

When constraints are in place, SQL engine rejects all the transactions that break the rules, therefore, constraints help you enforce the integrity of data automatically.

ANSI SQL provides *5 major constraints are used in SQL, such as*

- **NOT NULL:** That indicates that the column must have some value and cannot be left null
- **UNIQUE:** This constraint is used to ensure that each row and column has unique value and no value is being repeated in any other row or column
- **PRIMARY KEY:** This constraint is used in association with NOT NULL and UNIQUE constraints such as on one or the combination of more than one columns to identify the particular record with a unique identity.
- **FOREIGN KEY:** It is used to ensure the referential integrity of data in the table and also matches the value in one table with another using Primary Key
- **CHECK:** It is used to ensure whether the value in columns fulfills the specified condition

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

Auto-increment allows a unique number to be generated automatically.
The starting value for IDENTITY is 1, and it will increment by 1 for each new record.

REVIEW QUESTIONS:

1. Difference between Table level and Column level constraints?
2. How to remove a constraints?
3. Explain the purpose of NOT NULL and NULL constraints.
4. How to add constraints after the table has been created? Specify the com
   used for the same purpose.
5. Explain the difference between Foreign Key and Referential key

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #5-6

Course: Database Systems

Instructor: NAZISH ALI

Semester: Spring 2018

Note:

Maintain discipline during the lab.
Listen and follow the instructions as they are given.
Just raise hand if you have any problem.

Functions: Molding the Data You Retrieve

Objectives

Today we talk about functions. Functions in SQL enable you to perform feats such as determining the sum of a column or converting all the characters of a string to uppercase. By the end of the day, you will understand and be able to use all the following:

- l Aggregate functions
- l Date and time functions
- l Arithmetic functions
- l Character functions
- l Conversion functions
- l Miscellaneous functions

These functions greatly increase your ability to manipulate the information you retrieved using the basic functions of SQL that were described earlier this week. The first five

aggregate functions, **COUNT**, **SUM**, **AVG**, **MAX**, and **MIN**, are defined in the ANSI standard. Most implementations of SQL have extensions to these aggregate functions.

SQL Server String Functions

Function	Description
ASCII	Returns the number code that represents the specific character
CHAR	Returns the ASCII character based on the number code
CHARINDEX	Returns the location of a substring in a string
CONCAT	Concatenates two or more strings together
Concat with +	Concatenates two or more strings together
DATALENGTH	Returns the length of an expression (in bytes)
LEFT	Extracts a substring from a string (starting from left)
LEN	Returns the length of the specified string
LOWER	Converts a string to lower-case
LTRIM	Removes leading spaces from a string
NCHAR	Returns the Unicode character based on the number code
PATINDEX	Returns the location of a pattern in a string
REPLACE	Replaces a sequence of characters in a string with another set of characters
RIGHT	Extracts a substring from a string (starting from right)
RTRIM	Removes trailing spaces from a string
SPACE	Returns a string with a specified number of spaces
STR	Returns a string representation of a number
STUFF	Deletes a sequence of characters from a string and then inserts another sequence of characters into the string, starting at a specified position
SUBSTRING	Extracts a substring from a string
UPPER	Converts a string to upper-case

SQL Server Numeric Functions

Function	Description
ABS	Returns the absolute value of a number
AVG	Returns the average value of an expression
CEILING	Returns the smallest integer value that is greater than or equal to a number
COUNT	Returns the count of an expression
FLOOR	Returns the largest integer value that is equal to or less than a number
MAX	Returns the maximum value of an expression
MIN	Returns the minimum value of an expression
RAND	Returns a random number or a random number within a range
ROUND	Returns a number rounded to a certain number of decimal places
SIGN	Returns a value indicating the sign of a number
SUM	Returns the summed value of an expression

SQL Server Date Functions

Function	Description
CURRENT_TIMESTAMP	Returns the current date and time
DATEADD	Returns a date after a certain time/date interval has been added
DATEDIFF	Returns the difference between two date values, based on the interval specified
DATENAME	Returns a specified part of a given date, as a string value
DATEPART	Returns a specified part of a given date, as an integer value
DAY	Returns the day of the month (from 1 to 31) for a given date
GETDATE	Returns the current date and time
GETUTCDATE	Returns the current UTC date and time
MONTH	Returns the month (from 1 to 12) for a given date
YEAR	Returns the year (as a four-digit number) for a given date

SQL Server Conversion Functions

Function	Description
CAST	Converts an expression from one data type to another
CONVERT	Converts an expression from one data type to another

SQL Server Advanced Functions

Function	Description
COALESCE	Returns the first non-null expression in a list
CURRENT_USER	Returns the name of the current user in the SQL Server database
ISDATE	Returns 1 if the expression is a valid date, otherwise 0
ISNULL	Let's you return an alternative value when an expression is NULL
ISNUMERIC	Returns 1 if the expression is a valid number, otherwise 0
NULLIF	Compares two expressions
SESSION_USER	Returns the user name of the current session in the SQL Server database
SESSIONPROPERTY	Returns the setting for a specified option of a session
SYSTEM_USER	Returns the login name information for the current user in the SQL Server database
USER_NAME	Returns the user name in the SQL Server database

Aggregate Functions

These functions are also referred to as group functions. They return a value based on the values in a column. (After all, you wouldn't ask for the average of a single field.)

The examples in this section use the table TEAMSTATS:

SELECT * FROM TEAMSTATS;

NAME POS AB HITS WALKS SINGLES DOUBLES TRIPLES HR SO

--------- --- --- ---- ----- ------- ------- ------- -- --

JONES 1B 145 45 34 31 8 1 5 10

DONKNOW 3B 175 65 23 50 10 1 4 15

WORLEY LF 157 49 15 35 8 3 3 16

DAVID OF 187 70 24 48 4 0 17 42

HAMHOCKER 3B 50 12 10 10 2 0 0 13

CASEY DH 1 0 0 0 0 0 0 1

6 rows selected.

COUNT

The function **COUNT** returns the number of rows that satisfy the condition in the **WHERE** clause. Say you wanted to know how many ball players were hitting under 350. You would type

COUNT () Syntax

SELECT COUNT (*column_name*)
FROM *table_name*
WHERE *condition*;

SELECT COUNT(*)

2 FROM TEAMSTATS

3 WHERE HITS/AB < .35;

COUNT (*)

4

To make the code more readable, try an alias:

```
SELECT COUNT (*) NUM_BELOW_350
2 FROM TEAMSTATS
3 WHERE HITS/AB < .35;
NUM_BELOW_350
------------
4
```

Would it make any difference if you tried a column name instead of the asterisk? (Notice the use of parentheses around the column names.) Try this:

INPUT/OUTPUT:

```
SQL> SELECT COUNT(NAME) NUM_BELOW_350
2 FROM TEAMSTATS
3 WHERE HITS/AB < .35;
NUM_BELOW_350
------------
4
```

The answer is no. The **NAME** column that you selected was not involved in the **WHERE** statement. If you use **COUNT** without a **WHERE** clause, it returns the number of records in the table.

```
SELECT COUNT(*)
2 FROM TEAMSTATS;
COUNT(*)
---------
6
```

SUM

SUM does just that. It returns the sum of all values in a column. To find out how many singles have been hit, type

```
SELECT SUM (SINGLES) TOTAL_SINGLES
```

```
2 FROM TEAMSTATS;

TOTAL_SINGLES
-------------
174
```

To get several sums, use

```
SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES) TOTAL_DOUBLES,
SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR
2 FROM TEAMSTATS;
TOTAL_SINGLES TOTAL_DOUBLES TOTAL_TRIPLES TOTAL_HR
------------ ------------ ------------ --------
174 32 5 29
```

To collect similar information on all 300 or better players, type

```
SELECT SUM(SINGLES) TOTAL_SINGLES, SUM(DOUBLES) TOTAL_DOUBLES,
SUM(TRIPLES) TOTAL_TRIPLES, SUM(HR) TOTAL_HR
2 FROM TEAMSTATS
3 WHERE HITS/AB >= .300;
TOTAL_SINGLES TOTAL_DOUBLES TOTAL_TRIPLES TOTAL_HR
------------ ------------ ------------ --------
164 30 5 29
```

To compute a team batting average, type

```
SELECT SUM(HITS)/SUM(AB) TEAM_AVERAGE
2 FROM TEAMSTATS;
TEAM_AVERAGE
------------
.33706294
```

SUM works only with numbers. If you try it on a nonnumerical field, you get

```
SELECT SUM(NAME)
2 FROM TEAMSTATS;
ERROR:
ORA-01722: invalid number
no rows selected
```

This error message is logical because you cannot sum a group of names.

AVG

The AVG function computes the average of a column. To find the average number of strike outs, use this:

```
SELECT AVG(SO) AVE_STRIKE_OUTS
2 FROM TEAMSTATS;
AVE_STRIKE_OUTS
---------------
16.166667
```

The following example illustrates the difference between SUM and AVG:

```
SELECT AVG(HITS/AB) TEAM_AVERAGE
2 FROM TEAMSTATS;
TEAM_AVERAGE
------------
.26803448
```

ANALYSIS:

The team was batting over 300 in the previous example! What happened? AVG computed the average of the combined column hits divided by at bats, whereas the example with SUM divided the total number of hits by the number of at bats. For example, player A gets 50 hits in 100 at bats for a .500 average. Player B gets 0 hits in 1 at bat for a 0.0 average. The average of 0.0 and 0.5 is .250. If you compute the combined average of 50

hits in 101 at bats, the answer is a respectable .495. The following statement returns the correct batting average:

```
SELECT AVG(HITS)/AVG(AB) TEAM_AVERAGE
2 FROM TEAMSTATS;
TEAM_AVERAGE
-----------
.33706294
```

Like the **SUM** function, **AVG** works only with numbers.

MAX

If you want to find the largest value in a column, use **MAX**. For example, what is the highest number of hits?

```
SELECT MAX(HITS)
2 FROM TEAMSTATS;
MAX(HITS)
---------
70
```

Can you find out who has the most hits?

```
SELECT NAME
2 FROM TEAMSTATS
3 WHERE HITS = MAX(HITS);
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Unfortunately, you can't. The error message is a reminder that this group function (remember that *aggregate functions* are also called *group functions*) does not work in the **WHERE** clause. Don't despair, Day 7, "Subqueries: The Embedded **SELECT** Statement," covers the concept of subqueries and explains a way to find who has the **MAX** hits. What happens if you try a nonnumerical column?

```
SELECT MAX(NAME)
2 FROM TEAMSTATS;
MAX(NAME)
----------
WORLEY
```

Here's something new. **MAX** returns the highest (closest to Z) string. Finally, a function that works with both characters and numbers.

MIN

MIN does the expected thing and works like **MAX** except it returns the lowest member of a column. To find out the fewest at bats, type

```
SELECT MIN(AB)
2 FROM TEAMSTATS;
MIN(AB)
---------
1
```

The following statement returns the name closest to the beginning of the alphabet:

```
SELECT MIN(NAME)
2 FROM TEAMSTATS;
MIN(NAME)
----------
CASEY
```

You can combine **MIN** with **MAX** to give a range of values. For example:

```
SELECT MIN(AB), MAX(AB)
```

```
2 FROM TEAMSTATS;

MIN(AB) MAX(AB)
-------- --------
1 187
```

This sort of information can be useful when using statistical functions.

As we mentioned in the introduction, the first five aggregate functions are described in the ANSI standard. The remaining aggregate functions have become de facto standards, present in all important implementations of SQL.

VARIANCE

VARIANCE produces the square of the standard deviation, a number vital to many statistical calculations. It works like this:

```
SELECT VARIANCE(HITS)
2 FROM TEAMSTATS;

VARIANCE(HITS)
--------------
802.96667
```

If you try a string

```
SELECT VARIANCE(NAME)
2 FROM TEAMSTATS;

ERROR:
ORA-01722: invalid number
no rows selected
```

you find that **VARIANCE** is another function that works exclusively with numbers.

STDDEV

The final group function, **STDDEV**, finds the standard deviation of a column of numbers, as demonstrated by this example:

```
SELECT STDDEV(HITS)
2 FROM TEAMSTATS;
STDDEV(HITS)
-----------
28.336666
```

It also returns an error when confronted by a string:

```
SELECT STDDEV(NAME)
2 FROM TEAMSTATS;
ERROR:
ORA-01722: invalid number
no rows selected
```

These aggregate functions can also be used in various combinations:

```
SELECT COUNT(AB),
2 AVG(AB),
3 MIN(AB),
4 MAX(AB),
5 STDDEV(AB),
6 VARIANCE(AB),
7 SUM(AB)
8 FROM TEAMSTATS;
COUNT(AB) AVG(AB) MIN(AB) MAX(AB) STDDEV(AB) VARIANCE(AB) SUM(AB)
--------- ------- ------- ------- ---------- ------------ -------
6 119.167 1 187 75.589 5712.97 715
```

The next time you hear a sportscaster use statistics to fill the time between plays, youwill know that SQL is at work somewhere behind the scenes.

Date and Time Functions

We live in a civilization governed by times and dates, and most major implementations of SQL have functions to cope with these concepts. This section uses the table **PROJECT** to

Demonstrate the time and date functions.

SELECT * FROM PROJECT;

```
TASK STARTDATE ENDDATE
-------------- --------- ---------
KICKOFF MTG 01-APR-95 01-APR-95
TECH SURVEY 02-APR-95 01-MAY-95
USER MTGS 15-MAY-95 30-MAY-95
DESIGN WIDGET 01-JUN-95 30-JUN-95
CODE WIDGET 01-JUL-95 02-SEP-95
TESTING 03-SEP-95 17-JAN-96
```

6 rows selected.

This table used the Date data type. Most implementations of SQLhave a Da data type, but the exact syntax may vary.

ADD_MONTHS

This function adds a number of months to a specified date. For example, say something extraordinary happened, and the preceding project slipped to the right by two months

You could make a new schedule by typing

```
SELECT TASK,
2 STARTDATE,
3 ENDDATE ORIGINAL_END,
4 ADD_MONTHS(ENDDATE,2)
5 FROM PROJECT;
TASK STARTDATE ORIGINAL_ ADD_MONTH
-------------- --------- --------- ---------
KICKOFF MTG 01-APR-95 01-APR-95 01-JUN-95
TECH SURVEY 02-APR-95 01-MAY-95 01-JUL-95
USER MTGS 15-MAY-95 30-MAY-95 30-JUL-95
DESIGN WIDGET 01-JUN-95 30-JUN-95 31-AUG-95
```

CODE WIDGET 01-JUL-95 02-SEP-95 02-NOV-95

TESTING 03-SEP-95 17-JAN-96 17-MAR-96

6 rows selected.

Not that a slip like this is possible, but it's nice to have a function that makes it so easy. **ADD_MONTHS** also works outside the **SELECT** clause.

```
SELECT TASK TASKS_SHORTER_THAN_ONE_MONTH
2 FROM PROJECT
3 WHERE ADD_MONTHS(STARTDATE,1) > ENDDATE;
```

Produces the following result:

```
TASKS_SHORTER_THAN_ONE_MONTH
----------
KICKOFF MTG
TECH SURVEY
USER MTGS
DESIGN WIDGET
```

ANALYSIS:

You will find that all the functions in this section work in more than one place. However, **ADD MONTHS** does not work with other data types like character or number without the help of functions **TO_CHAR** and **TO_DATE**, which are discussed later today.

LAST_DAY

LAST_DAY returns the last day of a specified month. It is for those of us who haven't mastered the "Thirty days has September..." rhyme--or at least those

of us who have not yet taught it to our computers. If, for example, you need to know what the last day of the month is in the column **ENDDATE**, you would type

```
SELECT ENDDATE, LAST_DAY(ENDDATE)
  2 FROM PROJECT;
```

Here's the result:

```
ENDDATE   LAST_DAY(ENDDATE)
--------- ---------------
01-APR-95 30-APR-95
01-MAY-95 31-MAY-95
30-MAY-95 31-MAY-95
30-JUN-95 30-JUN-95
02-SEP-95 30-SEP-95
17-JAN-96 31-JAN-96
6 rows selected.
```

MONTHS_BETWEEN

If you need to know how many months fall between month x and month y, use **MONTHS_BETWEEN** like this:

```
SELECT TASK, STARTDATE, ENDDATE, MONTHS_BETWEEN (STARTDATE, ENDDATE)
DURATION
2 FROM PROJECT;
TASK          STARTDATE ENDDATE   DURATION
------------- --------- --------- --------
KICKOFF MTG   01-APR-95 01-APR-95         0
TECH SURVEY   02-APR-95 01-MAY-95 -.9677419
USER MTGS     15-MAY-95 30-MAY-95 -.483871
DESIGN WIDGET 01-JUN-95 30-JUN-95 -.9354839
CODE WIDGET   01-JUL-95 02-SEP-95 -2.032258
```

of us who have not yet taught it to our computers. If, for example, you need to know what the last day of the month is in the column **ENDDATE**, you would type

```
SELECT ENDDATE, LAST_DAY(ENDDATE)
2 FROM PROJECT;
```

Here's the result:

```
ENDDATE LAST_DAY(ENDDATE)
--------- ---------------
01-APR-95 30-APR-95
01-MAY-95 31-MAY-95
30-MAY-95 31-MAY-95
30-JUN-95 30-JUN-95
02-SEP-95 30-SEP-95
17-JAN-96 31-JAN-96
6 rows selected.
```

MONTHS_BETWEEN

If you need to know how many months fall between month x and month y, use **MONTHS_BETWEEN** like this:

```
SELECT TASK, STARTDATE, ENDDATE, MONTHS_BETWEEN (STARTDATE, ENDDATE)
DURATION
2 FROM PROJECT;
TASK STARTDATE ENDDATE DURATION
---------- --------- --------- ---------
KICKOFF MTG 01-APR-95 01-APR-95 0
TECH SURVEY 02-APR-95 01-MAY-95 -.9677419
USER MTGS 15-MAY-95 30-MAY-95 -.483871
DESIGN WIDGET 01-JUN-95 30-JUN-95 -.9354839
CODE WIDGET 01-JUL-95 02-SEP-95 -2.032258
```

TESTING 03-SEP-95 17-JAN-96 -4.451613

6 rows selected.

SELECT TASK, STARTDATE, ENDDATE,

2 MONTHS_BETWEEN(ENDDATE,STARTDATE) DURATION

3 FROM PROJECT;

TASK STARTDATE ENDDATE DURATION

---------- --------- --------- ---------

KICKOFF MTG 01-APR-95 01-APR-95 0

TECH SURVEY 02-APR-95 01-MAY-95 .96774194

USER MTGS 15-MAY-95 30-MAY-95 .48387097

DESIGN WIDGET 01-JUN-95 30-JUN-95 .93548387

CODE WIDGET 01-JUL-95 02-SEP-95 2.0322581

TESTING 03-SEP-95 17-JAN-96 4.4516129

6 rows selected.

ANALYSIS:

That's better. You see that **MONTHS_BETWEEN** is sensitive to the way you order the months. Negative months might not be bad. For example, you could use a negative result to determine whether one date happened before another. For example, the following statement shows all the tasks that started before May 19, 1995:

SELECT *

2 FROM PROJECT

3 WHERE MONTHS_BETWEEN('19 MAY 95', STARTDATE) > 0;

TASK STARTDATE ENDDATE

---------- --------- ---------

KICKOFF MTG 01-APR-95 01-APR-95

TECH SURVEY 02-APR-95 01-MAY-95

USER MTGS 15-MAY-95 30-MAY-9

NEXT_DAY

NEXT_DAY finds the name of the first day of the week that is equal to or la
than another specified date. For example, to send a report on the Friday
following the first day of each event, you would type

```
SELECT STARTDATE,
2 NEXT_DAY(STARTDATE, 'FRIDAY')
3 FROM PROJECT;
```

which would return

```
STARTDATE NEXT_DAY(
--------- ---------
01-APR-95 07-APR-95
02-APR-95 07-APR-95
15-MAY-95 19-MAY-95
01-JUN-95 02-JUN-95
01-JUL-95 07-JUL-95
03-SEP-95 08-SEP-95
6 rows selected.
```

ANALYSIS:

The output tells you the date of the first Friday that occurs after your **STARTDATE.**

SYSDATE

SYSDATE returns the system time and date:

```
SELECT DISTINCT SYSDATE
2 FROM PROJECT;
SYSDATE
---------
```

18-JUN-95 1020PM

If you wanted to see where you stood today in a certain project, you could type

```
SELECT *
2 FROM PROJECT
3 WHERE STARTDATE > SYSDATE;

TASK STARTDATE ENDDATE
---------- --------- ---------
CODE WIDGET 01-JUL-95 02-SEP-95
TESTING 03-SEP-95 17-JAN-96
```

Now you can see what parts of the project start after today.

Arithmetic Functions

Many of the uses you have for the data you retrieve involve mathematics. Most implementations of SQL provide arithmetic functions similar to the functions covered here. The examples in this section use the **NUMBERS** table:

```
SELECT *
2 FROM NUMBERS;

A B
--------- ---------
3.1415 4
-45 .707
5 9
-57.667 42
15 55
-7.2 5.3
6 rows selected.
```

ABS

The **ABS** function returns the absolute value of the number you point to. For example:

SELECT ABS(A) ABSOLUTE_VALUE

2 FROM NUMBERS;

ABSOLUTE_VALUE

3.1415

45

5

57.667

15

7.2

6 rows selected.

ABS changes all the negative numbers to positive and leaves positive number alone.

CEIL and FLOOR

CEIL returns the smallest integer greater than or equal to its argument. **FLOOR** does just the reverse, returning the largest integer equal to or less than its argument.

EXAMPLE:

SELECT B, CEIL(B) CEILING

2 FROM NUMBERS;

B CEILING

--------- ---------

4 4

.707 1

9 9

42 42

55 55

5.3 6

6 rows selected.

And

SELECT A, FLOOR (A) FLOOR

2 FROM NUMBERS;

A FLOOR

--------- ---------

3.1415 3

-45 -45

5 5

-57.667 -58

15 15

-7.2 -8

6 rows selected.

COS, COSH, SIN, SINH, TAN, and TANH

The **COS**, **SIN**, and **TAN** functions provide support for various trigonometric concepts. They all work on the assumption that n is in radians. The following statement returns some unexpected values if you don't realize **COS** expects **A** to be in radians.

INPUT:

SELECT A, COS(A)

2 FROM NUMBERS;

A COS(A)

--------- ---------

3.1415 -1

-45 .52532199

5 .28366219

-57.667 .437183

15 -.7596879

-7.2 .60835131

ANALYSIS:

You would expect the cos of 45 degrees to be in the neighborhood of .707, no .525. To make this function work the way you would expect it to in a degree-oriented world, you need to convert degrees to radians. (When was the last time you heard a news broadcast report that a politician had done a pi-radi turn? You hear about a 180-degree turn.)

Because 360 degrees = 2 pi radians, you can write

SELECT A, COS(A* 0.01745329251994)

2 FROM NUMBERS;

A COS(A*0.01745329251994)

--------- ----------

3.1415 .99849724

-45 .70710678

5 .9961947

-57.667 .5348391

15 .96592583

-7.2 .9921147

ANALYSIS:

Note that the number **0.01745329251994** is radians divided by degrees. The trigonometric

Functions work as follows:

SELECT A, COS(A*0.017453), COSH(A*0.017453)

2 FROM NUMBERS;

A COS(A*0.017453) COSH(A*0.017453)

---------- ---------- ----------

3.1415 .99849729 1.0015035

-45 .70711609 1.3245977

5 .99619483 1.00381

-57.667 .53485335 1.5507072

15 .96592696 1.0344645

-7.2 .99211497 1.0079058

6 rows selected.

And

SELECT A, SIN(A*0.017453), SINH(A*0.017453)

2 FROM NUMBERS;

A SIN(A*0.017453) SINH(A*0.017453)

--------- ---------- ----------

3.1415 .05480113 .05485607

-45 -.7070975 -.8686535

5 .08715429 .0873758

-57.667 -.8449449 -1.185197

15 .25881481 .26479569

-7.2 -.1253311 -.1259926

6 rows selected.

And

SELECT A, TAN(A*0.017453), TANH(A*0.017453)

2 FROM NUMBERS;

A TAN(A*0.017453) TANH(A*0.017453)

--------- ---------- ----------

3.1415 .05488361 .05477372

-45 -.9999737 -.6557867

5 .08748719 .08704416

-57.667 -1.579769 -.7642948

15 .26794449 .25597369

-7.2 -.1263272 -.1250043

6 rows selected.

EXP

EXP enables you to raise *e* (*e* is a mathematical constant used in various formulas) to a power. Here's how **EXP** raises *e* by the values in column A:

SELECT A, EXP(A)

2 FROM NUMBERS;

A EXP(A)

--------- ---------

3.1415 23.138549

-45 2.863E-20

5 148.41316

-57.667 9.027E-26

15 3269017.4

-7.2 .00074659

6 rows selected.

LN and LOG

These two functions center on logarithms. LN returns the natural logarithm of its argument. For example:

SELECT A, LN(A)

2 FROM NUMBERS;

ERROR:

ORA-01428: argument '-45' is out of range

Did we neglect to mention that the argument had to be positive? Write

SELECT A, LN(ABS(A))

2 FROM NUMBERS;

A LN(ABS(A))

--------- ----------

3.1415 1.1447004

-45 3.8066625

5 1.6094379

-57.667 4.0546851

15 2.7080502

-7.2 1.974081

6 rows selected.

ANALYSIS:

Notice how you can embed the function **ABS** inside the **LN** call. The other logarith-mic function, **LOG**, takes two arguments, returning the logarithm of the first argument in the base of the second. The following query returns the logarithms of column B in base 10.

SELECT B, LOG(B, 10)

2 FROM NUMBERS;

B LOG(B,10)

---------- ---------

4 1.660964

.707 -6.640962

9 1.0479516

42 .61604832

55 .57459287

5.3 1.3806894

6 rows selected.

MOD

You have encountered **MOD** before. On Day 3, "Expressions, Conditions, and Operators," you saw that the ANSI standard for the modulo operator **%** is sometimes implemented as the function **MOD**. Here's a query that returns a table showing the remainder of A divided by B:

SELECT A, B, MOD(A,B)

2 FROM NUMBERS;

A B MOD(A,B)

--------- --------- ---------

3.1415 4 3.1415

-45 .707 -.459

5 9 5

-57.667 42 -15.667

15 55 15

-7.2 5.3 -1.9

6 rows selected.

POWER

To raise one number to the power of another, use **POWER**. In this function the first argument is raised to the power of the second:

SELECT A, B, POWER(A,B)

2 **FROM NUMBERS;**

ERROR:

ORA-01428: argument '-45' is out of range

ANALYSIS:

At first glance you are likely to think that the first argument can't be negative. But that impression can't be true, because a number like -4 can be raised to a power.

Therefore, if the first number in the **POWER** function is negative, the second must be an integer. You can work around this problem by using **CEIL** (or **FLOOR**):

SELECT A, CEIL(B), POWER(A,CEIL(B))

2 **FROM NUMBERS;**

A CEIL(B) POWER(A,CEIL(B))

--------- --------- ----------

3.1415 4 97.3976

-45 1 -45

5 9 1953125

-57.667 42 9.098E+73

15 55 4.842E+64

-7.2 6 139314.07

6 rows selected.

That's better!

SIGN

SIGN returns **-1** if its argument is less than **0**, **0** if its argument is equal to **0**, and **1** if its argument is greater than **0**, as shown in the following example:

```
SELECT A, SIGN(A)
2 FROM NUMBERS;

A SIGN(A)
--------- ---------
3.1415 1
-45 -1
5 1
-57.667 -1
15 1
-7.2 -1
0 0

7 rows selected.
```

You could also use **SIGN** in a **SELECT WHERE** clause like this:

```
SELECT A
2 FROM NUMBERS
3 WHERE SIGN(A) = 1;

A
---------
3.1415
5
15
```

SQRT

The function **SQRT** returns the square root of an argument. Because the square root of a negative number is undefined, you cannot use **SQRT** on negative numbers.

```
SELECT A, SQRT(A)
2 FROM NUMBERS;

ERROR:
```

ORA-01428: argument '-45' is out of range

However, you can fix this limitation with ABS:

SELECT ABS(A), SQRT(ABS(A))

2 **FROM NUMBERS;**

ABS(A) SQRT(ABS(A))

--------- -----------

3.1415 1.7724277

45 6.7082039

5 2.236068

57.667 7.5938791

15 3.8729833

7.2 2.6832816

0 0

7 rows selected.

Character Functions

Many implementations of SQL provide functions to manipulate characters and strings of characters. This section covers the most common character functions. The examples in this section use the table **CHARACTERS**.

SELECT * FROM CHARACTERS;

LASTNAME FIRSTNAME M CODE

---------- ---------- - ---------

PURVIS KELLY A 32

TAYLOR CHUCK J 67

CHRISTINE LAURA C 65

ADAMS FESTER M 87

COSTALES ARMANDO A 77

KONG MAJOR G 52

6 rows selected.

CHR

CHR returns the character equivalent of the number it uses as an argument. The character it returns depends on the character set of the database. For this example the database is set to ASCII. The column **CODE** includes numbers.

SELECT CODE, CHR(CODE)

2 FROM CHARACTERS;

OUTPUT:

CODE CH

--------- --

32

67 C

65 A

87 W

77 M

52 4

6 rows selected.

The space opposite the **32** shows that **32** is a space in the ASCII character set.

CONCAT

You used the equivalent of this function on Day 3, when you learned about operators. The || symbol splices two strings together, as does **CONCAT**. It works like this:

SELECT CONCAT(FIRSTNAME, LASTNAME) "FIRST AND LAST NAMES"

2 FROM CHARACTERS;

FIRST AND LAST NAMES

KELLY PURVIS

CHUCK TAYLOR

LAURA CHRISTINE

FESTER ADAMS

ARMANDO COSTALES

MAJOR KONG

6 rows selected.

ANALYSIS:

Quotation marks surround the multiple-word alias **FIRST AND LAST NAMES**. Again, it is safest to check your implementation to see if it allows multiple-word aliases. Also notice that even though the table looks like two separate columns, what you are seeing is one column. The first value you concatenated, **FIRSTNAME**, is 15 characters wide. This operation retained all the characters in the field.

INITCAP

INITCAP capitalizes the first letter of a word and makes all other characters lowercase.

```
SELECT FIRSTNAME BEFORE, INITCAP(FIRSTNAME) AFTER
2 FROM CHARACTERS;
```

BEFORE AFTER

--------- ---------

KELLY Kelly

CHUCK Chuck

LAURA Laura

FESTER Fester

ARMANDO Armando

MAJOR Major

6 rows selected.

LOWER and UPPER

As you might expect, **LOWER** changes all the characters to lowercase; **UPPER** does just the reverse.

The following example starts by doing a little magic with the **UPDATE** function (you learn more about this next week) to change one of the values to lowercase:

```
UPDATE CHARACTERS
2 SET FIRSTNAME = 'kelly'
3 WHERE FIRSTNAME = 'KELLY';
```

1 row updated.

SELECT FIRSTNAME

2 FROM CHARACTERS;

FIRSTNAME

---

kelly

CHUCK

LAURA

FESTER

ARMANDO

MAJOR

6 rows selected.

Then you write

SELECT FIRSTNAME, UPPER(FIRSTNAME), LOWER(FIRSTNAME)

2 FROM CHARACTERS;

FIRSTNAME UPPER(FIRSTNAME LOWER(FIRSTNAME

--- --- ---

kelly KELLY kelly

CHUCK CHUCK chuck

LAURA LAURA laura

FESTER FESTER fester

ARMANDO ARMANDO armando

MAJOR MAJOR major

6 rows selected.

Now you see the desired behavior.

LPAD and RPAD

LPAD and **RPAD** take a minimum of two and a maximum of three arguments. The first argument is the character string to be operated on. The second is the number of characters

to pad it with, and the optional third argument is the character to pad it with. The third argument defaults to a blank, or it can be a single character or a character string. The following statement adds five pad characters, assuming that the field **LASTNAME** is def as a 15-character field:

```
SELECT LASTNAME, LPAD(LASTNAME,20,'*')
2 FROM CHARACTERS;
```

OUTPUT:

```
LASTNAME LPAD(LASTNAME,20,'*'
---------- --------------------
PURVIS *****PURVIS
TAYLOR *****TAYLOR
CHRISTINE *****CHRISTINE
ADAMS *****ADAMS
COSTALES *****COSTALES
KONG *****KONG
6 rows selected.
```

ANALYSIS:

Why were only five pad characters added? Remember that the **LASTNAME** column is 15 *characters wide and that* **LASTNAME** *includes the blanks to the right of the characters tha make up the name. Some column data types eliminate padding characters if the width of the column value is less than the total width allocated for the column.*

Check your implementation. Now try the right side:

```
SELECT LASTNAME, RPAD(LASTNAME,20,'*')
2 FROM CHARACTERS;
LASTNAME RPAD(LASTNAME,20,'*'
---------- --------------------
PURVIS PURVIS *****
TAYLOR TAYLOR *****
CHRISTINE CHRISTINE *****
ADAMS ADAMS *****
```

COSTALES COSTALES *****

KONG KONG *****

6 rows selected.

ANALYSIS:

Here you see that the blanks are considered part of the field name for these operations. The next two functions come in handy in this type of situation.

REPLACE

REPLACE does just that. Of its three arguments, the first is the string to be searched. The second is the search key. The last is the optional replacement string. If the third argument is left out or **NULL**, each occurrence of the search key on the string to be searched is removed and is not replaced with anything.

SELECT LASTNAME, REPLACE(LASTNAME, 'ST') REPLACEMENT

2 FROM CHARACTERS;

LASTNAME REPLACEMENT

--------------- ---------------

PURVIS PURVIS

TAYLOR TAYLOR

CHRISTINE CHRIINE

ADAMS ADAMS

COSTALES COALES

KONG KONG

6 rows selected.

If you have a third argument, it is substituted for each occurrence of the search key in the target string. For example:

SELECT LASTNAME, REPLACE(LASTNAME, 'ST','') REPLACEMENT**

2 FROM CHARACTERS;

LASTNAME REPLACEMENT

--------------- ------------

PURVIS PURVIS

TAYLOR TAYLOR

CHRISTINE CHRI**INE

ADAMS ADAMS

COSTALES CO**ALES

KONG KONG

6 rows selected.

If the second argument is NULL, the target string is returned with no changes.

SELECT LASTNAME, REPLACE(LASTNAME, NULL) REPLACEMENT

2 FROM CHARACTERS;

LASTNAME REPLACEMENT

---------- ----------

PURVIS PURVIS

TAYLOR TAYLOR

CHRISTINE CHRISTINE

ADAMS ADAMS

COSTALES COSTALES

KONG KONG

6 rows selected.

SUBSTR

This three-argument function enables you to take a piece out of a target string. The first argument is the target string. The second argument is the position of the first character to be output. The third argument is the number of characters to show.

SELECT FIRSTNAME, SUBSTR(FIRSTNAME,2,3)

2 FROM CHARACTERS;

FIRSTNAME SUB

---------- ---

kelly ell

CHUCK HUC

LAURA AUR

FESTER EST

ARMANDO RMA

MAJOR AJO

6 rows selected.

If you use a negative number as the second argument, the starting point is determined by counting backwards from the end, like this:

SELECT FIRSTNAME, SUBSTR(FIRSTNAME,-13,2)

2 FROM CHARACTERS;

FIRSTNAME SU

--- --

kelly ll

CHUCK UC

LAURA UR

FESTER ST

ARMANDO MA

MAJOR JO

6 rows selected.

ANALYSIS:

Remember the character field **FIRSTNAME** in this example is 15 characters long. That is why you used a **-13** to start at the third character. Counting back from 15 puts you at the start of the third character, not at the start of the second. If you don't have a third argument, use the following statement instead:

SELECT FIRSTNAME, SUBSTR(FIRSTNAME,3)

2 FROM CHARACTERS;

FIRSTNAME SUBSTR(FIRSTN

--- ---

kelly lly

CHUCK UCK

LAURA URA

FESTER STER

ARMANDO MANDO

MAJOR JOR

6 rows selected.

The rest of the target string is returned.

SELECT * FROM SSN_TABLE;

SSN_____

300541117

301457111

459789998

3 rows selected.

ANALYSIS:

Reading the results of the preceding output is difficult--Social Security numbers usually have dashes. Now try something fancy and see whether you like the results:

SELECT SUBSTR(SSN,1,3)||'-'||SUBSTR(SSN,4,2)||'-

'||SUBSTR(SSN,6,4) SSN

2 FROM SSN_TABLE;

SSN_____

300-54-1117

301-45-7111

459-78-9998

3 rows selected.

This particular use of the **substr** function could come in very handy with large numbers using commas such as 1,343,178,128 and in area codes and phone numbers such as 317-787-2915 using dashes.

Here is another good use of the **SUBSTR** function. Suppose you are writing a report and a few columns are more than 50 characters wide. You can use the **SUBSTR** function to reduce the width of the columns to a more manageable size if you know the nature of the actual data.

Consider the following two examples:

SELECT NAME, JOB, DEPARTMENT FROM JOB_TBL;

NAME_____

JOB_____DEPARTMENT_____

ALVIN SMITH

VICEPRESIDENT MARKETING

1 ROW SELECTED.

ANALYSIS:

Notice how the columns wrapped around, which makes reading the results a little too difficult. Now try this **select:**

SELECT SUBSTR(NAME, 1,15) NAME, SUBSTR(JOB,1,15) JOB,

DEPARTMENT

2 **FROM JOB_TBL;**

OUTPUT:

NAME_____JOB_____DEPARTMENT____

ALVIN SMITH VICEPRESIDENT MARKETING

Much better!

TRANSLATE

The function **TRANSLATE** takes three arguments: the target string, the **FROM** string, and the **TO** string. Elements of the target string that occur in the **FROM** string are translated to the corresponding element in the **TO** string.

SELECT FIRSTNAME, TRANSLATE(FIRSTNAME

2 **'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ**

3 **'NNNNNNNNNAAAAAAAAAAAAAAAAAAAAAAA)**

4 **FROM CHARACTERS;**

FIRSTNAME TRANSLATE(FIRST

---

kelly kelly

CHUCK AAAAA

LAURA AAAAA

FESTER AAAAAA

ARMANDO AAAAAAA

MAJOR AAAAA

6 rows selected.

Notice that the function is case sensitive.

INSTR

To find out where in a string a particular pattern occurs, use **INSTR**. Its first argument is the target string. The second argument is the pattern to match. The third and forth are numbers representing where to start looking and which match to report. This example returns a number representing the first occurrence of **O** starting with the second charac

```
SELECT LASTNAME, INSTR(LASTNAME, 'O', 2, 1)
2 FROM CHARACTERS;
LASTNAME INSTR(LASTNAME,'O',2,1)
---------- ----------
PURVIS 0
TAYLOR 5
CHRISTINE 0
ADAMS 0
COSTALES 2
KONG 2
6 rows selected.
```

ANALYSIS:

The default for the third and fourth arguments is **1**. If the third argument is negative, the search starts at a position determined from the end of the strin instead of from the beginning.

LENGTH

LENGTH returns the length of its lone character argument. For example:

```
SELECT FIRSTNAME, LENGTH(RTRIM(FIRSTNAME))
2 FROM CHARACTERS;

FIRSTNAME LENGTH(RTRIM(FIRSTNAME))
--------------- ------------------------

kelly 5

CHUCK 5

LAURA 5

FESTER 6

ARMANDO 7

MAJOR 5

6 rows selected.
```

ANALYSIS:

Note the use of the **RTRIM** function. Otherwise, **LENGTH** would return **15** for every value.

Conversion Functions

These three conversion functions provide a handy way of converting one type of data to another. These examples use the table **CONVERSIONS**.

```
SELECT * FROM CONVERSIONS;

NAME TESTNUM

--------------- ---------

40 95

13 23-

74 68
```

The **NAME** column is a character string 15 characters wide, and **TESTNUM** is a number.

TO_CHAR

The primary use of **TO_CHAR** is to convert a number into a character. Differen implementations may also use it to convert other data types, like Date, into character, or to include different formatting arguments. The next example illustrates the primary use of **TO_CHAR**:

```
SELECT TESTNUM, TO_CHAR(TESTNUM)
2 FROM CONVERT;
TESTNUM TO_CHAR(TESTNUM)
--------- ---------------
95 95
23 23
68 68
```

Not very exciting, or convincing. Here's how to verify that the function returned a character string:

```
SELECT TESTNUM, LENGTH(TO_CHAR(TESTNUM))
2 FROM CONVERT;
TESTNUM LENGTH(TO_CHAR(TESTNUM))
--------- ------------------------
95 2
23 2
68 2
```

ANALYSIS:

LENGTH of a number would have returned an error. Notice the difference between **TO CHAR** and the **CHR** function discussed earlier. **CHR** would have turned this number into a character or a symbol, depending on the character set.

TO_NUMBER

TO_NUMBER is the companion function to **TO_CHAR**, and of course, it converts a string into a number. For example:

```
SELECT NAME, TESTNUM, TESTNUM*TO_NUMBER(NAME)
2 FROM CONVERT;
```

NAME TESTNUM TESTNUM*TO_NUMBER(NAME)

---

40 95 3800

13 23 299

74 68 5032

ANALYSIS:

This test would have returned an error if **TO_NUMBER** had returned a character.

Miscellaneous Functions

Here are three miscellaneous functions you may find useful.

GREATEST and LEAST

These functions find the **GREATEST** or the **LEAST** member from a series of expressions. For example:

INPUT:

SQL> **SELECT GREATEST('ALPHA', 'BRAVO','FOXTROT', 'DELTA')**

2 **FROM CONVERT;**

OUTPUT:

GREATEST

FOXTROT

FOXTROT

FOXTROT

ANALYSIS:

Notice **GREATEST** found the word closest to the end of the alphabet. Notice also a seemingly unnecessary **FROM** and three occurrences of **FOXTROT**. If **FROM** is missing, you will get an error. Every **SELECT** needs a **FROM**. The particular table used in the **FROM** has three rows, so the function in the **SELECT** clause is performed for each of them.

SELECT LEAST(34, 567, 3, 45, 1090)

2 **FROM CONVERT;**

LEAST(34,567,3,45,1090)

3

3

3

As you can see, **GREATEST** and **LEAST** also work with numbers.

USER

USER returns the character name of the current user of the database.

```
SELECT USER FROM CONVERT;

USER
----------

PERKINS

PERKINS

PERKINS
```

There really is only one of me. Again, the echo occurs because of the number of rows in t table. **USER** is similar to the date functions explained earlier today. Even though **USER** is not an actual column in the table, it is selected for each row that is contained in the table

SUMMARY:

SQL Server user-defined functions are routines that accept parameters, perform an act such as a complex calculation, and return the result of that action as a value. The ret value can either be a single scalar value or a result set.

Functions are available, including facilities to count non-missing values; determine minimum and maximum values in specific columns; return the range of values; comp the mean, standard deviation, Returns the average of the values in a group.

Null values are ignored, and variance of specific values; and other aggregating functi The following table presents an alphabetical listing of the available PROC SQL summ functions; when multiple names for the same function are available, the ANSI-appro name appears first.

Scalar functions return a single value based on the input value.

REVIEW QUESTIONS:

1. What are the specific uses of SQL functions?
2. What is the purpose of the NVL function?
3. Which function returns the remainder in a division operation?
4. What are aggregate and scalar functions?
5. What are the Numeric functions ?name them

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #7

Course: Database Systems

Instructor: NAZISH ALI

Semester: Spring 2018

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.

The SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

The percent sign and the underscore can also be used in combinations!

LIKE Syntax

	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

You can also combine any number of conditions using AND or OR operators.

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

SQL LIKE Examples

The following SQL statement selects all customers with a CustomerName starting with "a":

Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a%';
```

The following SQL statement selects all customers with a CustomerName ending with "a":

Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE '%a';
```

The following SQL statement selects all customers with a CustomerName that have "or" in any position:

Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE '%or%';
```

The following SQL statement selects all customers with a CustomerName that have "r" in the second position:

Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE '_r%';
```

The following SQL statement selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a_%_%';
```

The following SQL statement selects all customers with a ContactName that starts with "a" and ends with "o":

Example

```
SELECT * FROM Customers
WHERE ContactName LIKE 'a%o';
```

The following SQL statement selects all customers with a CustomerName that does NOT start with "a":

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

Example

```
SELECT * FROM Customers
WHERE CustomerName NOT LIKE 'a%';
```

SQL Wildcards

SQL Wildcard Characters

A wildcard character is used to substitute any other character(s) in a string.

Wildcard characters are used with the SQL LIKE operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

In MS Access and SQL Server you can also use:

- [*charlist*] - Defines sets and ranges of characters to match
- [^*charlist*] or [!*charlist*] - Defines sets and ranges of characters NOT to match

The wildcards can also be used in combinations!

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

Below is a selection from the "Customers" table in the Northwind sample database.

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Using the % Wildcard

The following SQL statement selects all customers with a City starting with "ber":

Example

```
SELECT * FROM Customers
WHERE City LIKE 'ber%';
```

The following SQL statement selects all customers with a City containing the pattern "es":

Example

```
SELECT * FROM Customers
WHERE City LIKE '%es%';
```

Using the _ Wildcard

The following SQL statement selects all customers with a City starting with any character, followed by "erlin":

Example

```
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

The following SQL statement selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

Example

```
SELECT * FROM Customers
WHERE City LIKE 'L_n_on';
```

Using the [charlist] Wildcard

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

Example

```
SELECT * FROM Customers
WHERE City LIKE '[bsp]%';
```

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

Example

```
SELECT * FROM Customers
WHERE City LIKE '[a-c]%'
```

Using the [!charlist] Wildcard

The two follow-ing SQL statements selects all customers with a City NOT starting with "b", "s", or "p":

Example

```
SELECT * FROM Customers
WHERE City LIKE '[!bsp]%';
```

Or:

Example

```
SELECT * FROM Customers
WHERE City NOT LIKE '[bsp]%';
```

The SQL IN 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*)

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

IN Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" and "UK":

Example

SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

Example

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

The following* SQL statement selects all customers that are from the same countries as suppliers:

Example

```
SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers);
```

The SQL BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

BETWEEN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Below is a selection from the "Products" table in the Northwind sample database:

ProductID	ProductName	SupplierID	CategoryID	Unit
1	Chais	1	1	10 boxes x 20 bags
2	Chang	1	1	24 - 12 oz bottles
3	Aniseed Syrup	1	2	12 - 550 ml bottles
4	Chef Anton's Cajun Seasoning	1	2	48 - 6 oz jars
5	Chef Anton's Gumbo Mix	1	2	36 boxes

BETWEEN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20:

Example

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

NOT BETWEEN Example

To display the products outside the range of the previous example, use NOT BETWEEN:

Example

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

BETWEEN with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

Example

```
SELECT * FROM Products
WHERE (Price BETWEEN 10 AND 20)
AND NOT CategoryID IN (1,2,3);
```

BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

Example

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

NOT BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName NOT BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

Example

```
SELECT * FROM Products
WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

Sample Table

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	7/4/1996	3
10249	81	6	7/5/1996	1
10250	34	4	7/8/1996	2
10251	84	3	7/9/1996	1
10252	76	4	7/10/1996	2

BETWEEN Dates Example

The following SQL statement selects all orders with an OrderDate BETWEEN '04-July-199 and '09-July-1996':

Example

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;
```

SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.An alias only exists for the duration of the query.

Alias Column Syntax

```
SELECT column_name AS alias_name
FROM table_name;
```

Alias Table Syntax

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico

3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10354	58	8	1996-11-14	3
10355	4	6	1996-11-15	1
10356	86	6	1996-11-18	2

Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

Example

```
SELECT CustomerID as ID, CustomerName AS Customer
FROM Customers;
```

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. It requires double quotation marks or square brackets if the alias name contains spaces:

Example

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
FROM Customers;
```

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

Example

```
SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address
FROM Customers;
```

To get the SQL statement above to work in MySQL use the following:

```
SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address
FROM Customers;
```

Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=
(Around the Horn). We use the "Customers" and "Orders" tables, and give them the table
aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

Example

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;
```

The following SQL statement is the same as above, but without aliases:

Example

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName
FROM Customers, Orders
WHERE Customers.CustomerName="Around the
Horn" AND Customers.CustomerID=Orders.CustomerID;
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable
- Two or more columns are combined together

The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact on performance.

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

SQL Server / MS Access Syntax:

SELECT TOP *number|percent column_name(s)*
FROM *table_name*
WHERE *condition*;

Below is a selection from the "Customers" table in the Northwind sample database:

SQL TOP, LIMIT and ROWNUM Examples

The following SQL statement selects the first three records from the "Customers" table:

Example

```
SELECT TOP 3 * FROM Customers;
```

The following SQL statement shows the equivalent example using the LIMIT clause:

Example

```
SELECT * FROM Customers
LIMIT 3;
```

The following SQL statement shows the equivalent example using ROWNUM:

Example

```
SELECT * FROM Customers
WHERE ROWNUM <= 3;
```

SQL TOP PERCENT Example

The following SQL statement selects the first 50% of the records from the "Customers" table:

Example

```
SELECT TOP 50 PERCENT * FROM Customers;
```

ADD a WHERE CLAUSE

The following SQL statement selects the first three records from the "Customers" table, where the country is "Germany":

Example

```
SELECT TOP 3 * FROM Customers
WHERE Country='Germany';
```

The following SQL statement shows the equivalent example using the LIMIT clause:

Example

```
SELECT * FROM Customers
WHERE Country='Germany'
LIMIT 3;
```

The following SQL statement shows the equivalent example using ROWNUM:

Example

```
SELECT * FROM Customers
WHERE Country='Germany' AND ROWNUM <= 3;
```

SUMMARY:

- Two wildcard characters, the percent sign (%) and the underscore (_), can be used.
- The percent sign is analogous to the asterisk (*) wildcard character used with MS-DOS. The percent sign allows for the substitution of one or more characters in a field.
- *Wildcard* characters allowed in 'value' are % (percent) and _ (underscore).

- The LIKE operator allows you to search for a string of text based on a specified pattern
- You can use the LIKE operator in the WHERE clause of any valid SQL statement such as SELECT, UPDATE or DELETE.
- There are other comparison keywords available in sql which are used to enhance the search capabilities of a sql query. They are "IN", "BETWEEN...AND", "IS NULL", "LIKE".
- The operator BETWEEN and AND, are used to compare data for a range of values.
- A column value is NULL if it does not exist. The IS NULL operator is used to display all the rows for columns that do not have a value.
- The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.
- WHERE conditions can be combined with AND, OR, and NOT.
- A WHERE clause with AND requires that two conditions are true.
- A WHERE clause with OR requires that one of two conditions is true.
- A WHERE clause with NOT negates the specified condition.
- The SELECT TOP clause is used to specify the number of records to return
- SQL Aliases are defined for columns and tables. Basically aliases is created to make the column selected more readable
- Aliases reduce the amount of typing required to enter a query.
- Aliases are useful with JOINs and aggregates: SUM, COUNT, etc.

REVIEW QUESTIONS:

1. Can you sort a column using a column alias?
2. What are the various multiple row comparison operators in SQL?
3. What is the purpose of the condition operators BETWEEN and IN?
4. What is the purpose of TOP?
5. What do you mean by like operator?

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #8

Course: Database Systems

Instructor: NAZISH ALI

Semester: Spring 2018

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.

The SQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

ORDER BY Syntax

SELECT *column1, column2, ...*
FROM *table_name*
ORDER BY *column1, column2, ...* ASC|DESC;

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio	Antonio	Mataderos	Méxic	05023	Mexico

	Moreno Taquería	Moreno	2312	o D.F.		
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	Londo n	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsväg en 8	Luleå	S-958 22	Sweden

ORDER BY Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

Example

```
SELECT * FROM Customers
ORDER BY Country;
```

ORDER BY DESC Example

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

Example

```
SELECT * FROM Customers
ORDER BY Country DESC;
```

ORDER BY Several Columns Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column:

Example

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

ORDER BY Several Columns Example 2

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

Example

```
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

The SQL GROUP BY Statement-

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.

GROUP BY Syntax

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	Londo n	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsväge n 8	Luleå	S-958 22	Sweden

```
SELECT column_name(s)
FROM table_name

WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Below is a selection from the "Customers" table in the Northwind sample database:

SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

Example

```
SELECT COUNT (CustomerID), Country
FROM Customers
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

Example

```
SELECT COUNT (CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT (CustomerID) DESC;
```

Below is a selection from the "Orders" table in the Northwind sample database:

And a selection from the "Shippers" table:

ShipperID	ShipperName
1	Speedy Express
2	United Package
3	Federal Shipping

The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not 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);*

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo	Notes
1	Davolio	Nancy	1968-12-08	EmpID1.pic	Education includes a BA....
2	Fuller	Andrew	1952-02-19	EmpID2.pic	Andrew received his BTS....
3	Leverling	Janet	1963-08-30	EmpID3.pic	Janet has a BS degree

Transaction Control

Transactional control commands **COMMIT**, **ROLLBACK**, and **SAVEPOINT**. These commands allow the programmer to control when transactions are actually written to the database, how often, and when they should be undone.

SYNTAX:

```
BEGIN
DECLARE
...
BEGIN
statements...
IF condition THEN
COMMIT;
ELSE
ROLLBACK;
END IF;
...
EXCEPTION
...
END;
```

END;

The good thing about PL/SQL is that you can automate the use of transactional control commands instead of constantly monitoring large transactions, which can be very tedious.

Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The following commands are used to control transactions.

- **COMMIT** – to save the changes.
- **ROLLBACK** – to roll back the changes.
- **SAVEPOINT** – creates points within the groups of transactions in which to ROLLBACK.
- **SET TRANSACTION** – Places a name on a transaction.

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

```
COMMIT;
```

Example

Consider the CUSTOMERS table having the following records –

```
| 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 |
| 7 | Muffy    |  24 | Indore   | 10000.00 |
+----+----------+-----+----------+----------+
```

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
  WHERE AGE = 25;
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
```

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows –

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
| 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
7	Muffy	24	Indore	10000.00

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
  WHERE AGE = 25;
SQL> ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

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
7	Muffy	24	Indore	10000.00

The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can

ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| 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 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone −

```
SQL> ROLLBACK TO SP2;

Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2.

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  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 |
+----+----------+-----+---------+----------+
```

6 rows selected.

The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

The syntax for a SET TRANSACTION command is as follows.

```
SET TRANSACTION [READ WRITE | READ ONLY];
```

SUMMARY:

- Having clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in Group based SQL functions, just like WHERE clause.
- Having clause is used to filter data based on the group functions.

- The SQL GROUP BY Clause is used along with the group functions to retrieve data grouped according to one or more columns.
- Order by clause is used with **Select** statement for arranging retrieved data in sorted order.
- Transaction Control Language (TCL) commands are used to manage transactions in database. It also allows statements to be grouped together into logical transactions

REVIEW QUESTIONS:

1. What is the default ordering of data using the ORDER BY clause? How could it be changed?
2. Group functions can be or cannot be nested?
3. Write the syntax of Group By
4. What is TCL? Explain it.
5. What is the difference between Group BY and Order By?

University Of Karachi
Department Of Computer Science
DATABASE LAB
HANDOUT #10

Course: Database Systems

Instructor: NAZISH ALI **Semester:** Spring 2018

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.

Objectives

Today you will learn about joins. This information will enable you to gather and manipulate data across several tables. By the end of the day, you will understand and be able to do the following:

- l Perform an outer join
- l Perform a left join
- l Perform a right join
- l Perform an equi-join
- l Perform a non-equi-join
- l Join a table to itself

SQL: JOINS

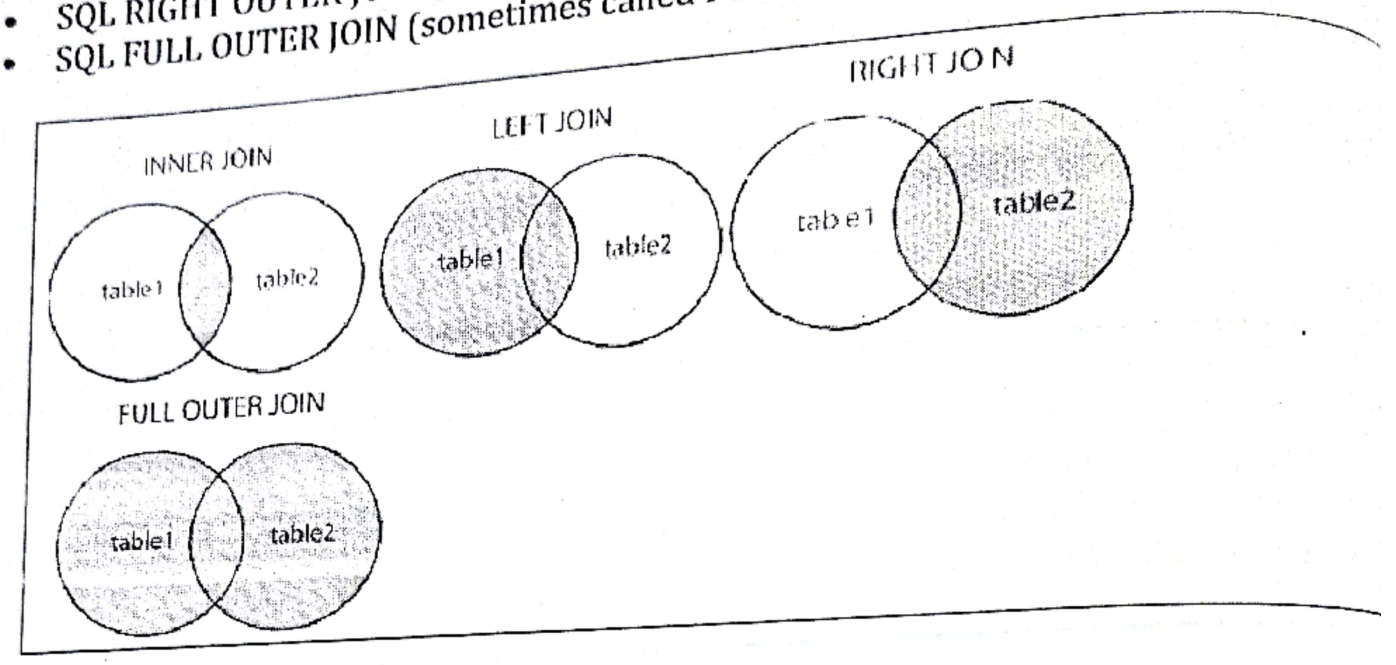
This SQL handout explains how to use SQL **JOINS** with syntax, visual illustrations, and examples

Description

SQL JOINS are used to retrieve data from multiple tables. A SQL JOIN is performed whenever two or more tables are listed in a SQL statement.

There are 4 different types of SQL joins:

- SQL INNER JOIN (sometimes called simple join)
- SQL LEFT OUTER JOIN (sometimes called LEFT JOIN)
- SQL RIGHT OUTER JOIN (sometimes called RIGHT JOIN)
- SQL FULL OUTER JOIN (sometimes called FULL JOIN)



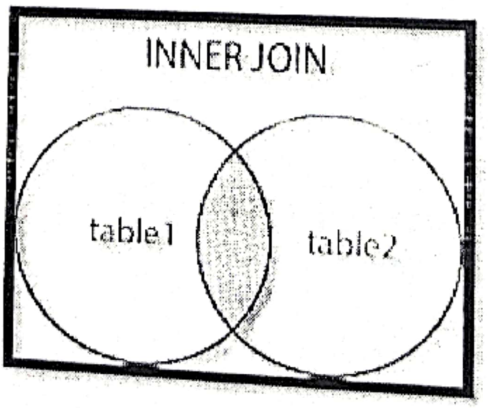
So let's discuss SQL JOIN syntax, look at visual illustrations of SQL JOINS and explore some examples.

SQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

INNER JOIN Syntax

SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2* ON *table1.column_name* = *table2.column_name*;



-Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

And a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

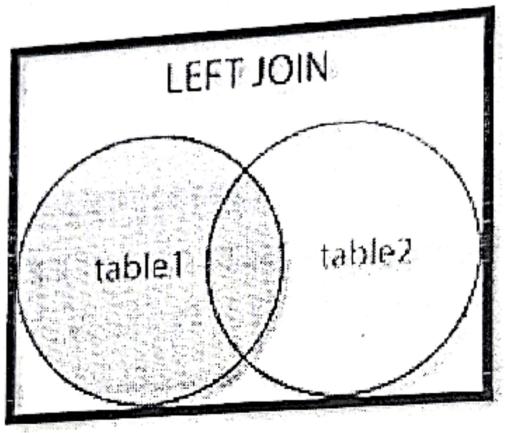
SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

In some databases LEFT JOIN is called LEFT OUTER JOIN.



Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

Example

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

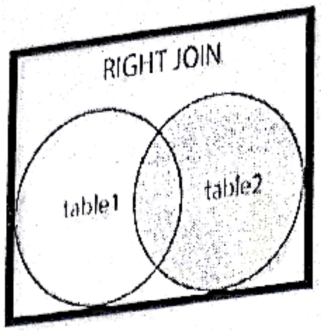
SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

RIGHT JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmpID1.pic
2	Fuller	Andrew	2/19/1952	EmpID2.pic
3	Leverling	Janet	8/30/1963	EmpID3.pic

SQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have have placed:

Example

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

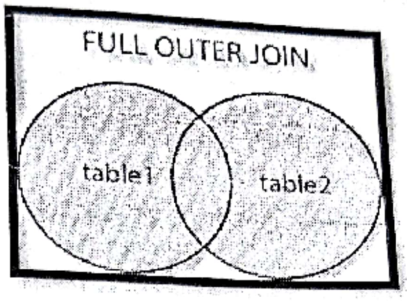
SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword return all records when there is a match in either left (table1) or right (table2) table records.

FULL OUTER JOIN can potentially return very large result-sets!

FULL OUTER JOIN Syntax

SELECT *column_name(s)*
FROM *table1*
FULL OUTER JOIN *table2* ON *table1.column_name = table2.column_name;*



Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Countr
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germa
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexic
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexi

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

A selection from the result set may look like this:

CustomerName	OrderID
Alfreds Futterkiste	10308
Ana Trujillo Emparedados y helados	10365
Antonio Moreno Taquería	

The FULL OUTER JOIN keyword returns all the rows from the left table (Customers), and all the rows from the right table (Orders). If there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

SQL Self JOIN

A self-JOIN is a regular join, but the table is joined with itself.

Self-JOIN Syntax

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

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Count
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germ
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexic
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexic

The variable **id_num** is declared to have the same data type as **emp_id** in the **EMPLOYEES** table. **%TYPE** declares the variable **name** to have the same data type as the column

emp_name in the **EMPLOYEES** table.

The %ROWTYPE Attribute

Variables are not limited to single values. If you declare a variable that is associated with a defined cursor, you can use the **%ROWTYPE** attribute to declare the data type of that variable to be the same as each column in one entire row of data from the cursor.

In Oracle's lexicon the **%ROWTYPE** attribute creates a record variable.

INPUT:

```
DECLARE
cursor employee_cursor is
select emp_id, emp_name from employees;
employee_record employee_cursor%ROWTYPE;
```

ANALYSIS:

This example declares a variable called **employee_record.** The **%ROWTYPE** attribute defines this variable as having the same data type as an entire row of data in the **employee_cursor.** Variables declared using the **%ROWTYPE** attribute are also called aggregate variables.

The %ROWCOUNT Attribute

The PL/SQL **%ROWCOUNT** attribute maintains a count of rows that the SQL statements in the particular block have accessed in a cursor.

INPUT:

```
DECLARE
cursor employee_cursor is
select emp_id, emp_name from employees;
records_processed := employee_cursor%ROWCOUNT;
```

ANALYSIS:

In this example the variable **records_processed** represents the current number of rows that the PL/SQL block has accessed in the **employee_cursor.**

WARNING: Beware of naming conflicts with table names when declaring variables. For instance, if you declare a variable that has the same name as a table that you are trying to access with the PL/SQL code, the local variable will take precedence over the table name.

The PROCEDURE Section

The **PROCEDURE** section is the only mandatory part of a PL/SQL block. This part of the block calls variables and uses cursors to manipulate data in the database. The **PROCEDURE** section is the main part of a block, containing conditional statements and

SQL commands.
BEGIN...END

In a block, the **BEGIN** statement denotes the beginning of a procedure. Similarly, the **END** statement marks the end of a procedure. The following example shows the basic structure of the **PROCEDURE** section:

SYNTAX:

```
BEGIN
open a cursor;
condition1;
statement1;
condition2;
statement2;
condition3;
statement3;
.
.
.
close the cursor;
END
```

Cursor Control Commands

Now that you have learned how to define cursors in a PL/SQL block, you need to know how to access the defined cursors. This section explains the basic cursor control commands: **DECLARE**, **OPEN**, **FETCH**, and **CLOSE**.

DECLARE

Earlier today you learned how to define a cursor in the **DECLARE** section of a block. The **DECLARE** statement belongs in the list of cursor control commands.

OPEN

Now that you have defined your cursor, how do you use it? You cannot use this book unless you open it. Likewise, you cannot use a cursor until you have opened it with the

OPEN command. For example:

SYNTAX:

```
BEGIN
open employee_cursor;
statement1;
statement2;
.
.
.
END
```

FETCH

FETCH populates a variable with values from a cursor. Here are two examples using **FETCH**: One populates an aggregate variable, and the other populates individual variables.

INPUT:

```
DECLARE
cursor employee_cursor is
select emp_id, emp_name from employees;
employee_record employee_cursor%ROWTYPE;
BEGIN
open employee_cursor;
loop
fetch employee_cursor into employee_record;
```

```
end loop;
close employee_cursor;
END
```

ANALYSIS:

The preceding example fetches the current row of the cursor into the aggregate variable **employee_record**. It uses a loop to scroll the cursor. Of course, the block is not actually accomplishing anything.

```
DECLARE
cursor employee_cursor is
select emp_id, emp_name from employees;
id_num employees.emp_id%TYPE;
name employees.emp_name%TYPE;
BEGIN
open employee_cursor;
loop
fetch employee_cursor into id_num, name;
end loop;
close employee_cursor;
END
```

ANALYSIS:

This example fetches the current row of the cursor into the variables **id_num** and **name**,which was defined in the **DECLARE** section.

CLOSE

When you have finished using a cursor in a block, you should close the cursor, as you normally close a book when you have finished reading it. The command you use is **CLOSE**.

SYNTAX:

```
BEGIN
open employee_cursor;
statement1;
statement2;
.
.
.
close employee_cursor;
END
```

ANALYSIS:
After a cursor is closed, the result set of the query no longer exists. You must reopen the cursor to access the associated set of data.

Conditional Statements

Now we are getting to the good stuff--the conditional statements that give you control over how your SQL statements are processed. The conditional statements in **PL/SQL resemble those in most third-generation languages.**

IF...THEN

The **IF...THEN** statement is probably the most familiar conditional statement to most programmers. The **IF...THEN** statement dictates the performance of certain actions if certain conditions are met. The structure of an **IF...THEN** statement is as follows:

SYNTAX:

```
IF condition1 THEN
statement1;
END IF;
```

If you are checking for two conditions, you can write your statement as follows:

SYNTAX:

```
IF condition1 THEN
statement1;
ELSE-
statement2;
END IF;
```

If you are checking for more than two conditions, you can write your statement as follows:

SYNTAX:

```
IF condition1 THEN
statement1;
ELSIF condition2 THEN
statement2;
ELSE
statement3;
END IF;
```

ANALYSIS:

The final example states: If **condition1** is met, then perform **statement1**; if **condition2** is met, then perform **statement2**; otherwise, perform **statement3**. **IF...THEN** statements may also be nested within other statements and/or loops.

LOOPS

Loops in a PL/SQL block allow statements in the block to be processed continuously for as long as the specified condition exists. There are three types of loops.

LOOP is an infinite loop, most often used to scroll a cursor. To terminate this type of loop, you must specify when to exit. For example, in scrolling a cursor you would exit the loop after the last row in a cursor has been processed:

INPUT:

```
BEGIN
open employee_cursor;
LOOP
FETCH employee_cursor into employee_record;
EXIT WHEN employee_cursor%NOTFOUND;
statement1;
.
.
.
END LOOP;
close employee_cursor;
END;
```

%NOTFOUND is a cursor attribute that identifies when no more data is found in the cursor.

The preceding example exits the loop when no more data is found. If you omit this statement from the loop, then the loop will continue forever.

The **WHILE-LOOP** executes commands while a specified condition is **TRUE**. When the condition is no longer true, the loop returns control to the next statement.

INPUT:

```
DECLARE
cursor payment_cursor is
select cust_id, payment, total_due from payment_table;
cust_id payment_table.cust_id%TYPE;
payment payment_table.payment%TYPE;
total_due payment_table.total_due%TYPE;
BEGIN
open payment_cursor;
WHILE payment < total_due LOOP
FETCH payment_cursor into cust_id, payment, total_due;
EXIT WHEN payment_cursor%NOTFOUND;
insert into underpay_table
values (cust_id, 'STILL OWES');
END LOOP;
close payment_cursor;
END;
```

ANALYSIS:

The preceding example uses the **WHILE-LOOP** to scroll the cursor and to execute the commands within the loop as long as the condition **payment < total_due>** is met. You can use the **FOR-LOOP** in the previous block to implicitly fetch the current row of the cursor into the defined variables.

```
INPUT:
DECLARE
cursor payment_cursor is
select cust_id, payment, total_due from payment_table;
cust_id payment_table.cust_id%TYPE;
payment payment_table.payment%TYPE;
total_due payment_table.total_due%TYPE;
BEGIN
open payment_cursor;
FOR pay_rec IN payment_cursor LOOP
IF pay_rec.payment < pay_rec.total_due THEN
insert into underpay_table
values (pay_rec.cust_id, 'STILL OWES');
END IF;
END LOOP;
close payment_cursor;
END;
```

ANALYSIS:

This example uses the **FOR-LOOP** to scroll the cursor. The **FOR-LOOP** is performing an implicit **FETCH**, which is omitted this time. Also, notice that the **%NOTFOUND** attribute has been omitted. This attribute is implied with the **FOR-LOOP**; therefore, this and the previous example yield the same basic results.

The EXCEPTION Section

The **EXCEPTION** section is an optional part of any PL/SQL block. If this section is omitted and errors are encountered, the block will be terminated. Some errors that are encountered may not justify the immediate termination of a block, so the **EXCEPTION** section can be used to handle specified errors or user-defined exceptions in an orderly manner. Exceptions can be user-defined, although many exceptions are predefined by
Oracle.

Raising Exceptions

Exceptions are raised in a block by using the command **RAISE**.
Exceptions can be raised explicitly by the programmer, whereas internal database errors are automatically, or implicitly, raised by the database server.

SYNTAX:

```
BEGIN
DECLARE
exception_name EXCEPTION;
BEGIN
IF condition THEN
RAISE exception_name;
END IF;
EXCEPTION
WHEN exception_name THEN
statement;
END;
END;
```

ANALYSIS:

This block shows the fundamentals of explicitly raising an exception. First **exception_name** is declared using the **EXCEPTION** statement. In the **PROCEDURE** section, the exception is raised using **RAISE** if a given condition is met. The **RAISE** then references the **EXCEPTION** section of the block, where the appropriate action is taken.

Handling Exceptions

The preceding example handled an exception in the **EXCEPTION** section of the block. Errors are easily handled in PL/SQL, and by using exceptions, the PL/SQL block can continue to run with errors or terminate gracefully.

SYNTAX:

```
EXCEPTION
WHEN exception1 THEN
statement1;
WHEN exception2 THEN
statement2;
WHEN OTHERS THEN
statement3;
```

ANALYSIS:

This example shows how the **EXCEPTION** section might look if you have more than one exception. This example expects two exceptions (**exception1** and **exception2**) when running this block. **WHEN OTHERS** tells **statement3** to execute if any other exceptions occur while the block is being processed. **WHEN OTHERS** gives you control over any errors that may occur within the block.

Executing a PL/SQL Block

PL/SQL statements are normally created using a host editor and are executed like normal SQL script files. PL/SQL uses semicolons to terminate each statement in a block--from variable assignments to data manipulation commands. The forward slash (/)is mainly associated with SQL script files, but PL/SQL also uses the forward slash to

terminate a block in a script file. The easiest way to start a PL/SQL block is by issuing
the **START** command, abbreviated as **STA** or **@**.
Your PL/SQL script file might look like this:

SYNTAX:

```
/* This file is called proc1.sql */
BEGIN
DECLARE
...
BEGIN
...
statements;
...
EXCEPTION
...
END;
END;
/
```

You execute your PL/SQL script file as follows:

SQL> **start proc1** or
SQL> **sta proc1** or
SQL> **@proc1**

NOTE: PL/SQL script files can be executed using the **START** command or the
character @. PL/SQL script files can also be called within other PL/SQL files, shell scripts, or other programs.

Displaying Output to the User

Particularly when handling exceptions, you may want to display output to keep users
informed about what is taking place. You can display output to convey information, and you can display your own customized error messages, which will probably make more sense to the user than an error number. Perhaps you want the user to contact the database administrator if an error occurs during processing, rather than to see the exact message.

PL/SQL does not provide a direct method for displaying output as a part of its syntax, but it does allow you to call a package that serves this function from within the block.

The package is called **DBMS_OUTPUT.**

```
EXCEPTION
WHEN zero_divide THEN
DBMS_OUTPUT.put_line('ERROR: DIVISOR IS ZERO. SEE YOUR DBA.');
```

ANALYSIS:

ZERO_DIVIDE is an Oracle predefined exception. Most of the common errors that occur during program processing will be predefined as exceptions and are raised implicitly (which means that you don't have to raise the error in the PROCEDURE section of the block).If this exception is encountered during block processing, the user will see:

INPUT:

```
SQL> @block1
ERROR: DIVISOR IS ZERO. SEE YOUR DBA.
PL/SQL procedure successfully completed.
```

Doesn't that message look friendly than:

INPUT/OUTPUT:

```
SQL> @block1
begin
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 20
```

SUMMARY:

PL/SQL includes procedural language elements such as conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variables of those types, and triggers.

It can handle exceptions (runtime errors). Arrays are supported involving the use of PL/SQL collections. Implementations from version 8 of Oracle Database onwards have included features associated with object-orientation.

One can create PL/SQL units such as procedures, functions, packages, types, and triggers, which are stored in the database for reuse by applications that use any of the Oracle Database programmatic interfaces.

PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.

PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program.

University Of Karachi
Department Of Computer Science
DATABASE LAB
HANDOUT #11

Course: Database Systems

Instructor: NAZISH ALI

Semester: Spring 2018

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.

Introduction

A procedure (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database. There are many useful applications of SQL procedures within a database or database application architecture

Defining an SQL procedure

A **stored procedure** is a set of Structured Query Language (SQL) statements with an assigned name, which are **stored** in a relational database management system as a group, so it can be reused and shared by multiple programs.

The CREATE PROCEDURE statement for SQL procedures:

- Names the procedure
- Creates the stored procedure
- Defines the parameters and their attributes
- Provides other information about the procedure which will be used when the procedure is called

- Defines the procedure body .

Here is the complete syntax of CREATE PROCEDURE (the syntax is based on SQL:2003 standard).

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
 < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters.
- IN : Identifies the parameter as an input parameter to the procedure.
- OUT : Identifies the parameter as an output parameter that is returned by the procedure.
- INOUT : Identifies the parameter as both an input and output parameter for the procedure.
- datatype : Specifies the data type of the parameter(s).
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalo procedure.

Example: SQL Procedure

Here is a simple example that takes as input student registration number, total marks an number of subjects and updates the percentage of marks:

SQL Code:

```
CREATE PROCEDURE STUDENT_MARKS
(IN STUDENT_REG_NO CHAR (15), IN TOTAL_MARKS DECIMAL (7, 2), NO_SUBJECTS INT
(3))
LANGUAGE SQL MODIFIES SQL DATA
UPDATE STUDENTMAST.MARKS
SET PERCENTAGE = TOTAL_MARKS/NO_SUBJECT
WHERE REG_NO = STUDENT_REG_NO
```

Explanation:

- Names of the procedure are STUDENT_MARKS
- Defines parameter STUDENT_REG_NO (character data type of length 15), TOTAL_MARKS (decimal data type) and NO_SUBJECTS (integer type) which all are input parameters.
- Indicates the procedure is an SQL procedure that modifies SQL data.
- Defines the procedure body as a single UPDATE statement. When the procedure is called, the UPDATE statement is executed using the values passed for STUDENT_REG_NO, TOTAL_MARKS, and NO_SUBJECTS
- Defines the procedure body

Call a procedure

The CALL statement is used to invoke a procedure that is stored in a DATABASE.

Syntax:

ALL sp_name([parameter[,...]])

ALL sp_name[()]

_name : Name of the procedure.

arameter, ... : List of parameters enclosed in parentheses and separated by commas.

lter a procedure

llowing command alter an existing procedure:

SQL Code:

```
ALTER PROCEDURE <procedureName>

--Add the parameters for the stored procedure here

<@Parameter1><Datatype_For_Parameter1>=<Default_Value>,
<@Parameter1><Datatype_For_Parameter1>=<Default_Value>
AS
BEGIN
   Insert statements for Procedure here

   SELECT *  FROM TableName WHERE <conditions>
END
GO
```

Drop a procedure

DROP PROCEDURE proc_name

proc_name: Name of the procedure

What is a Trigger

A trigger is a special kind of a store procedure that executes in response to certain action on the table like insertion, deletion or updation of data. It is a database object which is bound to a table and is executed automatically. You can't explicitly invoke triggers. The only way to do this is by performing the required action on the table that they are assigned to.

Types Of Triggers

There are three action query types that you use in SQL which are INSERT, UPDATE and DELETE. So, there are three types of triggers and hybrids that come from mixing and matching the events and timings that fire them. Basically, triggers are classified into two main types:

After Triggers (For Triggers)
Instead Of Triggers

After Triggers

These triggers run after an insert, update or delete on a table. They are **not supported for views.**
AFTER TRIGGERS can be classified further into three types as:

a. AFTER INSERT Trigger.
b. AFTER UPDATE Trigger.
c. AFTER DELETE Trigger

(ii) Instead Of Triggers

These can be used as an interceptor for anything that anyone tried to do on our table or view. If you define an *Instead Of trigger* on a table for the Delete operation, they try to delete rows, and they will not actually get deleted (unless you issue another delete instruction from within the trigger)

INSTEAD OF TRIGGERS can be classified further into three types as:

a. INSTEAD OF INSERT Trigger.
b. INSTEAD OF UPDATE Trigger.
c. INSTEAD OF DELETE Trig

Syntax

The following is the very easy and useful syntax of triggers:

```
CREATE TRIGGER triggerName ON table

AFTER INSERT |After Delete |After Upadte

AS BEGIN

INSERT INTO dbo.UserHistory.....

END
```

```
-- First create table Employee

CREATE TABLE Employee_Test

(

Emp_ID INT Identity,

Emp_name Varchar(100),

Emp_Sal Decimal (10,2)

)
```

-- Now insert records

```
CREATE TABLE Employee_Test
(
Emp_ID INT Identity,
Emp_name Varchar(100),
Emp_Sal Decimal (10,2)
)
INSERT INTO Employee_Test VALUES ('Anees',1000);
INSERT INTO Employee_Test VALUES ('Rick',1200);
INSERT INTO Employee_Test VALUES ('John',1100);
INSERT INTO Employee_Test VALUES ('Stephen',1300);
INSERT INTO Employee_Test VALUES ('Maria',1400);
```

Analysis:

I will be creating an AFTER INSERT TRIGGER which will insert the rows inserted into the table into another audit table.

The main purpose of this audit table is to record the changes in the main table.

This can be thought of as a generic audit trigger.

Now, create the audit table as:-

```
CREATE TABLE Employee_Test_Audit
(
Emp_ID int,
Emp_name varchar(100),
Emp_Sal decimal (10,2),
Audit_Action varchar(100),
Audit_Timestamp datetime)
```

(a) After Insert Trigger

This trigger is fired after an INSERT on the table. Let's create the trigger as:

```
CREATE TRIGGER trgAfterInsert ON [dbo].[Employee_Test]
FOR INSERT
AS
        declare @empid int;
        declare @empname varchar(100);
        declare @empsal decimal(10,2);
        declare @audit_action varchar(100);

        select @empid=i.Emp_ID from inserted i;
        select @empname=i.Emp_Name from inserted i;
        select @empsal=i.Emp_Sal from inserted i;
        set @audit_action='Inserted Record -- After Insert Trigger.';

        insert into Employee_Test_Audit
       (Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
        values(@empid,@empname,@empsal,@audit_action,getdate());

        PRINT 'AFTER INSERT trigger fired.'
GO
```

The CREATE TRIGGER statement is used to create the trigger. THE ON clause specifies the table name on which the trigger is to be attached. The FOR INSERT specifies that this is an AFTER INSERT trigger. In place of FOR INSERT, AFTER INSERT can be used. Both of them mean the same.

In the trigger body, table named **inserted** has been used. This table is a logical table and contains the row that has been inserted. I have selected the fields from the logical inserted table from the row that has been inserted into different variables, and finally inserted those values into the Audit table.

To see the newly created trigger in action, let's insert a row into the main table as:

```
Insert into Employee_Test values ('Chris', 1500);
```

Now, a record has been inserted into the Employee_Test table. The AFTER INSERT trigger attached to this table has inserted the record into the Employee_Test_Audit as:

```
6  Chris 1500.00  Inserted Record -- After Insert Trigger.      2008-04-26 12:00:55.700
```

(b) AFTER UPDATE Trigger

This trigger is fired after an update on the table. Let's create the trigger as:

```
CREATE TRIGGER trgAfterUpdate ON [dbo].[Employee_Test]
FOR UPDATE
AS
        declare @empid int;
        declare @empname varchar(100);
        declare @empsal decimal(10,2);
        declare @audit_action varchar(100);

        select @empid=i.Emp_ID from inserted i;
        select @empname=i.Emp_Name from inserted i;
        select @empsal=i.Emp_Sal from inserted i;

       if update(Emp_Name)
              set @audit_action='Updated Record -- After Update Trigger.';
       if update(Emp_Sal)
              set @audit_action='Updated Record -- After Update Trigger.';

        insert into
Employee_Test_Audit(Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
```

```
    Values (@empid, @empname, @empsal, @audit_action,getdate());
    PRINT 'AFTER UPDATE Trigger fired.'

GO
```

The AFTER UPDATE Trigger is created in which the updated record is inserted into the audit table. There is **no logical table updated like the logical table inserted.** We can obtain the updated value of a field from the update (column_name) function. In our trigger, we have used, if update (Emp_Name) to check if the column Emp_Name has been updated. We have similarly checked the column Emp_Sal for an update.

Let's update a record column and see what happens.

```
Update Employee_Test set Emp_Sal=1550 where Emp_ID=6
```

This inserts the row into the audit table as:

```
6 Chris 1550.00 Updated Record -- After Update Trigger.      2008-04-26
12:38:11.843
```

(c) AFTER DELETE Trigger

This trigger is fired after a delete on the table. Let's create the trigger as:

Hide Copy Code

```
CREATE TRIGGER trgAfterDelete ON [dbo].[Employee_Test]
AFTER DELETE
AS
    declare @empid int;
    declare @empname varchar(100);
    declare @empsal decimal(10,2);
    declare @audit_action varchar(100);

    select @empid=d.Emp_ID from deleted d;
    select @empname=d.Emp_Name from deleted d;
    select @empsal=d.Emp_Sal from deleted d;
```

```
set @audit_action='Deleted -- After Delete Trigger.';

        insert into Employee_Test_Audit
(Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
        values(@empid,@empname,@empsal,@audit_action,getdate());

        PRINT 'AFTER DELETE TRIGGER fired.'

GO
```

In this trigger, the deleted record's data is picked from the **logical deleted table** and inserted into the audit table. Let's fire a delete on the main table. A record has been inserted into the audit table as:

```
6 Chris  1550.00 Deleted -- After Delete Trigger. 2008-04-26 12:52:13.867
```

All the triggers can be enabled/disabled on the table using the statement

```
ALTER TABLE Employee_Test {ENABLE|DISBALE} TRIGGER ALL
```

Specific Triggers can be enabled or disabled as:

```
ALTER TABLE Employee_Test DISABLE TRIGGER trgAfterDelete
```

This disables the After Delete Trigger named trgAfterDelete on the specified table.

(ii) Instead Of Triggers

These can be used as an interceptor for anything that anyone tried to do on our table or view. If you define an *Instead Of trigger* on a table for the Delete operation, they try to delete rows, and they will not actually get deleted (unless you issue another delete instruction from within the trigger)

INSTEAD OF TRIGGERS can be classified further into three types as:

INSTEAD OF INSERT Trigger.
INSTEAD OF UPDATE Trigger.
INSTEAD OF DELETE Trigger.

Let's create an Instead Of Delete Trigger as:

```
CREATE TRIGGER trgInsteadOfDelete ON [dbo].[Employee_Test]
INSTEAD OF DELETE
AS
        declare @emp_id int;
        declare @emp_name varchar(100);
        declare @emp_sal int;

        select @emp_id=d.Emp_ID from deleted d;
        select @emp_name=d.Emp_Name from deleted d;
        select @emp_sal=d.Emp_Sal from deleted d;

        BEGIN
                if(@emp_sal>1200)
                begin
                        RAISERROR('Cannot delete where salary > 1200',16,1);
                        ROLLBACK;

                end
                else
                begin
                        delete from Employee_Test where Emp_ID=@emp_id;
                        COMMIT;
                        insert into
Employee_Test_Audit(Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
                        values(@emp_id,@emp_name,@emp_sal,'Deleted -- Instead Of Delete
Trigger.',getdate());
                        PRINT 'Record Deleted -- Instead Of Delete Trigger.'

                end
        END
```

GO

This trigger will prevent the deletion of records from the table where Emp_Sal > 1200. If such a record is deleted, the Instead Of Trigger will rollback the transaction, otherwise the transaction will be committed. Now, let's try to delete a record with the Emp_Sal >1200 as:

```
delete from Employee_Test where Emp_ID=4
```

This will print an error message as defined in the RAISE ERROR statement as:

```
Server: Msg 50000, Level 16, State 1, Procedure trgInsteadOfDelete, Line 15
Cannot delete where salary > 1200
```

And this record will not be deleted.

In a similar way, you can code Instead of Insert and Instead Of Update triggers on your tables.

SUMMARY:

- A stored procedure is nothing more than prepared SQL code that you save so you can reuse the code over and over again.
- Can contain SQL Procedural Language statements and features which support the implementation of control-flow logic around traditional static and dynamic SQL statements.
- Easy to implement, because they use a simple high-level, strongly typed language.
- SQL procedures are more reliable than equivalent external procedures.
- Support input, output, and input-output parameter passing modes.
- Support a simple, but powerful condition and error-handling model.
- Return multiple results sets to the caller or to a client application.
- Allow you to easily access the SQLSTATE and SQLCODE values as special variables.
- Reside in the database and are automatically backed up and restored.
- Can be invoked wherever the CALL statement is supported.
- Support nested procedure calls to other SQL procedures or procedures implemented in other languages.

- SQL procedures can be used to create simple scripts for quickly querying transforming, updating data, generating basic reports, improve application performance, modularizing applications, and improve overall database design, and database security.

REVIEW QUESTIONS:

1. What is Stored Procedure?
2. What is the difference between Function and Stored Procedure?
3. What is the main purpose of triggers in database?
4. Difference between quires and stored procedure :
5. Why stored procedure is faster than normal SQL query?

University Of Karachi

Department Of Computer Science

DATABASE LAB

HANDOUT #12-13

Course: Database Systems

Semester: Spring 2018

Instructor: NAZISH ALI

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.

Sub queries: The Embedded SELECT Statement

Objectives

A subquery is a query whose results are passed as the argument for another query. Subqueries enable you to bind several queries together. By the end of the day, you will understand and be able to do the following:

What is subquery in SQL?

A subquery is a SQL query nested inside a larger query.

Building a Subquery

Simply put, a subquery lets you tie the result set of one query to another. The general syntax is as follows:

SYNTAX:

```
SELECT *
FROM TABLE1
WHERE TABLE1.SOMECOLUMN =
SELECT SOMEOTHERCOLUMN
FROM TABLE2
WHERE SOMEOTHERCOLUMN = SOMEVALUE)
```

- A subquery may occur in :
  - **- A SELECT clause**
  - **- A FROM clause**
  - **- A WHERE clause**
- The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
- A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
- You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
- A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
- The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

- **Compare an expression to the result of the query.**
- **Determine if an expression is included in the results of the query.**
- **Check whether the query selects any rows.**

Syntax:

```
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.

SQL Subqueries Example:

In this section, you will learn the requirements of using subqueries. We have the following two tables 'student' and 'marks' with common field 'StudentID'.

StudentID	Name
V001	Abe
V002	Abhay
V003	Acelin
V004	Adelphos

Student

StudentID	Total_marks
V001	95
V002	80
V003	74
V004	81

marks

Now we want to write a query to identify all students who get better marks than that of the student who's StudentID is 'V002', but we do not know the marks of 'V002'.
- To solve the problem, we require two queries.

One query returns the marks (stored in Total_marks field) of 'V002' and a second query identifies the students who get better marks than the result of the first query.

First query:

```
SELECT *
FROM `marks`
WHERE studentid = 'V002';
```

Query result:

StudentID	Total_marks
V002	80

The result of the query is 80.

Analysis:

Using the result of this query, here we have written another query to identify the students who get better marks than 80. Here is the query:

Second query:

```
SELECT a.studentid, a.name, b.total_marks
FROM student a, marks b
WHERE a.studentid = b.studentid
AND b.total_marks >80;
```

Query result:

studentid	name	total_marks
V001	Abe	95
V004	Adelphos	81

Above two queries identified students who get the better number than the student who's StudentID is 'V002' (Abhay).

Analysis:

You can combine the above two queries by placing one query inside the other. The subquery (also called the 'inner query') is the query inside the parentheses. See the following code and query result :

SQL Code:

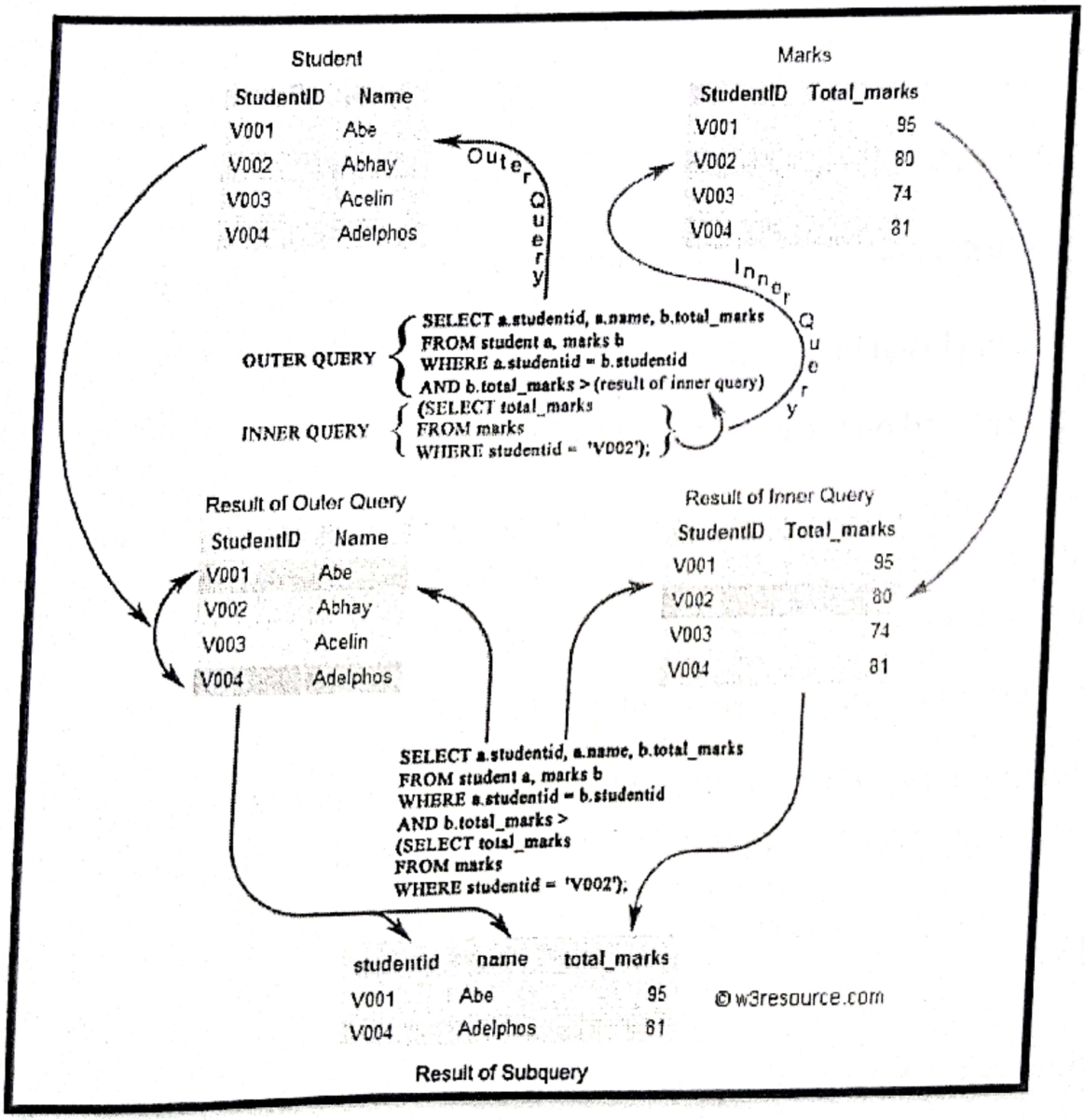
```
SELECT a.studentid, a.name, b.total_marks
FROM student a, marks b
```

```
WHERE a.studentid = b.studentid AND b.total_marks >
(SELECT total_marks
FROM marks
WHERE studentid = 'V002');
```

Query result:

studentid	name	total_marks
V001	Abe	95
V004	Adelphos	81

Pictorial Presentation of SQL Subquery:



Subqueries: General Rules

A subquery SELECT statement is almost similar to the SELECT statement and it is used to begin a regular or outer query. Here is the syntax of a subquery:

Syntax:

(SELECT [DISTINCT] subquery_select_argument

FROM {table_name }

{table_name } ...

[WHERE search_conditions]

[GROUP BY aggregate_expression [, aggregate_expression] ...]

[HAVING search_conditions])

Subqueries: Guidelines

There are some guidelines to consider when using subqueries:

- A subquery must be enclosed in parentheses.
- A subquery must be placed on the right side of the comparison operator.
- Subqueries cannot manipulate their results internally, therefore ORDER BY clause cannot be added into a subquery. You can use an ORDER BY clause in the main SELECT statement (outer query) which will be the last clause.
- Use single-row operators with single-row subqueries.
- If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

Type of Subqueries

- Single row subquery : Returns zero or one row.
- Multiple row subquery : Returns one or more rows.
- Multiple column subqueries : Returns one or more columns.

- *Correlated subqueries* : Reference one or more columns in the outer SQL statement. The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.
- *Nested subqueries* : Subqueries are placed within another subquery.

In the next session, we have thoroughly discussed the above topics. Apart from the above type of subqueries, you can use a subquery inside INSERT, UPDATE and DELETE statement. Here is a brief discussion :

Subqueries with INSERT statement

INSERT statement can be used with subqueries. Here are the syntax and an example of subqueries using INSERT statement.

Syntax:

INSERT INTO table_name [(column1 [, column2])]

SELECT [*|column1 [, column2]

FROM table1 [, table2]

[WHERE VALUE OPERATOR];

If we want to insert those orders from 'orders' table which have the advance_amount 2000 or 5000 into 'neworder' table the following SQL can be used:

Sample table: orders

ORD_NUM	ORD_AMOUNT	ADVANCE_AMOUNT	ORD_DATE	CUST_CODE	AGENT_CODE	ORD_DESCRIPTION
200114	3500	2000	15-AUG-08	C00002	A008	
200122	2500	400	16-SEP-08	C00003	A004	
200118	500	100	20-JUL-08	C00023	A006	
200119	4000	700	16-SEP-08	C00007	A010	
200121	1500	600	23-SEP-08	C00008	A004	
200130	2500	400	30-JUL-08	C00025	A011	
200134	4200	1800	25-SEP-08	C00004	A005	
200108	4000	600	15-FEB-08	C00008	A004	
200103	1500	700	15-MAY-08	C00021	A005	
200105	2500	500	18-JUL-08	C00025	A011	
200109	3500	800	30-JUL-08	C00011	A010	
200101	3000	1000	15-JUL-08	C00001	A008	
200111	1000	300	10-JUL-08	C00020	A008	
200104	1500	500	13-MAR-08	C00006	A004	

200106	2500			
200125	2000	700	20-APR-08 C00005	A002
200117	800	600	10-OCT-08 C00018	A005
200123	500	200	20-OCT-08 C00014	A001
200120	500	100	16-SEP-08 C00022	A002
200116	500	100	20-JUL-08 C00009	A002
200124	500	100	13-JUL-08 C00010	A009
200126	500	100	20-JUN-08 C00017	A007
200129	2500	100	24-JUN-08 C00022	A002
200127	2500	500	20-JUL-08 C00024	A006
200128	3500	400	20-JUL-08 C00015	A003
200135	2000	1500	20-JUL-08 C00009	A002
200131	900	600	10-SEP-08 C00007	A010
200133	1200	150	26-AUG-08 C00012	A012
200100	1000	400	29-JUN-08 C00009	A002
200110	3000	600	08-JAN-08 C00015	A003
200107	4500	500	15-APR-08 C00019	A010
200112	2000	900	30-AUG-08 C00007	A010
200113	4000	400	30-MAY-08 C00016	A007
200102	2000	600	10-JUN-08 C00022	A002
		300	25-MAY-08 C00012	A012

SQL Code:

```
INSERT INTO neworder
SELECT * FROM  orders
WHERE advance_amount in(2000,5000);
```

Output:

```
2 row(s)  inserted.

0.71 seconds
```

To see more details of subqueries using INSERT statement click here.

Subqueries with UPDATE statement

In a UPDATE statement, you can set new column value equal to the result returned by a single row subquery. Here are the syntax and an example of subqueries using UPDATE statement.

Syntax:

```
UPDATE table  SET column_name = new_value
```

[WHERE OPERATOR [VALUE]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[WHERE)]

If we want to update that ord_date in 'neworder' table with '15-JAN-10' which have the difference of ord_amount and advance_amount is less than the minimum ord_amount of 'orders' table the following SQL can be used:

Sample table: neworder

ORD_NUM	ORD_AMOUNT	ADVANCE_AMOUNT	ORD_DATE	CUST_CODE	AGENT_CODE	ORD_DESCRIPTION
200114	3500	2000	15-AUG-08	C00002	A008	
200122	2500	400	16-SEP-08	C00003	A004	
200118	500	100	20-JUL-08	C00023	A006	
200119	4000	700	16-SEP-08	C00007	A010	
200121	1500	600	23-SEP-08	C00008	A004	
200130	2500	400	30-JUL-08	C00025	A011	
200134	4200	1800	25-SEP-08	C00004	A005	
200108	4000	600	15-FEB-08	C00008	A004	
200103	1500	700	15-MAY-08	C00021	A005	
200105	2500	500	18-JUL-08	C00025	A011	
200109	3500	800	30-JUL-08	C00011	A010	
200101	3000	1000	15-JUL-08	C00001	A008	
200111	1000	300	10-JUL-08	C00020	A008	
200104	1500	500	13-MAR-08	C00006	A004	
200106	2500	700	20-APR-08	C00005	A002	
200125	2000	600	10-OCT-08	C00018	A005	
200117	800	200	20-OCT-08	C00014	A001	
200123	500	100	16-SEP-08	C00022	A002	
200120	500	100	20-JUL-08	C00009	A002	
200116	500	100	13-JUL-08	C00010	A009	
200124	500	100	20-JUN-08	C00017	A007	
200126	500	100	24-JUN-08	C00022	A002	
200129	2500	500	20-JUL-08	C00024	A006	
200127	2500	400	20-JUL-08	C00015	A003	
200128	3500	1500	20-JUL-08	C00009	A002	
200135	2000	800	16-SEP-08	C00007	A010	
200131	900	150	26-AUG-08	C00012	A012	
200133	1200	400	29-JUN-08	C00009	A002	
200100	1000	600	08-JAN-08	C00015	A003	
200110	3000	500	15-APR-08	C00019	A010	
200107	4500	900	30-AUG-08	C00007	A010	
200112	2000	400	30-MAY-08	C00016	A007	
200113	4000	600	10-JUN-08	C00022	A002	
200102	2000	300	25-MAY-08	C00012	A012	

SQL Code:

```
UPDATE neworder
SET ord_date='15-JAN-10'
WHERE ord_amount-advance_amount<
(SELECT MIN(ord_amount) FROM orders);
```

Output:

```
7 row(s) updated.

0.06 seconds
```

Subqueries with DELETE statement

DELETE statement can be used with subqueries. Here are the syntax and an example of subqueries using DELETE statement.

Syntax:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[ WHERE) ]
```

If we want to delete those orders from 'neworder' table which advance_amount are less than the maximum advance_amount of 'orders' table, the following SQL can be used:

Sample table: neworder

ORD_NUM	ORD_AMOUNT	ADVANCE_AMOUNT	ORD_DATE	CUST_CODE	AGENT_CODE	ORD_DESCRIPTION
200114	3500	2000	15-AUG-08	C00002	A008	
200122	2500	400	16-SEP-08	C00003	A004	
200118	500	100	20-JUL-08	C00023	A006	
200119	4000	700	16-SEP-08	C00007	A010	
200121	1500	600	23-SEP-08	C00008	A004	
200130	2500	400	30-JUL-08	C00025	A011	
200134	4200	1800	25-SEP-08	C00004	A005	
200108	4000	600	15-FEB-08	C00008	A004	
200103	1500	700	15-MAY-08	C00021	A005	

200105	2500	500	18-JUL-08	C00025	A011
200109	3500	800	30-JUL-08	C00011	A010
200101	3000	1000	15-JUL-08	C00001	A008
200111	1000	300	10-JUL-08	C00020	A008
200104	1500	500	13-MAR-08	C00006	A004
200106	2500	700	20-APR-08	C00005	A002
200125	2000	600	10-OCT-08	C00018	A005
200117	800	200	20-OCT-08	C00014	A001
200123	500	100	16-SEP-08	C00022	A002
200120	500	100	20-JUL-08	C00009	A002
200116	500	100	13-JUL-08	C00010	A009
200124	500	100	20-JUN-08	C00017	A007
200126	500	100	24-JUN-08	C00022	A002
200129	2500	500	20-JUL-08	C00024	A006
200127	2500	400	20-JUL-08	C00015	A003
200128	3500	1500	20-JUL-08	C00009	A002
200135	2000	800	16-SEP-08	C00007	A010
200131	900	150	26-AUG-08	C00012	A012
200133	1200	400	29-JUN-08	C00009	A002
200100	1000	600	08-JAN-08	C00015	A003
200110	3000	500	15-APR-08	C00019	A010
200107	4500	900	30-AUG-08	C00007	A010
200112	2000	400	30-MAY-08	C00016	A007
200113	4000	600	10-JUN-08	C00022	A002
200102	2000	300	25-MAY-08	C00012	A012

SQL Code:

```
DELETE FROM neworder
WHERE advance_amount<
(SELECT MAX(advance_amount) FROM orders);
```

Output:

```
34 row(s) deleted.

0.04 seconds
```

Example -1 : Nested subqueries

If we want to retrieve that unique job_id and there average salary from the employees table which unique job_id have a salary is smaller than (the maximum of averages of min_salary of each unique job_id from the jobs table which job_id are in the list, picking from (the job_history table which is within the department_id 50 and 100)) the following SQL statement can be used

Sample table: employees

employee_id	first_name	last_name	email	phone_number	hire_date	job_id	salary	commission_pct	manager_id	department_id
100	Steven	King	SKING	515.123.4567	6/17/1987	AD_PRES	24000			90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	6/18/1987	AD_VP	17000		100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	6/19/1987	AD_VP	17000		100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	6/20/1987	IT_PROG	9000		102	60
104	Bruce	Ernst	BERNST	590.423.4568	6/21/1987	IT_PROG	6000		103	60
105	David	Austin	DAUSTIN	590.423.4569	6/22/1987	IT_PROG	4800		103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	6/23/1987	IT_PROG	4800		103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	6/24/1987	IT_PROG	4200		103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	6/25/1987	FI_MGR	12000		101	100
109	Daniel	Faviet	DFAVIET	515.124.4169	6/26/1987	FI_ACCOUNT	9000		108	100
110	John	Chen	JCHEN	515.124.4269	6/27/1987	FI_ACCOUNT	8200		108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	6/28/1987	FI_ACCOUNT	7700		108	100
112	Jose Manue	Urman	JMURMAN	515.124.4469	6/29/1987	FI_ACCOUNT	7800		108	100
113	Luis	Popp	LPOPP	515.124.4567	6/30/1987	FI_ACCOUNT	6900		108	100
114	Den	Raphaely	DRAPHEAL	515.127.4561	7/1/1987	PU_MAN	11000		100	30
115	Alexander	Khoo	AKHOO	515.127.4562	7/2/1987	PU_CLERK	3100		114	30
116	Shelli	Baida	SBAIDA	515.127.4563	7/3/1987	PU_CLERK	2900		114	30
117	Sigal	Tobias	STOBIAS	515.127.4564	7/4/1987	PU_CLERK	2800		114	30
118	Guy	Himuro	GHIMURO	515.127.4565	7/5/1987	PU_CLERK	2600		114	30
119	Karen	Colmenares	KCOLMENA	515.127.4566	7/6/1987	PU_CLERK	2500		114	30
120	Matthew	Weiss	MWEISS	650.123.1234	7/7/1987	ST_MAN	8000		100	50
121	Adam	Fripp	AFRIPP	650.123.2234	7/8/1987	ST_MAN	8200		100	50
122	Payam	Kaufling	PKAUFLIN	650.123.3234	7/9/1987	ST_MAN	7900		100	50
123	Shanta	Vollman	SVOLLMAN	650.123.4234	7/10/1987	ST_MAN	6500		100	50
124	Kevin	Mourgos	KMOURGOS	650.123.5234	7/11/1987	ST_MAN	5800		100	50

125	Julia	Nayer	JNAYER		650.124.1214		
ST_CLERK	3200			120		50	7/12/1987
126	Irene	Mikkilinen	IMIKKILI		650.124.1224		
ST_CLERK	2700			120		50	7/13/1987
127	James	Landry	JLANDRY		650.124.1334		
ST_CLERK	2400			120		50	7/14/1987
128	Steven	Markle	SMARKLE		650.124.1434		
ST_CLERK	2200			120		50	7/15/1987
129	Laura	Bissot	LBISSOT		650.124.5234		
ST_CLERK	3300			121		50	7/16/1987
130	Mozhe	Atkinson	MATKINSO		650.124.6234		
ST_CLERK	2800			121		50	7/17/1987
131	James	Marlow	JAMRLOW		650.124.7234		
ST_CLERK	2500			121		50	7/18/1987
132	TJ	Olson	TJOLSON		650.124.8234		
ST_CLERK	2100			121		50	7/19/1987
133	Jason	Mallin	JMALLIN		650.127.1934		
ST_CLERK	3300			122		50	7/20/1987
134	Michael	Rogers	MROGERS		650.127.1834		
ST_CLERK	2900			122		50	7/21/1987
135	Ki	Gee	KGEE		650.127.1734		
ST_CLERK	2400			122		50	7/22/1987
136	Hazel	Philtanker	HPHILTAN		650.127.1634		
ST_CLERK	2200			122		50	7/23/1987
137	Renske	Ladwig	RLADWIG		650.121.1234		
ST_CLERK	3600			123		50	7/24/1987
138	Stephen	Stiles	SSTILES		650.121.2034		
ST_CLERK	3200			123		50	7/25/1987
139	John	Seo	JSEO		650.121.2019		
ST_CLERK	2700			123		50	7/26/1987
140	Joshua	Patel	JPATEL		650.121.1834		
ST_CLERK	2500			123		50	7/27/1987
141	Trenna	Rajs	TRAJS		650.121.8009		
ST_CLERK	3500			124		50	7/28/1987
142	Curtis	Davies	CDAVIES		650.121.2994		
ST_CLERK	3100			124		50	7/29/1987
143	Randall	Matos	RMATOS		650.121.2874		
ST_CLERK	2600			124		50	7/30/1987
144	Peter	Vargas	PVARGAS		650.121.2004		
ST_CLERK	2500			124		50	7/31/1987
145	John	Russell	JRUSSEL		011.44.1344.		
SA_MAN	14000	0.4		100		80	8/1/1987
146	Karen	Partners	KPARTNER		011.44.1344.		
SA_MAN	13500	0.3		100		80	8/2/1987
147	Alberto	Errazuriz	AERRAZUR		011.44.1344.		
SA_MAN	12000	0.3		100		80	8/3/1987
148	Gerald	Cambrault	GCAMBRAU		011.44.1344.		
SA_MAN	11000	0.3		100		80	8/4/1987
149	Eleni	Zlotkey	EZLOTKEY		011.44.1344.		
SA_MAN	10500	0.2		100		80	8/5/1987
150	Peter	Tucker	PTUCKER		011.44.1344.		
SA_REP	10000	0.3		145		80	8/6/1987
151	David	Bernstein	DBERNSTE		011.44.1344.		
SA_REP	9500	0.25		145		80	8/7/1987
152	Peter	Hall	PHALL		011.44.1344.		
SA_REP	9000	0.25		145		80	8/8/1987

153	Christophe	Olsen	0.2	COLSEN	011.44.1344.	8/9/1987
SA_REP	8000	Cambrault		145	80	
154	Nanette		0.2	NCAMBRAU	011.44.1344.	8/10/1987
SA_REP	7500	Tuvault		145	80	
155	Oliver		0.15	OTUVAULT	011.44.1344.	8/11/1987
SA_REP	7000	King		145	80	
156	Janette		0.35	JKING	011.44.1345.	8/12/1987
SA_REP	10000	Sully		146	80	
157	Patrick		0.35	PSULLY	011.44.1345.	8/13/1987
SA_REP	9500	McEwen		146	80	
158	Allan		0.35	AMCEWEN	011.44.1345.	8/14/1987
SA_REP	9000	Smith		146	80	
159	Lindsey		0.3	LSMITH	011.44.1345.	8/15/1987
SA_REP	8000	Doran		146	80	
160	Louise		0.3	LDORAN	011.44.1345.	8/16/1987
SA_REP	7500	Sewall		146	80	
161	Sarath		0.25	SSEWALL	011.44.1345.	8/17/1987
SA_REP	7000	Vishney		146	80	
162	Clara		0.25	CVISHNEY	011.44.1346.	8/18/1987
SA_REP	10500	Greene		147	80	
163	Danielle		0.15	DGREENE	011.44.1346.	8/19/1987
SA_REP	9500	Marvins		147	80	
164	Mattea		0.1	MMARVINS	011.44.1346.	8/20/1987
SA_REP	7200	Lee		147	80	
165	David		0.1	DLEE	011.44.1346.	8/21/1987
SA_REP	6800	Ande		147	80	
166	Sundar		0.1	SANDE	011.44.1346.	8/22/1987
SA_REP	6400	Banda		147	80	
167	Amit		0.1	ABANDA	011.44.1346.	8/23/1987
SA_REP	6200	Ozer		147	80	
168	Lisa		0.25	LOZER	011.44.1343.	8/24/1987
SA_REP	11500	Bloom		148	80	
169	Harrison		0.2	HBLOOM	011.44.1343.	8/25/1987
SA_REP	10000	Fox		148	80	
170	Tayler		0.2	TFOX	011.44.1343.	8/26/1987
SA_REP	9600	Smith		148	80	
171	William		0.15	WSMITH	011.44.1343.	8/27/1987
SA_REP	7400	Bates		148	80	
172	Elizabeth		0.15	EBATES	011.44.1343.	8/28/1987
SA_REP	7300	Kumar		148	80	
173	Sundita		0.1	SKUMAR	011.44.1343.	8/29/1987
SA_REP	6100	Abel		148	80	
174	Ellen		0.3	EABEL	011.44.1644.	8/30/1987
SA_REP	11000	Hutton		149	80	
175	Alyssa		0.25	AHUTTON	011.44.1644.	8/31/1987
SA_REP	8800	Taylor		149	80	
176	Jonathon		0.2	JTAYLOR	011.44.1644.	9/1/1987
SA_REP	8600	Livingston		149	80	
177	Jack		0.2	JLIVINGS	011.44.1644.	9/2/1987
SA_REP	8400	Grant		149	80	
178	Kimberely		0.15	KGRANT	011.44.1644.	9/3/1987
SA_REP	7000	Johnson		149		
179	Charles		0.1	CJOHNSON	011.44.1644.	9/4/1987
SA_REP	6200	Taylor		149	80	
180	Winston			WTAYLOR	650.507.9876	9/5/1987
SH_CLERK	3200			120	50	

181	SH_CLERK	Jean	3100	Fleaur	JFLEAUR	650.507.9877	9/6/1987
182	SH_CLERK	Martha	2500	Sullivan	MSULLIVA	650.507.9878	9/7/1987
183	SH_CLERK	Girard	2800	Geoni	GGEONI	650.507.9879	9/8/1987
184	SH_CLERK	Nandita	4200	Sarchand	NSARCHAN	650.509.1876	9/9/1987
185	SH_CLERK	Alexis	4100	Bull	ABULL	650.509.2876	9/10/1987
186	SH_CLERK	Julia	3400	Dellinger	JDELLING	650.509.3876	9/11/1987
187	SH_CLERK	Anthony	3000	Cabrio	ACABRIO	650.509.4876	9/12/1987
188	SH_CLERK	Kelly	3800	Chung	KCHUNG	650.505.1876	9/13/1987
189	SH_CLERK	Jennifer	3600	Dilly	JDILLY	650.505.2876	9/14/1987
190	SH_CLERK	Timothy	2900	Gates	TGATES	650.505.3876	9/15/1987
191	SH_CLERK	Randall	2500	Perkins	RPERKINS	650.505.4876	9/16/1987
192	SH_CLERK	Sarah	4000	Bell	SBELL	650.501.1876	9/17/1987
193	H_CLERK	Britney	3900	Everett	BEVERETT	650.501.2876	9/18/1987
194	H_CLERK	Samuel	3200	McCain	SMCCAIN	650.501.3876	9/19/1987
195	H_CLERK	Vance	2800	Jones	VJONES	650.501.4876	9/20/1987
196	H_CLERK	Alana	3100	Walsh	AWALSH	650.507.9811	9/21/1987
197	H_CLERK	Kevin	3000	Feeney	KFEENEY	650.507.9822	9/22/1987
198	H_CLERK	Donald	2600	OConnell	DOCONNEL	650.507.9833	9/23/1987
199	H_CLERK	Douglas	2600	Grant	DGRANT	650.507.9844	9/24/1987
200	_ASST	Jennifer	4400	Whalen	JWHALEN	515.123.4444	9/25/1987
201	_MAN	Michael	13000	Hartstein	MHARTSTE	515.123.5555	9/26/1987
202	_REP	Pat	6000	Fay	PFAY	603.123.6666	9/27/1987
203	_REP	Susan	6500	Mavris	SMAVRIS	515.123.7777	9/28/1987
204	_REP	Hermann	10000	Baer	HBAER	515.123.8888	9/29/1987
205	_MGR	Shelley	12000	Higgins	SHIGGINS	515.123.8080	9/30/1987
206	_ACCOUNT	William	8300	Gietz	WGIETZ	515.123.8181	10/1/1987

Sample table: jobs

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

SQL Code:

```
SELECT job_id,AVG(salary)
FROM employees
GROUP BY job_id  HAVING AVG(salary)<(SELECT MAX(AVG(min_salary))
FROM jobs   WHERE job_id IN    (SELECT job_id FROM job_history  WHERE department_id
BETWEEN 50 AND 100)    GROUP BY job_id);
```

Output

```
JOB_ID    AVG(SALARY)
---------- -----------
IT_PROG        5760
```

ACCOUNT	8300
MAN	7280
ASST	4400
CLERK	3215
ACCOUNT	7920
CLERK	2780
REP	8350
REP	6000
CLERK	2785
REP	6500

ANALYSIS:

This example contains three queries: a nested subquery, a subquery, and the outer query. These parts of queries are runs in that order.

Let's break the example down into three parts and observes the results returned. Atfirst the nested subquery as follows:

SQL Code:

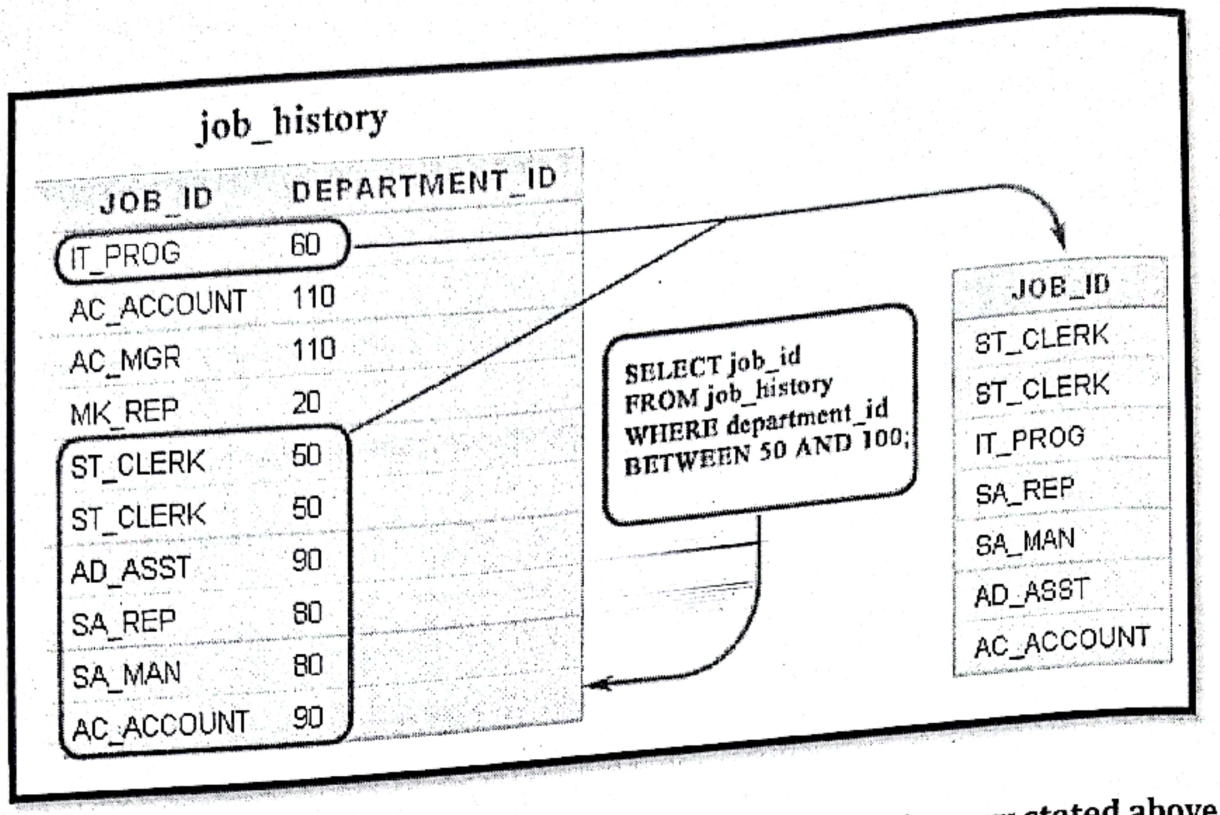
```
SELECT job_id FROM job_history
WHERE department_id
BETWEEN 50 AND 100;
```

This nested subquery retrieves the job_id(s) from job_history table which is within the department_id 50 and 100.

Output:

```
JOB_ID
---------
ST_CLERK
ST_CLERK
IT_PROG
SA_REP
SA_MAN
AD_ASST
AC_ACCOUNT
```

Here is the pictorial representation of how the above output comes.



Now the subquery that receives output from the nested subquery stated above.**

SELECT MAX(AVG(min_salary))
FROM jobs WHERE job_id
IN(.....output from the nested subquery.....)
GROUP BY job_id
The subquery internally works as follows:

SQL Code:

```
SELECT MAX(AVG(min_salary))
FROM jobs
WHERE job_id
IN(
'ST_CLERK','ST_CLERK','IT_PROG',
'SA_REP','SA_MAN','AD_ASST',
AC_ACCOUNT')
GROUP BY job_id;
```

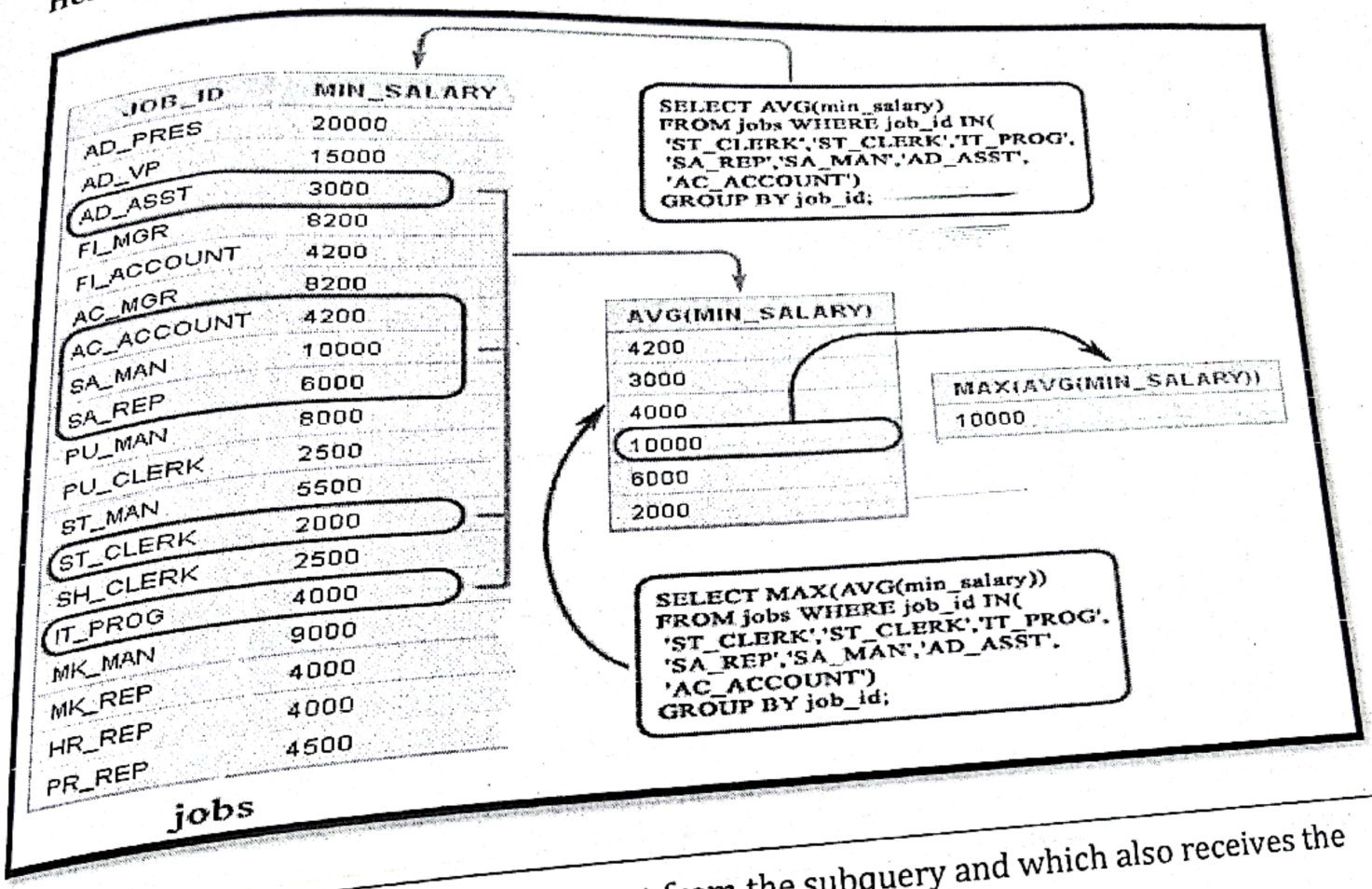
The subquery returns the maximum of averages of min_salary for each unique job_id return (i.e. 'ST_CLERK','ST_CLERK','IT_PROG', 'SA_REP','SA_MAN','AD_ASST', 'AC_ACCOUNT') by the previous subquery.

Here is the output:

Output:

```
MAX(AVG(MIN_SALARY))
--------------------
               10000
```

Here is the pictorial representation of how the above output returns.



Now the outer query that receives output from the subquery and which also receives the output from the nested subquery stated above.

```
SELECT job_id,AVG(salary)
FROM employees
GROUP BY job_id
HAVING AVG(salary)<
(.....output from the subquery(
output from the nested subquery).....)
```

The outer query internally works as follows:

SQL Code:

```
SELECT job_id,AVG(salary)
FROM employees
GROUP BY job_id
HAVING AVG(salary)<10000;
```

The outer query returns the job_id, average salary of employees that are less than maximum of average of min_salary returned by the previous query

Output:

```
JOB_ID    AVG(SALARY)
---------- -----------
IT_PROG        5760
AC_ACCOUNT     8300
ST_MAN         7280
AD_ASST        4400
SH_CLERK       3215
FI_ACCOUNT     7920
PU_CLERK       2780
SA_REP         8350
MK_REP         6000
ST_CLERK       2785
HR_REP         6500
```

Example -2: Nested subqueries

Here is an another nested subquery example.

Sample table: orders

```
ORD_NUM ORD_AMOUNT ADVANCE_AMOUNT ORD_DATE  CUST_CODE       AGENT_CODE
ORD_DESCRIPTION
--------- ---------- -------------- --------- -------------- -------------
- -----------------
    200114       3500           2000 15-AUG-08 C00002         A008
    200122       2500            400 16-SEP-08 C00003         A004
    200118        500            100 20-JUL-08 C00023         A006
    200119       4000            700 16-SEP-08 C00007         A010
    200121       1500            600 23-SEP-08 C00008         A004
    200130       2500            400 30-JUL-08 C00025         A011
    200134       4200           1800 25-SEP-08 C00004         A005
    200108       4000            600 15-FEB-08 C00008         A004
    200103       1500            700 15-MAY-08 C00021         A005
    200105       2500            500 18-JUL-08 C00025         A011
```

200109	3500	800	30-JUL-08	C00011	A010				
200101	3000	1000	15-JUL-08	C00001	A008				
200111	1000	300	10-JUL-08	C00020	A009				
200104	1500	500	13-MAR-08	C00006	A004				
200106	2500	700	20-APR-08	C00005	A002				
200125	2000	600	10-OCT-08	C00018	A005				
200117	800	200	20-OCT-08	C00014	A001				
200123	500	100	16-SEP-08	C00022	A002				
200120	500	100	20-JUL-08	C00009	A002				
200116	500	100	13-JUL-08	C00010	A009				
200124	500	100	20-JUN-08	C00017	A007				
200126	500	500	24-JUN-08	C00022	A002				
200129	2500	400	20-JUL-08	C00024	A006				
200127	2500	1500	20-JUL-08	C00015	A003				
200128	3500	800	16-SEP-08	C00009	A002				
200135	2000	150	26-AUG-08	C00007	A010				
200131	900	400	29-JUN-08	C00012	A012				
200133	1200	600	08-JAN-08	C00009	A002				
200100	1000	500	15-APR-08	C00015	A003				
200110	3000	900	30-AUG-08	C00019	A010				
200107	4500	400	30-MAY-08	C00007	A010				
200112	2000	600	10-JUN-08	C00016	A007				
200113	4000	300	25-MAY-08	C00022	A002				
200102	2000			C00012	A012				

Sample table : customer

CUST_CODE	CUST_NAME	CUST_CITY	WORKING_AREA	CUST_COUNTRY	GRADE	OPENING_AMT	RECEIVE_AMT	PAYMENT_AMT	OUTSTANDING_AMT	PHONE_NO	AGENT_CODE
C00013	Holmes	London	London	UK	2	6000.00	5000.00	7000.00	4000.00	BBBBBBB	A003
C00001	Micheal	New York	New York	USA	2	3000.00	5000.00	2000.00	6000.00	CCCCCCC	A008
C00020	Albert	New York	New York	USA	3	5000.00	7000.00	6000.00	6000.00	BBBBSBB	A008
C00025	Ravindran	Bangalore	Bangalore	India	2	5000.00	7000.00	4000.00	8000.00	AVAVAVA	A011
C00024	Cook	London	London	UK	2	4000.00	9000.00	7000.00	6000.00	FSDDSDF	A006
C00015	Stuart	London	London	UK	1	6000.00	8000.00	3000.00	11000.00	GFSGERS	A003

| C00002 | Bolt | New York | New York | USA | 3
| 5000.00 | 7000.00 | 9000.00 | 3000.00 | DDNRDRH |
A008 |
| C00018 | Fleming | Brisban | Brisban | Australia | 2
| 7000.00 | 7000.00 | 9000.00 | 5000.00 | NHBGVFC |
A005 |
| C00021 | Jacks | Brisban | Brisban | Australia | 1
| 7000.00 | 7000.00 | 7000.00 | 7000.00 | WERTGDF |
A005 |
| C00019 | Yearannaidu | Chennai | Chennai | India | 1
| 8000.00 | 7000.00 | 7000.00 | 8000.00 | ZZZZBFV |
A010 |
| C00005 | Sasikant | Mumbai | Mumbai | India | 1
| 7000.00 | 11000.00 | 7000.00 | 11000.00 | 147-25896312 |
A002 |
| C00007 | Ramanathan | Chennai | Chennai | India | 1
| 7000.00 | 11000.00 | 9000.00 | 9000.00 | GHRDWSD |
A010 |
| C00022 | Avinash | Mumbai | Mumbai | India | 2
| 7000.00 | 11000.00 | 9000.00 | 9000.00 | 113-12345678 |
A002 |
| C00004 | Winston | Brisban | Brisban | Australia | 1
| 5000.00 | 8000.00 | 7000.00 | 6000.00 | AAAAAAA |
A005 |
| C00023 | Karl | London | London | UK | 0
| 4000.00 | 6000.00 | 7000.00 | 3000.00 | AAAABAA |
A006 |
| C00006 | Shilton | Torento | Torento | Canada | 1
| 10000.00 | 7000.00 | 6000.00 | 11000.00 | DDDDDDD |
A004 |
| C00010 | Charles | Hampshair | Hampshair | UK | 3
| 6000.00 | 4000.00 | 5000.00 | 5000.00 | MMMMMMM |
A009 |
| C00017 | Srinivas | Bangalore | Bangalore | India | 2
| 8000.00 | 4000.00 | 3000.00 | 9000.00 | AAAAAAB |
A007 |
| C00012 | Steven | San Jose | San Jose | USA | 1
| 5000.00 | 7000.00 | 9000.00 | 3000.00 | KRFYGJK |
A012 |
| C00008 | Karolina | Torento | Torento | Canada | 1
| 7000.00 | 7000.00 | 9000.00 | 5000.00 | HJKORED |
A004 |
| C00003 | Martin | Torento | Torento | Canada | 2
| 8000.00 | 7000.00 | 7000.00 | 8000.00 | MJYURFD |
A004 |
| C00009 | Ramesh | Mumbai | Mumbai | India | 3
| 8000.00 | 7000.00 | 3000.00 | 12000.00 | Phone No |
A002 |
| C00014 | Rangarappa | Bangalore | Bangalore | India | 2
| 8000.00 | 11000.00 | 7000.00 | 12000.00 | AAAATGF |
A001 |
| C00016 | Venkatpati | Bangalore | Bangalore | India | 2
| 8000.00 | 11000.00 | 7000.00 | 12000.00 | JRTVFDD |
A007 |

C00011	Sundariya	Chennai	7000.00	Chennai	India	PPHGRT3	3
7000.00	11000.00			11000.00			
A010							

Sample table : agents

AGENT_CODE	AGENT_NAME	WORKING_AREA	COMMISSION	PHONE_NO	COUNTRY
A007	Ramasundar	Bangalore	0.15	077-25814763	
A003	Alex	London	0.13	075-12458969	
A008	Alford	New York	0.12	044-25874365	
A011	Ravi Kumar	Bangalore	0.15	077-45625874	
A010	Santakumar	Chennai	0.14	007-22388644	
A012	Lucida	San Jose	0.12	044-52981425	
A005	Anderson	Brisban	0.13	045-21447739	
A001	Subbarao	Bangalore	0.14	077-12346674	
A002	Mukesh	Mumbai	0.11	029-12358964	
A006	McDen	London	0.15	078-22255588	
A004	Ivan	Torento	0.15	008-22544166	
A009	Benjamin	Hampshair	0.11	008-22536178	

SQL Code:

```
SELECT ord_num,ord_date,ord_amount,advance_amount
FROM orders
WHERE ord_amount>2000
AND ord_date<'01-SEP-08'
AND ADVANCE_AMOUNT <
ANY(SELECT OUTSTANDING_AMT
FROM CUSTOMER
```

```
WHERE GRADE=3
AND CUST_COUNTRY<>'India'
AND opening_amt<7000
AND EXISTS
(SELECT *
FROM agents
WHERE commission<.12));
```

Output:

ORD_NUM	ORD_DATE	ORD_AMOUNT	ADVANCE_AMOUNT
200130	30-JUL-08	2500	400
200127	20-JUL-08	2500	400
200110	15-APR-08	3000	500
200105	18-JUL-08	2500	500
200129	20-JUL-08	2500	500
200108	15-FEB-08	4000	600
200113	10-JUN-08	4000	600
200106	20-APR-08	2500	700
200109	30-JUL-08	3500	800
200107	30-AUG-08	4500	900
200101	15-JUL-08	3000	1000
200128	20-JUL-08	3500	1500
200114	15-AUG-08	3500	2000

Analysis:

The last Inner query will fetched the rows from agents table who have commission is less than .12%.

The 2nd last inner query returns the outstanding amount for those customers who are in grade 3 and not belongs to the country India and their deposited opening amount is less than 7000 and their agents should have earned a commission is less than .12%.

The outer query returns ord_num, ord_date, ord_amount, advance_amount for those orders from orders table which ord_amount is more than 2000 and ord_date before the '01-sep-08' and the advance amount may be the outstanding amount for those customers who are in grade 3 and not belongs to the country India and there deposited opening amount is less than 7000 and their agents should have earned a commission is less than .12%.

Let's break the code and analyze what's going on in inner query. Here is the first code of inner query with output :

SQL Code:

```
SELECT *
FROM agents
WHERE commission<.12;
```

Output:

```
AGENT_CODE AGENT_NAME    WORKING_AREA    COMMISSION PHONE_NO
COUNTRY
---------- ---------- ---------- ---------- ----------
A009       Benjamin    Hampshair          .11 008-22536178
A002       Mukesh      Mumbai             .11 029-12358964
```

Here is the second code of inner query (including first one) with output :

SQL Code:

```
SELECT OUTSTANDING_AMT
FROM CUSTOMER
WHERE GRADE=3
AND CUST_COUNTRY<>'India'
AND opening_amt<7000
AND EXISTS(
SELECT *
FROM agents
WHERE commission<.12);
```

Output:

```
OUTSTANDING_AMT
---------------
        6000
        3000
        5000
```

REVIEW QUESTIONS:

1. What is SQL Sqbquery?
2. Where Subquery Occurs?
3. Write Type of Subqueries:
4. Why we used nested Sqbquery
5. Why are correlated subqueries used?

SUMMARY:

In SQL Server, a subquery is a query within a query. You can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

A subquery is a SQL query within a query. Subqueries are nested queries that provide data to the enclosing query. Subqueries can return individual values or a list of records Subqueries must be enclosed with parenthesis

SQL subquery is usually added in the WHERE Clause of the SQL statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value in the database.