

a) Background

SQL (Structured Query Language) is a database computer language designed for managing data in relational database management systems (RDBMS). SQL, is a standardized computer language that was originally developed by IBM for querying, altering and defining relational databases, using declarative statements.

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

Even if SQL is a standard, many of the database systems that exist today implement their own version of the SQL language. In this document, we will use the MySQL as an example. There are lots of different database systems, or DBMS – Database Management Systems, such as:

- MySQL (Oracle, previously Sun Microsystems) - MySQL can be used free of charge (open source license), Web sites that use MySQL: YouTube, Wikipedia, Facebook
- Microsoft SQL Server
 - Enterprise, Developer versions, etc.
 - Express version is free of charge
- Oracle
- Microsoft Access
- IBM DB2
- Sybase
- ... lots of other systems

b) Basic Structure

SQL is based on set and relational operations with certain modifications and enhancements

- A typical SQL query has the form:

select A1, A2, ..., An

from r1, r2, ..., rm

where P

– A is represent attributes

– ris represent relations

– P is a predicate.

- This query is equivalent to the relational algebra expression: $\Pi_{A1, A2, \dots, An}(\sigma_P(r1 \times r2 \times \dots \times rm))$
- The result of an SQL query is a relation.

c) DDL Data Definition Language

The **Data Definition Language (DDL)** manages table and index structure. The most basic items of DDL are the CREATE, ALTER, RENAME and DROP statements:

- **CREATE** creates an object (a table, for example) in the database.
- **DROP** deletes an object in the database, usually irretrievably.
- **ALTER** modifies the structure an existing object in various ways—for example, adding a column to an existing table.

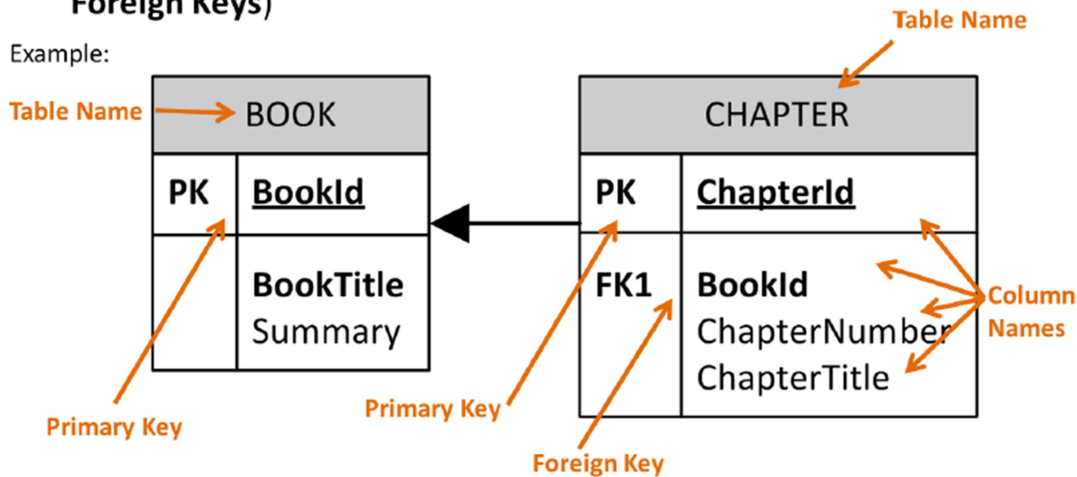
Before you start implementing your tables in the database, you should always spend some time design your tables properly using a design tool like, e.g., ERwin, Toad Data Modeler, PowerDesigner, Visio, etc. This is called Database Modeling.

Database Design – ER Diagram

ER Diagram (Entity-Relationship Diagram)

- Used for Design and Modeling of Databases.
- Specify Tables and **relationship** between them (**Primary Keys** and **Foreign Keys**)

Example:



Relational Database. In a relational database all the tables have one or more relation with each other using Primary Keys (PK) and Foreign Keys (FK). Note! You can only have one PK in a table, but you may have several FK's.

Create Table

The **CREATE TABLE** statement is used to create a table in a database.

Syntax:

```
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
column_name3 data_type,
....
)
```

The data type specifies what type of data the column can hold.

You have special data types for numbers, text dates, etc.

Examples:

- Numbers: **int**, **float**
- Text/Stings: **varchar(X)** – where X is the length of the string
- Dates: **datetime**
- etc.

	Column Name	Data Type	Allow Nulls
PK	CustomerId	int	<input type="checkbox"/>
	CustomerNumber	int	<input type="checkbox"/>
	LastName	varchar(50)	<input type="checkbox"/>
	FirstName	varchar(50)	<input type="checkbox"/>
	AreaCode	int	<input checked="" type="checkbox"/>
	Address	varchar(50)	<input checked="" type="checkbox"/>
	Phone	varchar(20)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

```
CREATE TABLE CUSTOMER
(
    CustomerId int IDENTITY(1,1) PRIMARY KEY,
    CustomerNumber int NOT NULL UNIQUE,
    LastName varchar(50) NOT NULL,
    FirstName varchar(50) NOT NULL,
    AreaCode int NULL,
    Address varchar(50) NULL,
    Phone varchar(50) NULL,
)
GO
```

Best practice:

When creating tables you should consider following these guidelines:

- Tables: Use upper case and singular form in table names – not plural, e.g., “STUDENT” (not students)
- Columns: Use Pascal notation, e.g., “StudentId”
- Primary Key:
 - o If the table name is “COURSE”, name the Primary Key column “CourseId”, etc.
 - o “Always” use Integer and Identity(1,1) for Primary Keys. Use UNIQUE constraint for other columns that needs to be unique, e.g. RoomNumber
- Specify Required Columns (NOT NULL) – i.e., which columns that need to have data or not
- Standardize on few/these Data Types: int, float, varchar(x), datetime, bit
- Use English for table and column names
- Avoid abbreviations! (Use RoomNumber – not RoomNo, RoomNr, ...)

ALTER and DROP

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table. To add a column in a table, use the following syntax:

```
ALTER TABLE table_name  
ADD column_name datatype
```

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name DROP COLUMN column_name
```

To change the data type of a column in a table, use the following syntax:

```
ALTER TABLE table_name ALTER COLUMN column_name datatype
```

If we use CREATE TABLE and the table already exists in the table we will get an error message, so if we combine CREATE TABLE and ALTER TABLE we can create robust database scripts that gives no errors, as the example shown below:

```
if not exists (select * from dbo.sysobjects where id = object_id(N'[CUSTOMER]') and  
OBJECTPROPERTY(id, N'IsUserTable') = 1)  
CREATE TABLE CUSTOMER  
(  
  CustomerId int PRIMARY KEY,  
  CustomerNumber int NOT NULL UNIQUE,  
  LastName varchar(50) NOT NULL,  
  FirstName varchar(50) NOT NULL,  
  AreaCode int NULL,  
  Address varchar(50) NULL,  
  Phone varchar(50) NULL,  
)  
GO
```

```
if exists(select * from dbo.syscolumns where id = object_id(N'[CUSTOMER]') and  
OBJECTPROPERTY(id, N'IsUserTable') = 1 and name = 'CustomerId')  
ALTER TABLE CUSTOMER ALTER COLUMN CustomerId int  
Else  
ALTER TABLE CUSTOMER ADD CustomerId int  
GO
```

```
if exists(select * from dbo.syscolumns where id = object_id(N'[CUSTOMER]') and  
OBJECTPROPERTY(id, N'IsUserTable') = 1 and name = 'CustomerNumber')  
ALTER TABLE CUSTOMER ALTER COLUMN CustomerNumber int  
Else  
ALTER TABLE CUSTOMER ADD CustomerNumber int  
GO  
...
```

d) Database Modification

The **Data Manipulation Language (DML)** is the subset of SQL used to add, update and delete data.

The acronym **CRUD** refers to all of the major functions that need to be implemented in a relational database application to consider it complete. Each letter in the acronym can be mapped to a standard SQL statement:

Operation	SQL	Description
Create	INSERT INTO	inserts new data into a database
Read (Retrieve)	SELECT	extracts data from a database
Update	UPDATE	updates data in a database
Delete (Destroy)	DELETE	deletes data from a database

INSERT INTO

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

It is possible to write the INSERT INTO statement in two forms.

The first form doesn't specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name  
VALUES (value1, value2, value3,...)
```

Example:

```
INSERT INTO CUSTOMER VALUES ('1000', 'Smith', 'John', 12,  
'California', '11111111')
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3,...)  
VALUES (value1, value2, value3,...)
```

This form is recommended!

Example:

```
INSERT INTO CUSTOMER (CustomerNumber, LastName, FirstName, AreaCode,  
Address, Phone)  
VALUES ('1000', 'Smith', 'John', 12, 'California', '11111111')
```

Insert Data Only in Specified Columns:

It is also possible to only add data in specific columns.

Example:

```
INSERT INTO CUSTOMER (CustomerNumber, LastName, FirstName)
VALUES ('1000', 'Smith', 'John')
```

Note! You need at least to include all columns that cannot be NULL.

UPDATE

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

The syntax is as follows:

UPDATE table_name

SET column1=value, column2=value2,...

WHERE some_column=some_value

Note! Notice the WHERE clause in the UPDATE syntax. The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

Example:

update CUSTOMER set AreaCode=46 where CustomerId=2

Before update:

Before update:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

After update:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	46	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

If you don't include the WHERE clause the result becomes:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	46	California	11111111
2	2	1001	Jackson	Smith	46	London	22222222
3	3	1002	Johnsen	John	46	London	33333333

→ So make sure to include the WHERE clause when using the UPDATE command!

DELETE

The DELETE statement is used to delete rows in a table.

Syntax:

```
DELETE FROM table_name  
WHERE some_column=some_value
```

Note! Notice the WHERE clause in the DELETE syntax. The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

Example:

delete from CUSTOMER where CustomerId=2

Before delete:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

After delete:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	3	1002	Johnsen	John	32	London	33333333

Delete All Rows:

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
```

Note! Make sure to do this only when you really mean it! You cannot UNDO this statement!

e) Set Operations

The set operations union, intersect, and except operate on relations and correspond to the relational algebra operations \cup , \cap , and $-$.

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions union all, intersect all and except all. Suppose a tuple occurs m times in r and n times in s , then, it occurs:

- $m + n$ times in r union all s
- $\min(m, n)$ times in r intersect all s
- $\max(0, m - n)$ times in r except all s

- Find all customers who have a loan, an account, or both:

(select customer-name from depositor) union (select customer-name from borrower)

- Find all customers who have both a loan and an account.

(select customer-name from depositor) intersect (select customer-name from borrower)

- Find all customers who have an account but no loan.

(select customer-name from depositor) except (select customer-name from borrower)

f) Aggregate Functions

These functions operate on the multiset of values of a column of a relation, and return a value

Aggregate function	Description
AVG	The AVG() aggregate function calculates the average of non-NULL values in a set.
COUNT	The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.
MAX	The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.
MIN	The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.
SUM	The SUM() aggregate function returns the summation of all non-NULL values a set.

AVG example

The following statement use the AVG () function to return the average list price of all products in the products table:

```
SELECT  AVG(list_price) avg_product_price
FROM    production.products;
```

The following shows the output:

avg_product_price
1520.591401

COUNT example

The following statement uses the COUNT () function to return the number of products whose price is greater than 500:

```
SELECT  COUNT(*) product_count
FROM    production.products
WHERE   list_price > 500;
```

The following shows the output:

product_count
213

MAX example

The following statement uses the MAX () function to return the highest list price of all products:

```
SELECT  MAX(list_price) max_list_price
FROM    production.products;
```

The following picture shows the output:

max_list_price
11999.99

MIN example

Similarly, the following statement uses the MIN () function to return the lowest list price of all products:

```
SELECT  MIN(list_price) min_list_price
FROM    production.products;
```

The output is:

min_list_price
89.99

SUM example

The following statement uses the SUM() function to calculate the total stock by product id in all warehouses:

```
SELECT  product_id,  SUM(quantity) stock_count
FROM    production.stocks
GROUP BY  product_id
ORDER BY  stock_count DESC;
```

Here is the output:

product_id	stock_count
188	86
64	82
109	79
196	79
61	78
182	77
166	77
219	75
142	75
252	75

g) NULL Values

It is possible for tuples to have a null value, denoted by null, for some of their attributes; null signifies an unknown value or that a value does not exist.

- The result of any arithmetic expression involving null is null.
- Roughly speaking, all comparisons involving null return false.

More precisely,

- Any comparison with null returns unknown
- (true or unknown) = true, (false or unknown) = unknown (unknown or unknown) = unknown, (true and unknown) = unknown, (false and unknown) = false, (unknown and unknown) = unknown
- Result of where clause predicate is treated as false if it evaluates to unknown
- “P is unknown” evaluates to true if predicate P evaluates to unknown
- Find all loan numbers which appear in the loan relation with null values for amount.

select loan-number from loan where amount is null

- Total all loan amounts

select sum (amount) from loan

Above statement ignores null amounts; result is null if there is no non-null amount.

- All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes.

h) Nested Queries

SQL provides a mechanism for the nesting of subqueries.

- A subquery is a select-from-where expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- Find all customers who have both an account and a loan at bank.

select distinct customer-name

from borrower

where customer-name in (select customer-name from depositor)

- Find all customers who have a loan at the bank but do not have an account at the bank.

```
select distinct customer-name  
from borrower  
where customer-name not in (select customer-name from depositor)
```

- Find all customers who have both an account and a loan at the Perryridge branch.

```
select distinct customer-name  
from borrower, loan  
where  
borrower.loan-number = loan.loan-number  
and  
branch-name = "Perryridge"  
and  
(branch-name, customer-name) in (select branch-name, customer-name from  
depositor, account where depositor.account-number = account.account-number)
```

- Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct T.branch-name  
from branch as T, branch as S  
where T.assets > S.assets and S.branch-city = "Brooklyn"
```

i) Views

Views are used to implement the security mechanism in SQL. Views are generally used to restrict the user from viewing certain columns and rows. Views display only the data specified in the query, so it shows only the data that is returned by the query defined during the creation of the view. The rest of the data is totally abstract from the end user.

Types of views in SQL

There are the following two types of views:

1. User-Defined Views
2. System-Defined Views

First we discuss the User-Defined Views.

User Define Views:

First we create two tables. First create a Employee_Details table for the basic info of an employee.

```
CREATE TABLE [dbo].[Employee_Details]
(
    [Emp_Id] [int] IDENTITY(1,1) NOT NULL,
    [Emp_Name] [nvarchar](50) NOT NULL,
    [Emp_City] [nvarchar](50) NOT NULL,
    [Emp_Salary] [int] NOT NULL,
    CONSTRAINT [PK_Employee_Details] PRIMARY KEY CLUSTERED
    (
        [Emp_Id] ASC
    )
    WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
)
ON [PRIMARY]
GO
```

Now insert some data into the table as in the following:

```
Insert Into Employee_Details Values('Pankaj','Alwar',25000)
Insert Into Employee_Details Values('Rahul','Jaipur',26000)
Insert Into Employee_Details Values('Rajan','Delhi',27000)
Insert Into Employee_Details Values('Sandeep','Alwar',28000)
Insert Into Employee_Details Values('Sanjeev','Jaipur',32000)
Insert Into Employee_Details Values('Narendra','Alwar',34000)
Insert Into Employee_Details Values('Neeraj','Delhi',29000)
Insert Into Employee_Details Values('Div','Jaipur',25000)
Insert Into Employee_Details Values('Tanuj','Alwar',22000)
Insert Into Employee_Details Values('Nitin','Jaipur',20000)
```

We create another table named Employee_Contact.

```
CREATE TABLE [dbo].[Employee_Contact]
(
    [Emp_Id] [int] NOT NULL,
    [MobileNo] [nvarchar](50) NOT NULL
) ON [PRIMARY]

GO

ALTER TABLE [dbo].[Employee_Contact] WITH CHECK ADD CONSTRAINT
[FK_Employee_Contact_Employee_Details] FOREIGN KEY([Emp_Id])
REFERENCES [dbo].[Employee_Details] ([Emp_Id])

GO

ALTER TABLE [dbo].[Employee_Contact] CHECK CONSTRAINT
[FK_Employee_Contact_Employee_Details]

GO
```


Now insert values in Employee_Contact

```
Insert Into Employee_Contact Values(1,'9813220191')
Insert Into Employee_Contact Values(2,'9813220192')
Insert Into Employee_Contact Values(3,'9813220193')
Insert Into Employee_Contact Values(4,'9813220194')
Insert Into Employee_Contact Values(5,'9813220195')
Insert Into Employee_Contact Values(6,'9813220196')
Insert Into Employee_Contact Values(7,'9813220197')
Insert Into Employee_Contact Values(8,'9813220198')
Insert Into Employee_Contact Values(9,'9813220199')
Insert Into Employee_Contact Values(10,'9813220135')
```

Now we start a detailed discussion of User Defined Views (UDVs).

Create SQL VIEW in SQL

```
CREATE VIEW view_name AS
SELECT columns
FROM tables
WHERE conditions;
```

Let us create some views.

Method 1: We can select all columns of a table. The following example demonstrates that:

```
Create View Employee_View1
```

```
as
```

```
select * from Employee_Details
```

Method 2: We can select specific columns of a table. The following example demonstrates that:

```
Create View Employee_View2
```

```
as
```

```
select Emp_Id,Emp_Name,Emp_City from Employee_Details
```

Method 3: We can select columns from a table with specific conditions. The following example demonstrates that:

```
Create View Employee_View3
```

```
as
```

```
select * from Employee_Details where Emp_Id>3
```

Method 4: We can create a view that will hold the columns of different tables. The following example demonstrates that:

```
Create View Employee_View4
```

```
as
```

```
select
```

```
Employee_Details.Emp_Id,Employee_Details.Emp_Name,Employee_Details.Emp_Salary,Employee_C  
ontact.MobileNo from Employee_Details
```

```
Left Outer Join
```

```
Employee_Contact
```

```
on
```

```
Employee_Details .Emp_Id= Employee_Contact.Emp_Id
```

```
Where Employee_Details.Emp_Id>2
```

Retrieve Data From View in SQL

This SQL CREATE VIEW example would create a virtual table based on the result set of the select statement. Now we can retrieve data from a view as follows:

```
Select * from Employee_View4
```

```
Select Emp_Id,Emp_Name,Emp_Salary from Employee_View4
```

The preceding query shows that we can select all the columns or some specific columns from a view.

Dropping a View in SQL

We can use the Drop command to drop a view. For example, to drop the view Employee_View3, we can use the following statement

```
Drop View Employee_View1
```

j) Embedded SQL

This is a method for combining data manipulation capabilities of SQL and computing power of any programming language. Then embedded statements are in line with the program source code of the host language. The code of embedded SQL is parsed by a pre-processor which is also embedded and is replaced by the host language called for the code library it is then compiled via the compiler of the host.

Two steps which define by SQL standards community, they are –

Module language defining which is formalization and then an embedded SQL standard derived from the module language. Most popular hosting language is C, it is called for example Pro*C in Oracle and Sybase database management systems and ECPG in the PostgreSQL database management system.

When you embed SQL with another language. The language that is embedded is known as host language and the SQL standard which defines the embedding of SQL is known as embedded SQL.

- The result of a query is made available to the program which is embedded as one tuple or record at a time
- For identification of this, we request to the preprocessor via EXEC SQL statement: EXEC SQL embedded SQL statement END-EXEC
- Its statements are declare cursor, fetch and open statements.
- It can execute the update, insert a delete statement

k) Stored Procedures and Functions

Stored Procedure

A Stored Procedure is nothing more than prepared SQL code that you save so you can reuse the code over and over again. So if you think about a query that you write over and over again, instead of having to write that query each time you would save it as a Stored Procedure and then just call the Stored Procedure to execute the SQL code that you saved as part of the Stored Procedure.

In addition to running the same SQL code over and over again you also have the ability to pass parameters to the Stored Procedure, so depending on what the need is, the Stored Procedure can act accordingly based on the parameter values that were passed.

Stored Procedures can also improve performance. Many tasks are implemented as a series of SQL statements. Conditional logic applied to the results of the first SQL statements determine which subsequent SQL statements are executed. If these SQL statements and conditional logic are written into a Stored Procedure, they become part of a single execution plan on the server. The results do not need to be returned to the client to have the conditional logic applied; all of the work is done on the server.

User Defined Functions

Like functions in programming languages, SQL User Defined Functions are routines that accept parameters, perform an action such as a complex calculation, and returns the result of that action as a value. The return value can either be a single scalar value or a result set.

Functions in programming languages are subroutines used to encapsulate frequently performed logic. Any code that must perform the logic incorporated in a function can call the function rather than having to repeat all of the function logic.

SQL supports two types of functions

- *Built-in functions*

Operate as defined in the Transact-SQL Reference and cannot be modified. The functions can be referenced only in Transact-SQL statements using the syntax defined in the Transact-SQL Reference.

- *User Defined Functions*

Allow you to define your own Transact-SQL functions using the CREATE FUNCTION statement. User Defined Functions use zero or more input parameters, and return a single value. Some User Defined Functions return a single, scalar data value, such as an int, char, or decimal value.

Benefits of User Defined Functions

- *They allow modular programming*

You can create the function once, store it in the database, and call it any number of times in your program. User Defined Functions can be modified independently of the program source code.

- *They allow faster execution*

Similar to Stored Procedures, Transact-SQL User Defined Functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times. CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.

- *They can reduce network traffic*

An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then be invoked in the WHERE clause to reduce the number of rows sent to the client.

Benefits of Stored Procedures

- *Precompiled execution*

SQL Server compiles each Stored Procedure once and then reutilizes the execution plan. This results in tremendous performance boosts when Stored Procedures are called repeatedly.

- *Reduced client/server traffic*

If network bandwidth is a concern in your environment then you'll be happy to learn that Stored Procedures can reduce long SQL queries to a single line that is transmitted over the wire.

- *Efficient reuse of code and programming abstraction*

Stored Procedures can be used by multiple users and client programs. If you utilize them in a planned manner then you'll find the development cycle requires less time.

- *Enhanced security controls*

You can grant users permission to execute a Stored Procedure independently of underlying table permissions.

1. Creating a hello world in a stored procedure in SQL vs a function

Let's create a simple "Hello world" in a stored procedure and a function to verify which one is easier to create.

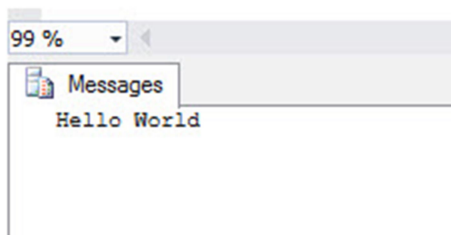
We will first create a simple stored procedure using the print statement in SSMS:

```
CREATE PROCEDURE HelloWorldprocedure  
  
AS  
  
PRINT 'Hello World'
```

Execute the code and then call the stored procedure in SQL:

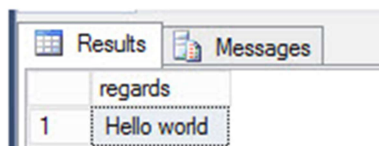
```
exec HelloWorldprocedure
```

If you execute the code, you will be able to see the "Hello World" message:



Now let's try to do the same with a function:

```
CREATE FUNCTION dbo.helloworldfunction()  
RETURNS varchar(20)  
AS  
BEGIN  
RETURN 'Hello world'  
END
```



If you compare the code, the function requires more code to do the same thing. The BEGIN and END blocks are mandatory in a function while the stored procedure do not require them if it is just one line. In a function, it is mandatory to use the RETURNS and RETURN arguments, whereas in a stored procedure is not necessary.

In few words, a stored procedure is more flexible to write any code that you want, while functions have a rigid structure and functionality.

2. Invoking a stored procedure in SQL vs invoking a function

You can invoke a stored procedure in different ways:

```
exec HelloWorldprocedure
```

```
execute HelloWorldprocedure
```

```
execute dbo.HelloWorldprocedure
```

```
HelloWorldprocedure
```

You can invoke using exec or execute and even you can invoke the stored procedure without the execute statement. You do not necessarily need to specify the schema name.

The functions are less flexible. You need to specify the schema to invoke it (which is a good practice to avoid conflicts with other object with the same name and different schema).

Let's call a function without the schema:

```
select helloworldfunction() as regards
```

The message displayed is the following:

Msg 195, Level 15, State 10, Line 20 'helloworldfunction' is not a recognized built-in function name

As you can see, the schema name is mandatory to invoke a function:

```
select dbo.helloworldfunction() as regards
```

3. Using variables in a stored procedure in SQL vs a function

We are going to convert Celsius degrees to Fahrenheit using stored procedures and functions to see the differences. Let's start with a stored procedure:

```
CREATE PROCEDURE CONVERTCELSIUSTOFAHRENHEIT
```

```
@celsius real
```

```
as
```

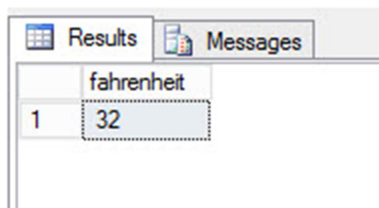
```
select @celsius*1.8+32 as Fahrenheit
```

Celsius is the input parameter and we are doing the calculations in the select statement to convert to Fahrenheit degrees.

If we invoke the stored procedure, we will verify the result converting 0 °C:

```
exec CONVERTCELSIUSTOFAHRENHEIT 0
```

The result will be 32 °F:



The screenshot shows a SQL Server Results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with one column named 'fahrenheit' and one row containing the value '32'.

	fahrenheit
1	32

Let's try to do the same with a function:

```
CREATE FUNCTION dbo.f_celsiustofahrenheit(@celcius real)
```

```
RETURNS real
```

```
AS
```

```
BEGIN
```

```
RETURN @celcius*1.8+32
```

```
END
```

You can call the function created in the following way:

```
select dbo.f_celsiustofahrenheit(0) as fahrenheit
```

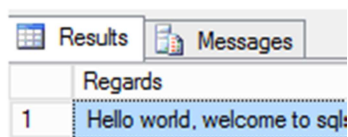
We are converting 0 °C to °F. As you can see, the code is very simple in both cases.

4. Reusability

The main advantage about a function is that it can be reused in code. For example, you can do the following:

```
select CONCAT(dbo.helloworldfunction(),', welcome to sql) Regards
```

In this example, we are concatenating the function of the example 1 with a string. The result is the following:



Results	
	Regards
1	Hello world, welcome to sql:

As you can see, you can easily concatenate a function with a string. To do something similar with a stored procedure in SQL, we will need an output variable in a stored procedure to concatenate the output variable with a string. Let's take a look to the stored procedure:

```
create procedure outputparam  
@paramout varchar(20) out  
as  
select @paramout='Hello world'
```

The procedure is assigning the Hello Word string to an output parameter. You can use the out or output word to specify that the parameter is an output parameter.

The code may be simple, but calling the procedure to use the output parameter to be concatenated is a little bit more complex than a function:

```
declare @message varchar(20)  
exec outputparam @paramout=@message out  
select @message as regards  
select CONCAT(@message,', welcome to sqlshack')
```

As you can see, you need to declare a new variable named @message or any other name of your preference. When you call the stored procedure, you need to specify that it is an outer parameter. An advantage of the stored procedures is that you can have several parameters while in functions, you can return just one variable (scalar function) or one table (table-valued functions).

5. Invoke functions/procedures inside functions/Stored procedures in SQL

Can we invoke stored procedures inside a function?

Let's take a look:

```
CREATE FUNCTION dbo.procedureinsidefunction()  
  
RETURNS varchar(22)  
  
AS  
  
BEGIN  
  
execute HelloWorldprocedure  
  
Declare @hellovar varchar(22)=' welcome to sqlshack'  
  
RETURN @hellovar  
  
END
```

The function will invoke the HelloWorldprocedure created in the section 1.

If we invoke the function, we will have the following message:

Msg 557, Level 16, State 2, Line 65 Only functions and some extended stored procedures can be executed from within a function.

As you can see, you cannot call a function from a stored procedure. Can you call a function from a procedure?

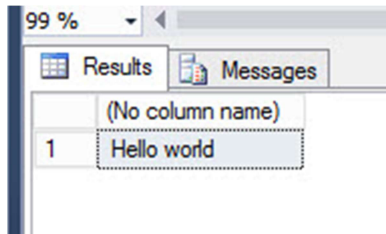
Here it is the procedure:

```
create procedure functioninsideprocedure  
  
as  
  
select dbo.helloworldfunction()
```

If we invoke the stored procedure in SQL, we will be able to check if it works or not:

`exec functioninsideprocedure`

The result displayed is the following:



As you can see, you can invoke functions inside a stored procedure and you cannot invoke a stored procedure inside a function.

You can invoke a function inside a function. The following code shows a simple example:

```
CREATE FUNCTION dbo.functioninsidefunction()  
RETURNS varchar(50)  
AS  
BEGIN  
RETURN dbo.helloworldfunction()  
END
```

We can call the function as usual:

```
select dbo.functioninsidefunction() as regards
```

Is it possible to call procedures inside other procedures?

Yes, you can. Here you have an example about it:

```
create procedure  
procedureinsideprocedure  
as  
  
execute dbo.HelloWorldprocedure
```

You can execute the procedure as usual:

```
exec dbo.procedureinsideprocedure
```

6. Conclusions

Stored procedures in SQL are easier to create and functions have a more rigid structure and support less clauses and functionality. By the other hand, you can easily use the function results in T-SQL. We show how to concatenate a function with a string. Manipulating results from a stored procedure is more complex.

In a scalar function, you can return only one variable and in a stored procedure multiple variables. However, to call the output variables in a stored procedure, it is necessary to declare variables outside the procedure to invoke it.

In addition, you cannot invoke procedures within a function. By the other hand, in a procedure you can invoke functions and stored procedures.

7. Differences between Stored Procedure and User Defined Function in SQL

User Defined Function	Stored Procedure
Function must return a value.	Stored Procedure may or not return values.
Will allow only Select statements, it will not allow us to use DML statements.	Can have select statements as well as DML statements such as insert, update, delete and so on
It will allow only input parameters, doesn't support output parameters.	It can have both input and output parameters.
It will not allow us to use try-catch blocks.	For exception handling we can use try catch blocks.
Transactions are not allowed within functions.	Can use transactions within Stored Procedures.
We can use only table variables, it will not allow using temporary tables.	Can use both table variables as well as temporary table in it.
Stored Procedures can't be called from a function.	Stored Procedures can call functions.
Functions can be called from a select statement.	Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure.
A UDF can be used in join clause as a result set.	Procedures can't be used in Join clause

I) Dynamic SQL features

SQL are SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application. **Dynamic SQL** is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries.

Dynamic SQL is a programming technique that enables you to build SQL statements dynamically at runtime. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation.

Using Static SQL has a benefit which is the optimization of the statement that results an application with high performance as it offers a good flexibility better than Dynamic SQL, and since access plans for dynamic statements are generated at run-time so they must be prepared in the application, and this is something you will never look at in the static SQL, but these are not the only differences between them, so we can say that dynamic SQL has only one advantage over static statements which can be clearly noticed once the application

is edited or upgraded, so with Dynamic statements there's no need for pre-compilation or re-building as long as the access plans are generated at run-time, whereas static statements require regeneration of access plans if they were modified, in addition to the fact that Dynamic SQL requires more permissions, it also might be a way to execute unauthorized code, we don't know what kind of users we'll have, so for security it can be **dangerous** if the programmer didn't handle it.

Creating a dynamic SQL is simple, you just need to make it a string as follows:

```
'SELECT * FROM production.products';
```

To execute a dynamic SQL statement, you call the stored procedure `sp_executesql` as shown in the following statement:

```
EXEC sp_executesql N'SELECT * FROM production.products';
```

Because the `sp_executesql` accepts the dynamic SQL as a Unicode string, you need to prefix it with an `N`.

Though this dynamic SQL is not very useful, it illustrates a dynamic SQL very well.

Using dynamic SQL to query from any table example

First, declare two variables, `@table` for holding the name of the table from which you want to query and `@sql` for holding the dynamic SQL.

```
DECLARE
    @table NVARCHAR(128),
    @sql NVARCHAR(MAX);
```

Second, set the value of the `@table` variable to `production.products`.

```
SET @table = N'production.products';
```

Third, construct the dynamic SQL by concatenating the `SELECT` statement with the table name parameter:

```
SET @sql = N'SELECT * FROM ' + @table;
```

Fourth, call the `sp_executesql` stored procedure by passing the `@sql` parameter.

```
EXEC sp_executesql @sql;
```

Putting it all together:

```
DECLARE
    @table NVARCHAR(128),
    @sql NVARCHAR(MAX);

SET @table = N'production.products';

SET @sql = N'SELECT * FROM ' + @table;
```

```
EXEC sp_executesql @sql;
```

The code block above produces the exact result set as the following statement:

```
SELECT * FROM production.products;
```

To query data from another table, you change the value of the @table variable. However, it's more practical if we wrap the above T-SQL block in a stored procedure.

Dynamic SQL and stored procedures

This stored procedure accepts any table and returns the result set from a specified table by using the dynamic SQL:

```
CREATE PROC usp_query (
    @table NVARCHAR(128)
)
AS
BEGIN

    DECLARE @sql NVARCHAR(MAX);
    -- construct SQL
    SET @sql = N'SELECT * FROM ' + @table;
    -- execute the SQL
    EXEC sp_executesql @sql;
```

```
END;
```

The following statement calls the usp_query stored procedure to return all rows from the production.brands table:

```
EXEC usp_query 'production.brands';
```

This stored procedure returns the top 10 rows from a table by the values of a specified column:

```
CREATE OR ALTER PROC usp_query_topn(
    @table NVARCHAR(128),
```

```
@topN INT,  
@byColumn NVARCHAR(128)  
)  
AS  
BEGIN  
    DECLARE  
        @sql NVARCHAR(MAX),  
        @topNStr NVARCHAR(MAX);  
  
    SET @topNStr = CAST(@topN as nvarchar(max));  
  
    -- construct SQL  
    SET @sql = N'SELECT TOP ' + @topNStr +  
        ' * FROM ' + @table +  
        ' ORDER BY ' + @byColumn + ' DESC';  
    -- execute the SQL  
    EXEC sp_executesql @sql;  
  
END;
```

For example, you can get the top 10 most expensive products from the production.products table:

```
EXEC usp_query_topn  
    'production.products',  
    10,  
    'list_price';
```

This statement returns the top 10 products with the highest quantity in stock:

```
EXEC usp_query_topn  
    'production.tocks',  
    10,  
    'quantity';
```


Dynamic Statement Execution (Execute Immediate)

The Execute Immediate statement provides the simplest form of dynamic SQL. This statement passes the text of SQL statements to DBMS and asks the DBMS to execute the SQL statements immediately.

For using the statement our program goes through the following steps.

1. The program constructs a SQL statement as a string of text in one of its data areas (called a buffer).
2. The program passes the SQL statements to the DBMS with the EXECUTE IMMEDIATE statement.
3. The DBMS executes the statement and sets the SQL CODE/SQL STATE values to flag the finishing status same like if the statement had been hard coded using static SQL.

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

Static (Embedded) SQL	Dynamic (Interactive) SQL
In Static SQL, how database will be accessed is predetermined in the embedded SQL statement.	In Dynamic SQL, how database will be accessed is determined at run time.
It is more swift and efficient.	It is less swift and efficient.
SQL statements are compiled at compile time.	SQL statements are compiled at run time.
Parsing, Validation, Optimization and Generation of application plan are done at compile time.	Parsing, Validation, Optimization and Generation of application plan are done at run time.
It is generally used for situations where data is distributed uniformly.	It is generally used for situations where data is distributed non uniformly.
EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are not used.	EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are used.
It is less flexible.	It is more flexible.

Limitation of Dynamic SQL:

We cannot use some of the SQL statements Dynamically.
Performance of these statements is poor as compared to Static SQL.

Limitations of Static SQL:

They do not change at runtime thus are hard-coded into applications.