- Introduction to relational databases and SQL,
- Introduction to ADO,
- Accessing data with ADO,
- Controlling transactions in ASP.

Introduction to relational databases and SQL,

Relational database: collection of *tables* (also called *relations*)

Each table is known as a relation, which contains one or more data category columns. Each table record (or row) contains a unique data instance defined for a corresponding column category. One or more data or record characteristics relate to one or many records to form functional dependencies. These are classified as follows:

- One to One: One table record relates to another record in another table.
- One to Many: One table record relates to many records in another table.
- Many to One: More than one table record relates to another table record.
- Many to Many: More than one table record relates to more than one record in another table.

- **Table**: ***tuples or records***
    - Collection of rows (also called tuples or records).
    - Each row in a table contains a set of columns (also called fields or attributes).
    - Each column has a type:
        - String: VARCHAR(20)
        - Integer: INTEGER
        - Floating-point: FLOAT, DOUBLE
        - Date/time DATE, TIME, DATETIME
        - Several others...
    - Primary key: provides a unique identifier for each row (need not be present, but almost always is in practice).
    - Schema: the structure of the database, including for each table
        - The table name
        - The names and types of its columns
        - Various optional additional information (constraints, etc.)
        - The number of rows in a table is not part of the schema
- **SQL**: a language for creating and manipulating relational databases
    - Initially created at IBM as part of System-R
    - Implemented with modifications in numerous products: Oracle, Sybase, DB-2, SQL Server, MySQL
    - Vendor versions are not completely compatible
    - Has been (partially) standardized
    - Examples that follow use MySQL syntax
- Create a table for the students:

```
CREATE TABLE students (
  id INT AUTO_INCREMENT,
  name VARCHAR(30),
  birth DATE,
```

```
gpa FLOAT,
grad INT,
PRIMARY KEY(id));
```

- Add rows to the students table:

```
INSERT INTO students(name, birth, gpa, grad)
   VALUES ('Anderson', '1987-10-22', 3.9, 2009);
INSERT INTO students(name, birth, gpa, grad)
   VALUES ('Jones', '1990-4-16', 2.4, 2012);
INSERT INTO students(name, birth, gpa, grad)
  VALUES ('Hernandez', '1989-8-12', 3.1, 2011);
INSERT INTO students(name, birth, gpa, grad)
  VALUES ('Chen', '1990-2-4', 3.2, 2011);
```

- Delete row(s):

```
DELETE FROM students WHERE name='Anderson';
```

- Delete table:

```
DROP TABLE students;
```

- Queries: the strength of relational databases
  - Lots of ways to extract information
  - You specify what you want
  - The database system figures out how to get it efficiently
  - Refer to data by *contents*, not just name
  - Show entire contents of a table:

```
SELECT * FROM students;
+----+-----------+------------+------+------+
| id | name      | birth      | gpa  | grad |
+----+-----------+------------+------+------+
|  1 | Anderson  | 1987-10-22 |  3.9 | 2009 |
|  2 | Jones     | 1990-04-16 |  2.4 | 2012 |
|  3 | Hernandez | 1989-08-12 |  3.1 | 2011 |
|  4 | Chen      | 1990-02-04 |  3.2 | 2011 |
+----+-----------+------------+------+------+
```

  - Show just a few columns from a table:

```
SELECT name, gpa FROM students;
+-----------+------+
| name      | gpa  |
+-----------+------+
| Anderson  |  3.9 |
| Jones     |  2.4 |
| Hernandez |  3.1 |
```

```
| Chen      |  3.2 |
+-----------+------+
```

- Filtering: only display a subset of the rows:

```
SELECT name, gpa
   FROM students
   WHERE gpa > 3.0;
+-----------+------+
| name      | gpa  |
+-----------+------+
| Anderson  |  3.9 |
| Hernandez |  3.1 |
| Chen      |  3.2 |
+-----------+------+
```

- Sorting:

```
SELECT gpa, name, grad
   FROM students
   WHERE gpa > 3.0
   ORDER BY gpa DESC;
+------+-----------+------+
| gpa  | name      | grad |
+------+-----------+------+
|  3.9 | Anderson  | 2009 |
|  3.2 | Chen      | 2011 |
|  3.1 | Hernandez | 2011 |
+------+-----------+------+
```

- Updates:

```
UPDATE students
   SET gpa = 2.6, grad = 2013
   WHERE id = 2;
```

- Joins: how to manage relationships between tables?
  - Join: a query that merges the contents of 2 or more tables, displays information from the results.
  - Can produce the equivalent of a linked list in a programming language, and many other effects.
- Join example: ***many-to-one relationship***
  - Students have advisors; introduce new table describing faculty.

```
+----+----------+-----------+
| id | name     | title     |
+----+----------+-----------+
|  1 | Fujimura | assocprof |
```

```
| 2 | Bolosky  | prof     |
+----+---------+----------+
```

- Add new column advisor_id to the students table. This is a *foreign key*.

```
+----+-----------+------------+------+------+------------+
| id | name      | birth      | gpa  | grad | advisor_id |
+----+-----------+------------+------+------+------------+
|  1 | Anderson  | 1987-10-22 |  3.9 | 2009 |          2 |
|  2 | Jones     | 1990-04-16 |  2.4 | 2012 |          1 |
|  3 | Hernandez | 1989-08-12 |  3.1 | 2011 |          1 |
|  4 | Chen      | 1990-02-04 |  3.2 | 2011 |          1 |
+----+-----------+------------+------+------+------------+
```

- Example query:

```
SELECT s.name, s.gpa
   FROM students s, advisors p
   WHERE s.advisor_id = p.id AND p.student = 'Fujimura';
+-----------+------+
| name      | gpa  |
+-----------+------+
| Jones     |  2.4 |
| Hernandez |  3.1 |
| Chen      |  3.2 |
+-----------+------+
```

- A join creates the cross-product of 2 or more tables
  - Potentially very expensive!
  - In practice, optimized carefully by the database system.

- Join example: ***many-to-many relationship***
  - Courses: students take many courses, courses have many students
  - Add new table describing courses:

```
+----+--------+-----------------+-------------+
| id | number | name            | quarter     |
+----+--------+-----------------+-------------+
|  1 | CS142  | Web stuff       | Winter 2009 |
|  2 | ART101 | Finger painting | Fall 2008   |
|  3 | ART101 | Finger painting | Winter 2009 |
|  4 | PE204  | Mud wrestling   | Winter 2009 |
+----+--------+-----------------+-------------+
```

  - Create a *join table* courses_students describing which students took which courses.

```
+-----------+------------+
| course_id | student_id |
```

```
+-----------+------------+
|         1 |          1 |
|         3 |          1 |
|         4 |          1 |
|         1 |          2 |
|         2 |          2 |
|         1 |          3 |
|         2 |          4 |
|         4 |          4 |
+-----------+------------+
```

- Find all students who took a particular course:

```
SELECT s.name, c.quarter
  FROM students s, courses c, courses_students cs
  WHERE c.id = cs.course_id AND s.id = cs.student_id
  AND c.number = 'ART101';
+----------+-------------+
| name     | quarter     |
+----------+-------------+
| Jones    | Fall 2008   |
| Chen     | Fall 2008   |
| Anderson | Winter 2009 |
+----------+-------------+
```

Comparison of ADO and ADO.NET

ADO	ADO.Net
ADO is base on **COM : Component Object Modelling** based.	ADO.Net is based on **CLR : Common Language Runtime** based.
ADO stores data in **binary** format.	ADO.Net stores data in **XML** format i.e. parsing of data.
ADO can't be integrated with XML because ADO have limited access of XML.	ADO.Net can be integrated with XML as having robust support of XML.
In ADO, data is provided by **RecordSet**.	In ADO.Net data is provided by **DataSet** or **DataAdapter**.
ADO is **connection** oriented means it requires continuous active connection.	ADO.Net is **disconnected**, does not need continuous connection.
ADO gives rows as single table view, it scans sequentially the rows using **MoveNext**method.	ADO.Net gives rows as collections so you can access any record and also can go through a table via loop.
In ADO, You can create only **Client** side cursor.	In ADO.Net, You can create both **Client & Server** side cursor.

Using a single connection instance, ADO can not handle multiple transactions.	Using a single connection instance, ADO.Net can handle multiple transactions.

Introduction to ADO,

- **ActiveX Data Objects** (**ADO**) comprises a set of Component Object Model (COM) objects for accessing data sources.
- A part of MDAC (Microsoft Data Access Components), it provides a middleware layer between programming languages and OLE DB (a means of accessing data stores, whether databases or not, in a uniform manner).
- ADO allows a developer to write programs that access data without knowing how the database is implemented; developers must be aware of the database for connection only.
- No knowledge of SQL is required to access a database when using ADO, although one can use ADO to execute SQL commands directly (with the disadvantage of introducing a dependency upon the type of database used).

In Short What is ADO?

- ADO is a Microsoft technology
- ADO stands for **A**ctiveX **D**ata **O**bjects
- ADO is a Microsoft Active-X component
- ADO is automatically installed with Microsoft IIS
- ADO is a programming interface to access data in a database

Steps are required in order to be able to access and manipulate data using ADO :

1. Create a connection object to connect to the database.
2. Create a recordset object in order to receive data in.
3. Open the connection
4. Populate the recordset by opening it and passing the desired table name or SQL statement as a parameter to open function.
5. Do all the desired searching/processing on the fetched data.
6. Commit the changes you made to the data (if any) by using Update or UpdateBatch methods.
7. Close the recordset
8. Close the connection

NOTE:

a) Download "**mysql-connector-odbc-5.3.11-winx64**" for 64bit OS or "mysql-connector-odbc-5.3.11-winx86" for 32bit OS and **Install**
b) If there is need of "Visual C++ Redistributable", download from Microsoft.com and install
c) Go to "Control Panel\All Control Panel Items\Administrative Tools" and Click "**ODBC Data Sources (64-bit)**" for 64bit OS or "ODBC Data Sources (32-bit)" for 32bit OS
d) Click on Tab "**System DSN**" and Click on Add
e) Double click on "**MySQL ODBC 5.3 Unicode Driver**"
f) Datasource Name=test, TCP/IP Server=localhost, User="root, Database=csit and Click on Finish

Major ADO Objects

a) Connection Objects

Before a database can be accessed from a web page, a database connection has to be established.

Syntax:

```
set objConnection=Server.CreateObject("ADODB.connection")
```

b) Recordset Objects

To be able to read database data, the data must first be loaded into a recordset.

Syntax:

```
set objRecordset=Server.CreateObject("ADODB.recordset")
```

c) Command Objects

Is used to execute the query against the database. The query performs actions like create, insert, update, delete, select etc

Syntax:

```
set objCommand=Server.CreateObject("ADODB.command")
```

Connection Object

- The ADO Connection Object is used to create an open connection to a data source. Through this connection, you can access and manipulate a database.
- If you want to access a database multiple times, you should establish a connection using the Connection object. You can also make a connection to a database by passing a connection string via a Command or Recordset object. However, this type of connection is only good for one specific, single query.

```
<%
	Set con = Server.CreateObject("ADODB.Connection")
	dim conStr
	conStr="DRIVER={MySQL ODBC 5.3 Unicode Driver};SERVER=127.0.0.1; DATABASE=bsccsit; User=root;Password=; OPTION=3;"
	con.Open conStr

%>
```

Properties

Property	Description
Attributes	Sets or returns the attributes of a Connection object
CommandTimeout	Sets or returns the number of seconds to wait while attempting to execute a command
ConnectionString	Sets or returns the details used to create a connection to a data source
ConnectionTimeout	Sets or returns the number of seconds to wait for a connection to open
CursorLocation	Sets or returns the location of the cursor service
DefaultDatabase	Sets or returns the default database name
IsolationLevel	Sets or returns the isolation level
Mode	Sets or returns the provider access permission
Provider	Sets or returns the provider name
State	Returns a value describing if the connection is open or closed
Version	Returns the ADO version number

Methods

Method	Description
BeginTrans	Begins a new transaction
Cancel	Cancels an execution
Close	Closes a connection
CommitTrans	Saves any changes and ends the current transaction

Execute	Executes a query, statement, procedure or provider specific text
Open	Opens a connection
OpenSchema	Returns schema information from the provider about the data source
RollbackTrans	Cancels any changes in the current transaction and ends the transaction

Events

Note: You cannot handle events using VBScript or JScript (only Visual Basic, Visual C++, and Visual J++ languages can handle events).

Event	Description
BeginTransComplete	Triggered after the BeginTrans operation
CommitTransComplete	Triggered after the CommitTrans operation
ConnectComplete	Triggered after a connection starts
Disconnect	Triggered after a connection ends
ExecuteComplete	Triggered after a command has finished executing
InfoMessage	Triggered if a warning occurs during a ConnectionEvent operation
RollbackTransComplete	Triggered after the RollbackTrans operation
WillConnect	Triggered before a connection starts
WillExecute	Triggered before a command is executed

Collections

Collection	Description
Errors	Contains all the Error objects of the Connection object
Properties	Contains all the Property objects of the Connection object

Command Object

- The ADO Command object is used to execute a single query against a database. The query can perform actions like creating, adding, retrieving, deleting or updating records.
- If the query is used to retrieve data, the data will be returned as a RecordSet object. This means that the retrieved data can be manipulated by properties, collections, methods, and events of the Recordset object.
- The major feature of the Command object is the ability to use stored queries and procedures with parameters.

```
<%
    Set cmd= Server.CreateObject("ADODB.Command")
    Set cmd.ActiveConnection = con
    cmd.CommandText = "INSERT INTO Student(FirstName,LastName) VALUES ('jeewan','rai')"
    cmd.Execute
%>
```

Properties

Property	Description
ActiveConnection	Sets or returns a definition for a connection if the connection is closed, or the current Connection object if the connection is open
CommandText	Sets or returns a provider command

CommandTimeout	Sets or returns the number of seconds to wait while attempting to execute a command
CommandType	Sets or returns the type of a Command object
Name	Sets or returns the name of a Command object
Prepared	Sets or returns a Boolean value that, if set to True, indicates that the command should save a prepared version of the query before the first execution
State	Returns a value that describes if the Command object is open, closed, connecting, executing or retrieving data

Methods

Method	Description
Cancel	Cancels an execution of a method
CreateParameter	Creates a new Parameter object
Execute	Executes the query, SQL statement or procedure in the CommandText property

Collections

Collection	**Description**
Parameters	Contains all the Parameter objects of a Command Object
Properties	Contains all the Property objects of a Command Object

Accessing Data with ADO

Creating a Connection String

The first step in creating a Web data application is to provide a way for ADO to locate and identify your data source. This is accomplished by means of a *connection string*, a series of semicolon delimited arguments that define parameters such as the data source provider and the location of the data source. ADO uses the connection string to identify the OLE DB *provider* and to direct the provider to the data source. The provider is a component that represents the data source and exposes information to your application in the form of rowsets.

The following table lists OLE DB connection strings for several common data sources:

Data source	**OLE DB connection string**
Access	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=*physical path to .mdb file*
SQL Server	Provider=SQLOLEDB.1;Data Source=*path to database on server*
Oracle	Provider=MSDAORA.1;Data Source=*path to database on server*

Data source driver	**ODBC connection string**
Access	Driver={Microsoft Access Driver (*.mdb)};DBQ=*physical path to .mdb file*
SQL Server	DRIVER={SQL Server};SERVER=*path to server*

Oracle	DRIVER={Microsoft ODBC for Oracle};SERVER=*path to server*
MySQL	DRIVER={MySQL ODBC 5.3 Unicode Driver};SERVER=localhost; DATABASE=bsccsit; UID=root;PASSWORD=;Port=8306; OPTION=3"

Connecting to a Data Source

MySQL Database Connection:-

1. Download Connector/ODBC from following given URL
   https://dev.mysql.com/downloads/connector/odbc/5.3.html
2. Complete Installation
3. Open Administrative Tools
4. Click on ODBC Data Sources
5. Tab on System DSN and Click on Add
6. Select MySQL ODBC 5.3 Unicode Driver and Click on Finish button OK

ADO provides the **Connection** object for establishing and managing connections between your applications and OLE DB compliant data sources or ODBC compliant databases. The **Connection** object features properties and methods you can use to open and close database connections, and to issue queries for updating information.

To establish a database connection, you first create an instance of the **Connection** object. For example, the following script instantiates the **Connection** object and proceeds to open a connection:

```
<%
 'Create a connection object.
 Set con = Server.CreateObject("ADODB.Connection")
 'Open a connection using the OLE DB connection string.
 con.Open "DRIVER={MySQL ODBC 5.3 Unicode Driver};SERVER=127.0.0.1; DATABASE=bsccsit; User=root;Password=; OPTION=3;"
%>
```

Executing SQL Queries with the Connection Object

With the **Execute** method of the **Connection** object you can issue commands to the data sources, such as Structured Query Language (SQL) queries. (SQL, an industry standard language for communicating with databases, defines commands for retrieving and updating information.) The **Execute**method can accept parameters that specify the command (or query), the number of data records affected, and the type of command being used.

The following script uses the **Execute** method to issue a query in the form of a SQL **INSERT** command, which inserts data into a specific database table. In this case, the script block inserts the name *Jose Lugo* into a database table named *Customers*.

```
<%
 'Define the OLE DB connection string.
 strConnectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Employees.mdb"

 'Instantiate the Connection object and open a database connection.
 Set con = Server.CreateObject("ADODB.Connection")
 con.Open strConnectionString

 'Define SQL SELECT statement.
```

```
 strSQL = "INSERT INTO Customers (FirstName, LastName) VALUES ('Jose','Lugo')"

 'Use the Execute method to issue a SQL query to database.
 con.Execute
%>
```

Retrieving a Record Set

Successful Web data applications employ both the **Connection** object, to establish a link, and the **Recordset** object, to manipulate returned data. By combining the specialized functions of both objects you can develop database applications to carry out almost any data handling task. For example, the following server-side script uses the **Recordset** object to execute a SQL **SELECT** command. The **SELECT** command retrieves a specific set of information based on query constraints. The query also contains a SQL **WHERE** clause, used to narrow down a query to a specific criterion. In this example, the **WHERE** clause limits the query to all records containing the last name *Smith* from the *Customers* database table.

```
<%
 'Establish a connection with data source.
 strConnectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Employees.mdb"
 Set con = Server.CreateObject("ADODB.Connection")
 con.Open strConnectionString

 'Instantiate a Recordset object.
 Set rstCustomers = Server.CreateObject("ADODB.Recordset")

 'Open a recordset using the Open method
 'and use the connection established by the Connection object.
 strSQL = "SELECT FirstName, LastName FROM Customers WHERE LastName = 'Smith' "
 rstCustomers.Open  strSQL, con

 'Cycle through record set and display the results
 'and increment record position with MoveNext method.
  Set objFirstName = rstCustomers("FirstName")
  Set objLastName = rstCustomers("LastName")
  Do Until rstCustomers.EOF
    Response.Write objFirstName & " " & objLastName & "<BR>"
    rstCustomers.MoveNext
  Loop

%>
```

Improving Queries with the Command Object

With the ADO **Command** object you can execute queries in the same way as queries executed with the **Connection** and object you can execute queries in the same way as queries executed with the Connection and **Recordset** object, except that with the **Command** object you can prepare, or compile, your query on the database source and then repeatedly reissue the query with a different set of values. The benefit of compiling queries in this manner is that you can vastly reduce the time required to reissue modifications to an existing query. In addition, you can leave your SQL queries partially undefined, with the option of altering portions of your queries just prior to execution.

The **Command** object's **Parameters** collection saves you the trouble of reconstructing your query each time you want to reissue your query. For example, if you need to regularly update supply and cost information in your Web-based inventory system, you can predefine your query in the following way:

```
<%
  'Open a connection using Connection object. Notice that the Command object
  'does not have an Open method for establishing a connection.
  strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Inventory.mdb"
  Set con = Server.CreateObject("ADODB.Connection")
  con.Open strConnectionString

  'Instantiate Command object; use ActiveConnection property to attach
  'connection to Command object.
  Set cmd= Server.CreateObject("ADODB.Command")
  Set cmd.ActiveConnection = con

  'Define SQL query.
  cmd.CommandText = "INSERT INTO Inventory (Material, Quantity) VALUES (?, ?)"

  'Save a prepared (or pre-compiled) version of the query specified in CommandText
  'property before a Command object's first execution.
  cmd.Prepared = True

  'Define query parameter configuration information.
  cmd.Parameters.Append
      cmd.CreateParameter("material_type",adVarChar, ,255 )
  cmd.Parameters.Append
      cmd.CreateParameter("quantity",adVarChar, ,255 )

  'Define and execute first insert.
  cmd ("material_type") = "light bulbs"
  cmd ("quantity") = "40"
  cmd.Execute ,,adCmdText + adExecuteNoRecords

  'Define and execute second insert.
  cmd ("material_type") = "fuses"
  cmd ("quantity") = "600"
  cmd.Execute ,,adCmdText + adExecuteNoRecords
 %>
```

Combining HTML Forms and Database Access

Web pages containing HTML forms can enable users to remotely query a database and retrieve specific information. With ADO you can create surprisingly simple scripts that collect user form information, create a custom database query, and return information to the user. Using the ASP **Request** object, you can retrieve information entered into an HTML form and incorporate this information into your SQL statements. For example, the following script block inserts information supplied by an HTML form into a table. The script collects the user information with the **Request** object 's **Form** collection.

```
<%
 'Open a connection using Connection object. The Command object
 'does not have an Open method for establishing a connection.
  strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\CompanyCatalog\Seeds.mdb"
  Set con = Server.CreateObject("ADODB.Connection")
  con.Open strConnectionString
```

```
 'Instantiate Command object
 'and  use ActiveConnection property to attach
 'connection to Command object.
 Set cmd= Server.CreateObject("ADODB.Command")
 Set cmd.ActiveConnection = con

 'Define SQL query.
 cmd.CommandText = "INSERT INTO MySeedsTable (Type) VALUES (?)"

 'Define query parameter configuration information.
 cmd.Parameters.Append cmd.CreateParameter("type",adVarChar, ,255)

 'Assign input value and execute update.
 cmd("type") = Request.Form("SeedType")
 cmd.Execute
%>
```

Display the Field Names and Field Values

We have a database named "Northwind" and we want to display the data from the "Customers" table (remember to save the file with an .asp extension):

```
<html>
<body>

<%
set conn=Server.CreateObject("ADODB.Connection")
conn.Provider="Microsoft.Jet.OLEDB.4.0"
conn.Open "c:/webdata/northwind.mdb"

set rs = Server.CreateObject("ADODB.recordset")
rs.Open "SELECT * FROM Customers", conn

do until rs.EOF
for each x in rs.Fields
Response.Write(x.name)
Response.Write(" = ")
Response.Write(x.value & "<br>")
next
Response.Write("<br>")
rs.MoveNext
loop

rs.close
conn.close
%>

</body>
</html>
```

Controlling transactions in ASP.

When we perform some database operations in such a way that either all the database operations are successful or all of them fail. This would result in the amount information being same once the transaction is complete or it fails.
To illustrate the above process, let say I have two account holders, one person is trying to transfer some money to other person. From the database perspective this operation consist of two sub-operations i.e.

- Debiting the first account by specified amount.
- Secondly, crediting the second account with required amount.

Properties of Transaction

- **Atomic**: Atomic means that all the statements (SQL statement or operations) that are a part of the transaction should work as atomic operation i.e. either all are successful or all should fail.
- **Consistent**: This means that in case the atomic transaction success, the database should be in a state that reflect changes. If the transaction fails then database should be exactly like it was when the transaction started.
- **Isolated**: If more than one transactions are in process then each of these transactions should work independently and should not effect the other transactions.
- **Durable**: Durability means that once the transaction is committed, the changes should be permanent i.e. these changes will get saved in database and should persist no matter what(like power failure or something).

These 3 methods is used with the Connection object to save or cancel changes made to the data source.
Note: Not all providers support transactions.
Note: These 3 methods are not available on a client-side Connection object.

a) BeginTrans
- The BeginTrans method starts a new transaction.
- This method can also be used to return a long value that is the level of nested transactions. A top level transaction has a return value of 1. Each additional level increments by one.

b) CommitTrans
- The CommitTrans method saves all changes made since the last BeginTrans method call, and ends the current transaction.
- Since transactions can be nested, all lower-level transactions must be resolved before you can resolve higher-level transactions.

c) RollbackTrans
- The RollbackTrans method cancels all changes made since the last BeginTrans method call, and ends the transaction.
- Since transactions can be nested, all lower-level transactions must be resolved before you can resolve higher-level transactions.

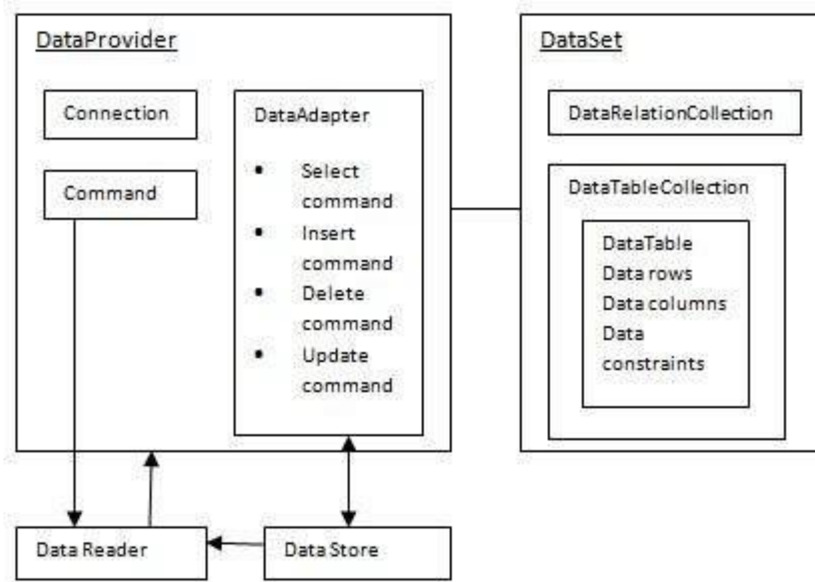
Syntax

```
level=objconn.BeginTrans()
objconn.BeginTrans
objconn.CommitTrans
objconn.RollbackTrans
```

ADO.NET

ADO.NET provides a bridge between the front end controls and the back end database. The ADO.NET objects encapsulate all the data access operations and the controls interact with these objects to display data, thus hiding the details of movement of data.

The following figure shows the ADO.NET objects at a glance:



The DataSet Class

The dataset represents a subset of the database. It does not have a continuous connection to the database. To update the database a reconnection is required. The DataSet contains DataTable objects and DataRelation objects. The DataRelation objects represent the relationship between two tables.

Following table shows some important properties of the DataSet class:

Properties	Description
CaseSensitive	Indicates whether string comparisons within the data tables are case-sensitive.
Container	Gets the container for the component.
DataSetName	Gets or sets the name of the current data set.
DefaultViewManager	Returns a view of data in the data set.
DesignMode	Indicates whether the component is currently in design mode.
EnforceConstraints	Indicates whether constraint rules are followed when attempting any update operation.
Events	Gets the list of event handlers that are attached to this component.
ExtendedProperties	Gets the collection of customized user information associated with the DataSet.
HasErrors	Indicates if there are any errors.
IsInitialized	Indicates whether the DataSet is initialized.

Locale	Gets or sets the locale information used to compare strings within the table.
Namespace	Gets or sets the namespace of the DataSet.
Prefix	Gets or sets an XML prefix that aliases the namespace of the DataSet.
Relations	Returns the collection of DataRelation objects.
Tables	Returns the collection of DataTable objects.

The following table shows some important methods of the DataSet class:

Methods	Description
AcceptChanges	Accepts all changes made since the DataSet was loaded or this method was called.
BeginInit	Begins the initialization of the DataSet. The initialization occurs at run time.
Clear	Clears data.
Clone	Copies the structure of the DataSet, including all DataTable schemas, relations, and constraints. Does not copy any data.
Copy	Copies both structure and data.
CreateDataReader()	Returns a DataTableReader with one result set per DataTable, in the same sequence as the tables appear in the Tables collection.
CreateDataReader(DataTable[])	Returns a DataTableReader with one result set per DataTable.
EndInit	Ends the initialization of the data set.
Equals(Object)	Determines whether the specified Object is equal to the current Object.
Finalize	Free resources and perform other cleanups.
GetChanges	Returns a copy of the DataSet with all changes made since it was loaded or the AcceptChanges method was called.
GetChanges(DataRowState)	Gets a copy of DataSet with all changes made since it was loaded or the AcceptChanges method was called, filtered by DataRowState.
GetDataSetSchema	Gets a copy of XmlSchemaSet for the DataSet.

GetObjectData	Populates a serialization information object with the data needed to serialize the DataSet.
GetType	Gets the type of the current instance.
GetXML	Returns the XML representation of the data.
GetXMLSchema	Returns the XSD schema for the XML representation of the data.
HasChanges()	Gets a value indicating whether the DataSet has changes, including new, deleted, or modified rows.
HasChanges(DataRowState)	Gets a value indicating whether the DataSet has changes, including new, deleted, or modified rows, filtered by DataRowState.
IsBinarySerialized	Inspects the format of the serialized representation of the DataSet.
Load(IDataReader, LoadOption, DataTable[])	Fills a DataSet with values from a data source using the supplied IDataReader, using an array of DataTable instances to supply the schema and namespace information.
Load(IDataReader, LoadOption, String[])	Fills a DataSet with values from a data source using the supplied IDataReader, using an array of strings to supply the names for the tables within the DataSet.
Merge()	Merges the data with data from another DataSet. This method has different overloaded forms.
ReadXML()	Reads an XML schema and data into the DataSet. This method has different overloaded forms.
ReadXMLSchema(0)	Reads an XML schema into the DataSet. This method has different overloaded forms.
RejectChanges	Rolls back all changes made since the last call to AcceptChanges.
WriteXML()	Writes an XML schema and data from the DataSet. This method has different overloaded forms.
WriteXMLSchema()	Writes the structure of the DataSet as an XML schema. This method has different overloaded forms.

The DataTable Class

The DataTable class represents the tables in the database. It has the following important properties; most of these properties are read only properties except the PrimaryKey property:

Properties	Description

ChildRelations	Returns the collection of child relationship.
Columns	Returns the Columns collection.
Constraints	Returns the Constraints collection.
DataSet	Returns the parent DataSet.
DefaultView	Returns a view of the table.
ParentRelations	Returns the ParentRelations collection.
PrimaryKey	Gets or sets an array of columns as the primary key for the table.
Rows	Returns the Rows collection.

The following table shows some important methods of the DataTable class:

Methods	**Description**
AcceptChanges	Commits all changes since the last AcceptChanges.
Clear	Clears all data from the table.
GetChanges	Returns a copy of the DataTable with all changes made since the AcceptChanges method was called.
GetErrors	Returns an array of rows with errors.
ImportRows	Copies a new row into the table.
LoadDataRow	Finds and updates a specific row, or creates a new one, if not found any.
Merge	Merges the table with another DataTable.
NewRow	Creates a new DataRow.
RejectChanges	Rolls back all changes made since the last call to AcceptChanges.
Reset	Resets the table to its original state.
Select	Returns an array of DataRow objects.

The DataRow Class

The DataRow object represents a row in a table. It has the following important properties:

Properties	**Description**
HasErrors	Indicates if there are any errors.

Items	Gets or sets the data stored in a specific column.
ItemArrays	Gets or sets all the values for the row.
Table	Returns the parent table.

The following table shows some important methods of the DataRow class:

Methods	**Description**
AcceptChanges	Accepts all changes made since this method was called.
BeginEdit	Begins edit operation.
CancelEdit	Cancels edit operation.
Delete	Deletes the DataRow.
EndEdit	Ends the edit operation.
GetChildRows	Gets the child rows of this row.
GetParentRow	Gets the parent row.
GetParentRows	Gets parent rows of DataRow object.
RejectChanges	Rolls back all changes made since the last call to AcceptChanges.

The DataAdapter Object

The DataAdapter object acts as a mediator between the DataSet object and the database. This helps the Dataset to contain data from multiple databases or other data source.

The DataReader Object

The DataReader object is an alternative to the DataSet and DataAdapter combination. This object provides a connection oriented access to the data records in the database. These objects are suitable for read-only access, such as populating a list and then breaking the connection.

DbCommand and DbConnection Objects

The DbConnection object represents a connection to the data source. The connection could be shared among different command objects.

The DbCommand object represents the command or a stored procedure sent to the database from retrieving or manipulating data.

Example

So far, we have used tables and databases already existing in our computer. In this example, we will create a table, add column, rows and data into it and display the table using a GridView object.

The source file code is as given:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="createdatabase._Default" %>


<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">


<html xmlns="http://www.w3.org/1999/xhtml" >


  <head runat="server">
    <title>
      Untitled Page
    </title>
  </head>


  <body>
    <form id="form1" runat="server">


      <div>
        <asp:GridView ID="GridView1" runat="server">
        </asp:GridView>
      </div>


    </form>
  </body>


</html>
```

The code behind file is as given:

```
namespace createdatabase
{
```

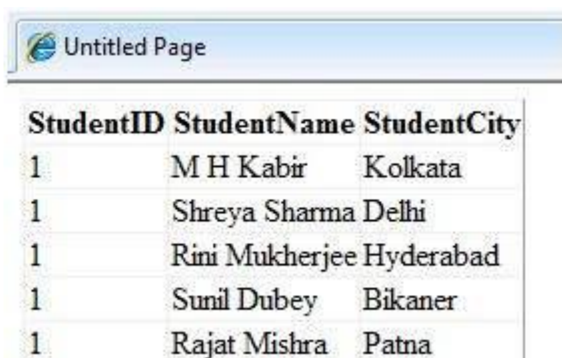
```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    if (!IsPostBack)
    {
      DataSet ds = CreateDataSet();
      GridView1.DataSource = ds.Tables["Student"];
      GridView1.DataBind();
    }
  }

  private DataSet CreateDataSet()
  {
    //creating a DataSet object for tables
    DataSet dataset = new DataSet();

    // creating the student table
    DataTable Students = CreateStudentTable();
    dataset.Tables.Add(Students);
    return dataset;
  }

  private DataTable CreateStudentTable()
  {
    DataTable Students = new DataTable("Student");

    // adding columns
    AddNewColumn(Students, "System.Int32", "StudentID");
```

```
      AddNewColumn(Students, "System.String", "StudentName");
      AddNewColumn(Students, "System.String", "StudentCity");

      // adding rows
      AddNewRow(Students, 1, "M H Kabir", "Kolkata");
      AddNewRow(Students, 1, "Shreya Sharma", "Delhi");
      AddNewRow(Students, 1, "Rini Mukherjee", "Hyderabad");
      AddNewRow(Students, 1, "Sunil Dubey", "Bikaner");
      AddNewRow(Students, 1, "Rajat Mishra", "Patna");

      return Students;
    }

    private void AddNewColumn(DataTable table, string columnType, string  columnName)
    {
      DataColumn column = table.Columns.Add(columnName,  Type.GetType(columnType));
    }

    //adding data into the table
    private void AddNewRow(DataTable table, int id, string name, string city)
    {
      DataRow newrow = table.NewRow();
      newrow["StudentID"] = id;
      newrow["StudentName"] = name;
      newrow["StudentCity"] = city;
      table.Rows.Add(newrow);
    }
  }
}
```

When you execute the program, observe the following:

- The application first creates a data set and binds it with the grid view control using the DataBind() method of the GridView control.
- The Createdataset() method is a user defined function, which creates a new DataSet object and then calls another user defined method CreateStudentTable() to create the table and add it to the Tables collection of the data set.
- The CreateStudentTable() method calls the user defined methods AddNewColumn() and AddNewRow() to create the columns and rows of the table as well as to add data to the rows.

When the page is executed, it returns the rows of the table as shown:


Untitled Page

StudentID	StudentName	StudentCity
1	M H Kabir	Kolkata
1	Shreya Sharma	Delhi
1	Rini Mukherjee	Hyderabad
1	Sunil Dubey	Bikaner
1	Rajat Mishra	Patna

DataReader

DataReader is used to read the data from database and it is a read and forward only connection oriented architecture during fetch the data from database. DataReader will fetch the data very fast when compared with dataset. Generally we will use ExecuteReader object to bind data to datareader.

To bind DataReader data to GridView we need to write the code like as shown below:

```
1. Protected void BindGridview() {
2.    using(SqlConnection conn = new SqlConnection("Data Source=abc;Integrated Security=true;
   Initial Catalog=Test")) {
3.        con.Open();
4.        SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location
   FROM Users", conn);
5.        SqlDataReader sdr = cmd.ExecuteReader();
6.        gvUserInfo.DataSource = sdr;
7.        gvUserInfo.DataBind();
8.        conn.Close();
9.    }
10. }
```

- Holds the connection open until you are finished (don't forget to close it!).
- Can typically only be iterated over once
- Is not as useful for updating back to the database

DataSet

DataSet is a disconnected orient architecture that means there is no need of active connections during work with datasets and it is a collection of DataTables and relations between tables. It is used to hold multiple tables with data. You can select data form tables, create views based on table and ask child

rows over relations. Also DataSet provides you with rich features like saving data as XML and loading XML data.

```
1. protected void BindGridview() {
2.     SqlConnection conn = new SqlConnection("Data Source=abc;Integrated Security=true;Initial
   Catalog=Test");
3.     conn.Open();
4.     SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location F
   ROM Users", conn);
5.     SqlDataAdapter sda = new SqlDataAdapter(cmd);
6.     DataSet ds = new DataSet();
7.     da.Fill(ds);
8.     gvUserInfo.DataSource = ds;
9.     gvUserInfo.DataBind();
10. }
```

DataAdapter

DataAdapter will acts as a Bridge between DataSet and database. This dataadapter object is used to read the data from database and bind that data to dataset. Dataadapter is a disconnected oriented architecture. Check below sample code to see how to use DataAdapter in code:

```
1. protected void BindGridview() {
2.     SqlConnection con = new SqlConnection("Data Source=abc;Integrated Security=true;Initial
   Catalog=Test");
3.     conn.Open();
4.     SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location F
   ROM Users", conn);
5.     SqlDataAdapter sda = new SqlDataAdapter(cmd);
6.     DataSet ds = new DataSet();
7.     da.Fill(ds);
8.     gvUserInfo.DataSource = ds;
9.     gvUserInfo.DataBind();
10. }
```

- Lets you close the connection as soon it's done loading data, and may even close it for you automatically
- All of the results are available in memory
- You can iterate over it as many times as you need, or even look up a specific record by index
- Has some built-in faculties for updating back to the database.

DataTable

DataTable represents a single table in the database. It has rows and columns. There is no much difference between dataset and datatable, dataset is simply the collection of datatables.

```
1. protected void BindGridview() {
2.     SqlConnection con = new SqlConnection("Data Source=abc;Integrated Security=true;Initial
   Catalog=Test");
3.     conn.Open();
4.     SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location F
   ROM Users", conn);
5.     SqlDataAdapter sda = new SqlDataAdapter(cmd);
```

```
6.     DataTable dt = new DataTable();
7.     da.Fill(dt);
8.     gridview1.DataSource = dt;
9.     gvidview1.DataBind();
10. }
```

Stored Procedure

- A stored procedure is a precompiled set of one or more SQL statements which perform some specific task.
- A stored procedure should be executed stand alone using EXEC
- A stored procedure can return multiple parameters
- A stored procedure can be used to implement transact
- Get database result sets from some business logic on data.
- Execute multiple database operations in a single call.
- Used to migrate data from one table to another table.
- Can be called for other programming languages, like Java.

Advantages of using stored procedures

- A stored procedure allows modular programming.

  You can create the procedure once, store it in the database, and call it any number of times in your program.
- A stored procedure allows faster execution.

  If the operation requires a large amount of SQL code that is performed repetitively, stored procedures can be faster. They are parsed and optimized when they are first executed, and a compiled version of the stored procedure remains in a memory cache for later use. This means the stored procedure does not need to be reparsed and reoptimized with each use, resulting in much faster execution times.
- A stored procedure can reduce network traffic.

  An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.
- Stored procedures provide better security to your data

  Users can be granted permission to execute a stored procedure even if they do not have permission to execute the procedure's statements directly.

  In SQL Server we have different types of stored procedures:
  - System stored procedures
  - User-defined stored procedures
  - Extended stored Procedures
- **System**-stored procedures are stored in the master database and these start with a sp_ prefix. These procedures can be used to perform a variety of tasks to support SQL Server functions for external application calls in the system tables
  Example: sp_helptext [StoredProcedure_Name]
- **User-defined** stored procedures are usually stored in a user database and are typically designed to complete the tasks in the user database. While coding these procedures **don't use**the sp_ prefix

because if we use the sp_ prefix first, it will check the master database, and then it comes to user defined database.

- **Extended** stored procedures are the procedures that call functions from DLL files. Nowadays, extended stored procedures are deprecated for the reason it would be better to avoid using extended stored procedures.

Comparison with Function

- A function is a subprogram written to perform certain computations.
- A scalar function returns only one value (or NULL), whereas a table function returns a (relational) table comprising zero or more rows, each row with one or more columns.
- Functions must return a value (using the RETURN keyword), but for stored procedures this is not mandatory.
- Stored procedures can use RETURN keyword but with no value being passed.
- Functions could be used in SELECT statements, provided they do no data manipulation. However, procedures cannot be included in SELECT statements.
- A stored procedure can return multiple values using the OUT parameter, or return no value.
- A stored procedure saves the query compiling time.
- A stored procedure is a database object.
- A stored procedure is a material object.

Comparison with View

Views relvars that contain tuples.

Stored Procedures are scripts.

Views are useful if there is a certain combination of tables, or a subset of data you consistently want to query, for example, an user joined with its permissions. Views should in fact be treated as tables.

Stored procedures are pieces of sql code that are 'compiled', as it where, to run more optimally than a random other query. The execution plan of sql code in a stored procedure is already built, so execution runs slightly smoother than that of an ordinary sql statement.

Use stored procedure instead of view if you don't want insertion to be possible. Inserting in a view may not give what it seems to do. It will insert in a table, a row which may not match the query from the view, a row which will then not appear in the view; inserted somewhere, but not where the statement make it seems.

Use a view if you can't use the result of a stored procedure from another stored procedure (I was never able to make the latter works, at least with MySQL).

The main advantage of stored procedures is that they allow you to incorporate logic (scripting). This logic may be as simple as an IF/ELSE or more complex such as DO WHILE loops, SWITCH/CASE

A stored procedure:

* accepts parameters
* can NOT be used as building block in a larger query
* can contain several statements, loops, IF ELSE, etc.
* can perform modifications to one or several tables
* can NOT be used as the target of an INSERT, UPDATE or DELETE statement.

A view:

* does NOT accept parameters

* can be used as building block in a larger query
* can contain only one single SELECT query
* can NOT perform modifications to any table
* but can (sometimes) be used as the target of an INSERT, UPDATE or DELETE statement.