Materiały do przedmiotu Database system administration na kierunku Technologie Informatyczne w Logistyce w języku angielskim realizowanego w ramach projektu InfoGeolog

Łódź 2015

Course content

1 Introduction to relational database systems....3
- 1.1 Basic concepts....3
- 1.2 Architecture of database management system....3
- 1.3 Structured Query Language (SQL)....5
- 1.4 Main database objects....6
- 1.5 Examples of RDBMSs....6

2 Architecture of Database Management System (SQL Server)....6
- 2.1 SQL Server instances....7
- 2.2 SQL Server tools....7
- 2.3 Connecting to the SQL Server....10

3 Database architecture....12
- 3.1 Database files....12
  - 3.1.1 Pages and extents....13
  - 3.1.2 Filegroups....13
  - 3.1.3 System databases....14
  - 3.1.4 User-defined databases....14

4 Creating database objects....18
- 4.1 Creating tables....18
- 4.2 Main SQL Server data types....19
- 4.3 Altering tables....20
- 4.4 Dropping tables....22
- 4.5 Views....22
- 4.6 Indices....24
  - 4.6.1 Creating indexes....25
- 4.7 Altering indexes....26
- 4.8 Working with indices....26

5 Query optimization....31
- 5.1 Main elements of SQL Server execution plans....34

6 Stored procedures, functions and triggers....36
- 6.1 Conditional statements....36
- 6.2 Loops....37
- 6.3 Cursors....38
- 6.4 Stored procedures....38
- 6.5 Functions....40
- 6.6 Triggers....41

7 Security and user authorization in SQL Server....43
- 7.1 SQL Server authentication....44
- 7.2 SQL Server authorization....48
- 7.3 Ownership chains 55

8 Backups in SQL Server....64
- 8.1 Full database backup....64
- 8.2 Differential backups....68
- 8.3 Transaction log backups....68

9 Restoring SQL Server Databases....70
- 9.1 Restoring process....70
- 9.2 Restoring to the point of failure....70
- 9.3 Restoring to the point in time....71

9.4 Restoring to the marked transactions....72
9.5 Restoring system databases....73
9.6 Backup Recovery Assignment....74
9.7 Sample recovery procedure....74
10 High Availability....75
10.1 Log Shipping....76
10.2 Database Mirroring....76
10.3 AlwaysOn Availability Groups....77
10.4 AlwaysOn Failover Cluster Instances....77
11 Bibliography....80

1 Introduction to relational database systems

1.1 Basic concepts

A database consists of files which are stored on disks. A database management system (DBMS) involves a software which is responsible for managing these files. Database management systems must deal with the following issues:

- Efficient data access.
- Transactions (ACID).
- Data integrity.
- Concurrency control.
- Data consistency.
- Backup and restoring mechanisms.
- Efficient query processing and optimization.

1.2 Architecture of database management system

The most popular framework for database management systems is called ANSI/SPARC [TK78, RG03]. It provides three levels: *external*, *conceptual*, and *internal*. The first level defines the data model for users and client applications. It consists of user schemas, privileges, tables, views, etc. The conceptual level provides the data model for the whole enterprise. It involves the database catalog and its schema. The internal level provides physical data storage.

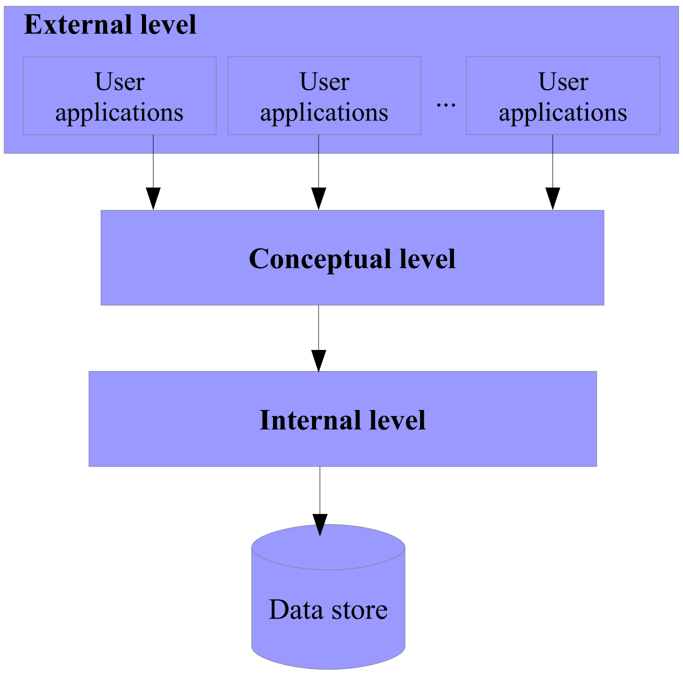


Figure 1.1: ANSI/SPARC architecture

Database management system involves the following components:

- *Query parser* – it performs syntactic and semantic analysis of user queries.
- *Query optimizer* – it analyses different execution plans for a given query and selects the plan with the lowest cost.
- *Query processor* – it runs the proper query plan and returns its result.
- *Resource manager* - it is responsible for managing records in files.
- *Buffer memory manager* – it moves data pages from disk to memory buffers.
- *Data storage manger* – it allows to obtain data from disk.
- *Transaction manager* – it ensures atomicity, isolation, consistency and durability of transactions.
- *Recovery manager* – it is responsible for recovering a database in the case of failure.

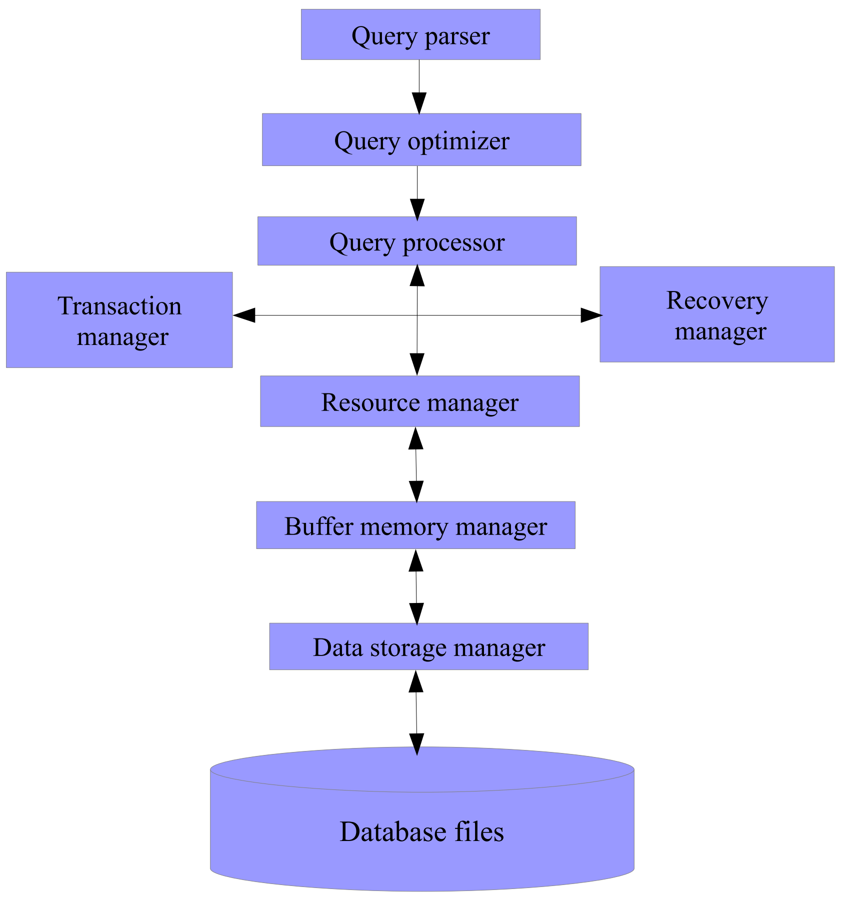


Figure 1.2: Architecture of DBMS

1.3 Structured Query Language (SQL)

Each RDBMS defines a language to manipulate database objects and their data. It is called SQL (Structured Query Language). It has several dialects (e.g. T-SQL). SQL involves the following components:

- Data Definition Language (DDL) – it is used to create, modify, and remove database objects (e.g. tables, indices, views).
- Data Manipulation Language (DML) – it is used to select, insert, update, and delete data from tables.
- Data Control Language (DCL) – it is used to grant and revoke database privileges.

Example 1.1: Sample SQL query.

The query below retrieves information (first name, last name and gender) about students whose age is greater than 21.

```
SELECT FIRST_NAME, LAST_NAME, GENDER
FROM STUDENTS
WHERE AGE > 21
```

1.4 Main database objects

The most important database objects are:

- *Table* – it is a set of rows in which each row has the same number of fields (columns).
- *Constraint* – it provides rules to enforce the integrity of a database.
- *View*- it is a virtual table (named query).
- *Index* – it is auxiliary (redundant) data structure which lets to speed up query processing.
- *Procedure* – it is a saved collection of SQL statements.
- *Function* – it is a stored procedure that must return a value.
- *Trigger* – it is a special kind of the stored procedure which automatically runs whenever the proper event occurs in the database.

1.5 Examples of RDBMSs

The most known relational database management systems are:

- Microsoft SQL Server (https://www.microsoft.com/sqlserver)
- Oracle Server (www.oracle.com/database)
- DB2 (www.ibm.com/db2)
- Sybase (www.sybase.com)
- MySQL (http://www.mysql.com/)
- PostgreSQL (http://www.postgresql.org/)

2 Architecture of Database Management System (SQL Server)

SQL Server is very powerful relational database management system. It provides among others the following elements:

- High availability mechanisms (AlwaysOn Availability Groups, AlwaysOn Failover Cluster Instances, Log Shipping, Database Mirroring).
- Transaction management.
- Authentication and authorization mechanisms.
- Backup and restore mechanisms.

There are several SQL Server editions [BOL12]:

- Datacenter (the most powerful in terms of scalability and virtualisation).
- Enterprise (it allows to use more processors and consume more memory than the Standard one).
- Standard.

- Developer (it is dedicated to developers).
- Express (free edition).

2.1 SQL Server instances

Database administrator can install more than one copy of SQL Server on the same server. Each copy is called an *instance* of SQL Server. Each instance consists of a set of specific services. There are two instance types:

- default instance (called *MSSQLSERVER*),
- named instance (User must provide a name during installation).

The most common SQL Server services are [BOL12]:

- SQL Server service (database engine).
- SQL Server Agent - it allows to automate some administrative tasks (e.g. backups).
- Integration Services (ETL – Extract-Transform-Load).
- Analysis Services.
- Reporting Services.

Starting a named instance:

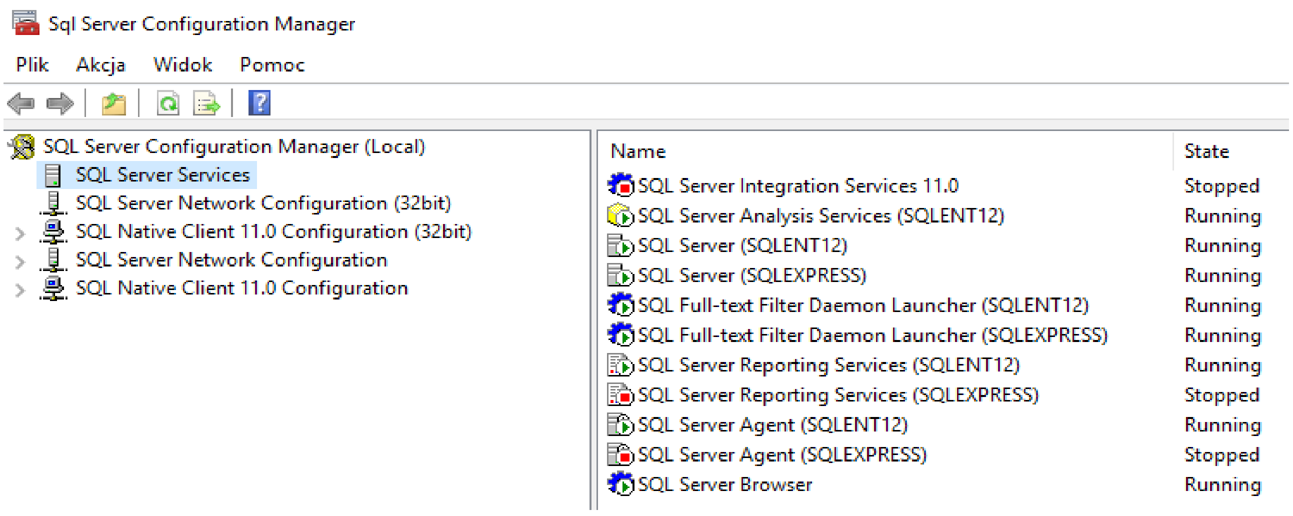
```
net start MSSQL$instance_name
```

Starting the default instance:

```
net start MSSQL
```

2.2 SQL Server tools

SQL Server Configuration Manager (SSCM) [BOL12] lets to configure instance services and network protocols.

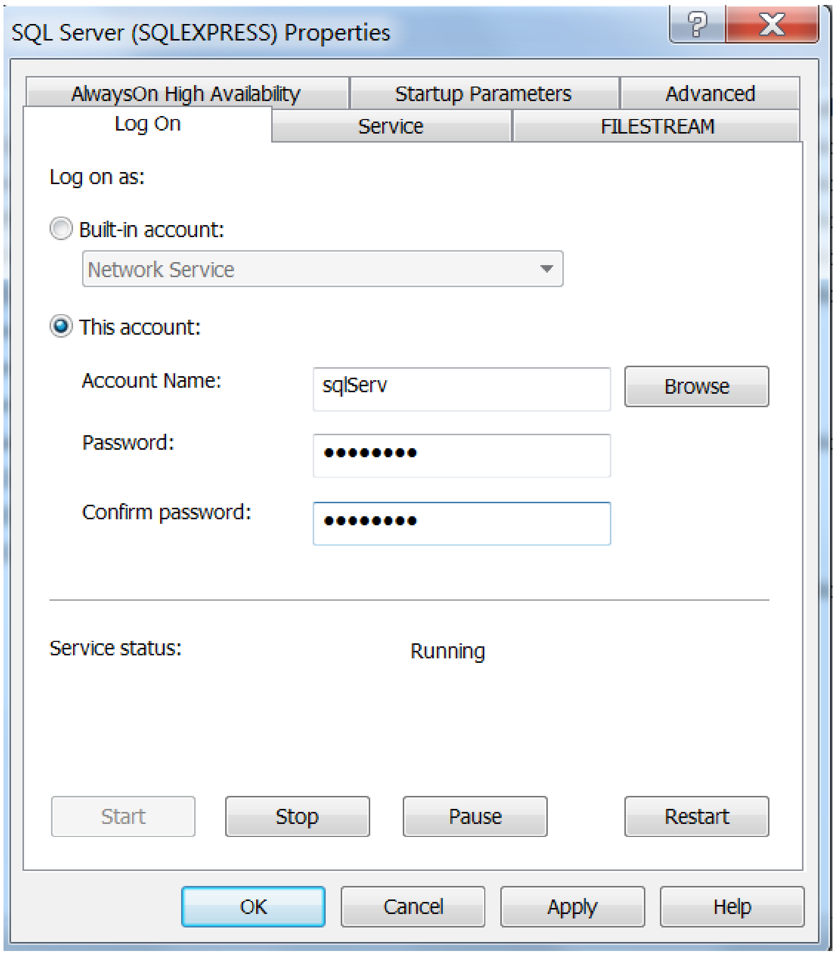

Sql Server Configuration Manager
Plik
Akcja
Widok
Pomoc
SQL Server Configuration Manager (Local)
SQL Server Services
SQL Server Network Configuration (32bit)
SQL Native Client 11.0 Configuration (32bit)
SQL Server Network Configuration
SQL Native Client 11.0 Configuration

Name	State
SQL Server Integration Services 11.0	Stopped
SQL Server Analysis Services (SQLENT12)	Running
SQL Server (SQLENT12)	Running
SQL Server (SQLEXPRESS)	Running
SQL Full-text Filter Daemon Launcher (SQLENT12)	Running
SQL Full-text Filter Daemon Launcher (SQLEXPRESS)	Running
SQL Server Reporting Services (SQLENT12)	Running
SQL Server Reporting Services (SQLEXPRESS)	Stopped
SQL Server Agent (SQLENT12)	Running
SQL Server Agent (SQLEXPRESS)	Stopped
SQL Server Browser	Running

Figure 2.1. SQL Server Configuration Manager

Each service has the service account. SSCM allows to mange service accounts. The following service accounts are supported:

- Network Service.
- Local Service.
- Local System.
- Local Windows account.
- Domain Windows Account.


SQL Server (SQLEXPRESS) Properties

Log On tab

Log on as:

Built-in account: Network Service

[✓ This account:]

Account Name: sqlServ

Password: ●●●●●●●●●

Confirm password: ●●●●●●●●●

Service status: Running

Figure 2.2. Configuring service account

SQL Server Management Studio

SQL Server Management Studio (SSMS) [BOL12] is an application which allows to manage all SQL Server components.

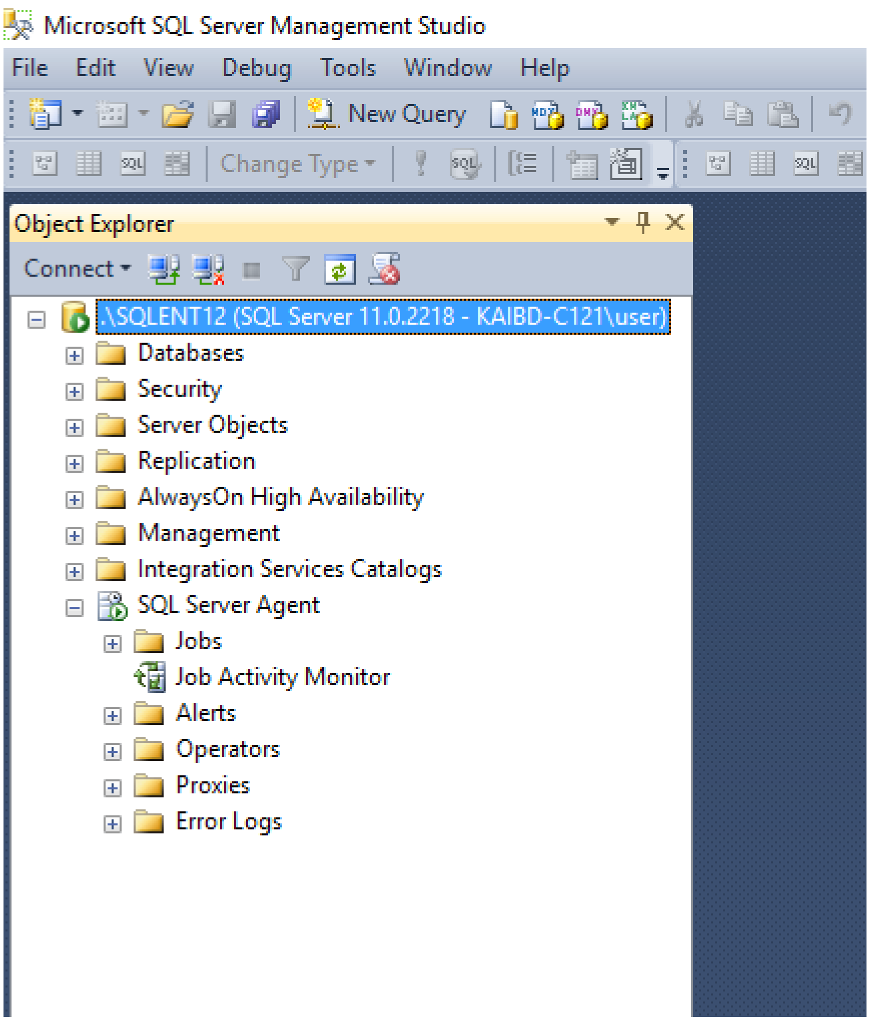
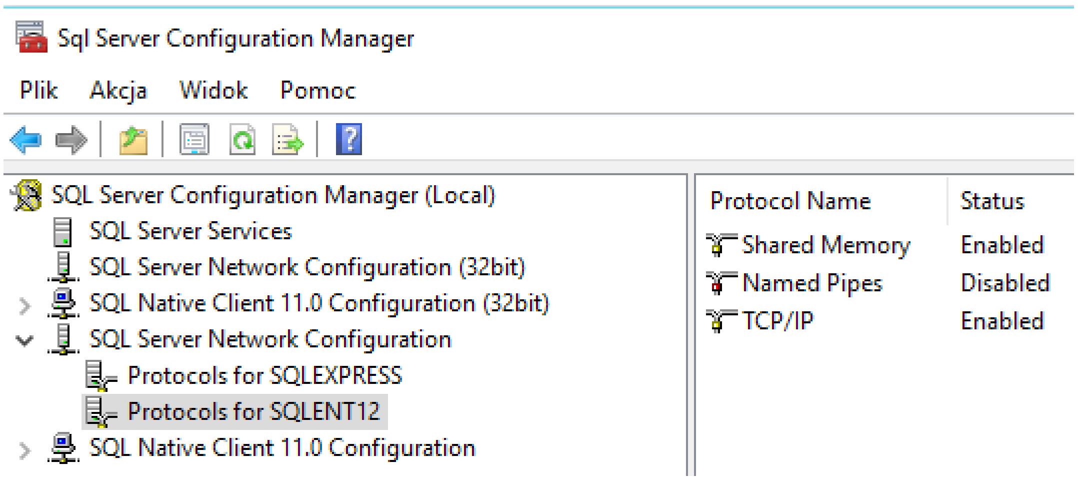

Microsoft SQL Server Management Studio
File Edit View Debug Tools Window Help
New Query
Change Type
Object Explorer
Connect
.\SQLENT12 (SQL Server 11.0.2218 - KAIBD-C121\user)
Databases
Security
Server Objects
Replication
AlwaysOn High Availability
Management
Integration Services Catalogs
SQL Server Agent
Jobs
Job Activity Monitor
Alerts
Operators
Proxies
Error Logs

Figure 2.3. SQL Server Management Studio

2.3 Connecting to the SQL Server

SQL Server supports the following network protocols:


Sql Server Configuration Manager

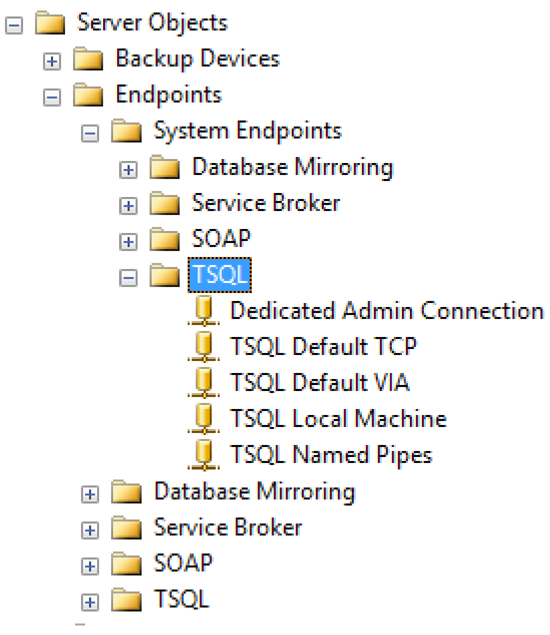
Plik Akcja Widok Pomoc

SQL Server Configuration Manager (Local)
SQL Server Services
SQL Server Network Configuration (32bit)
SQL Native Client 11.0 Configuration (32bit)
SQL Server Network Configuration
Protocols for SQLEXPRESS
Protocols for SQLENT12
SQL Native Client 11.0 Configuration

Protocol Name	Status
Shared Memory	Enabled
Named Pipes	Disabled
TCP/IP	Enabled

- Shared Memory – it is dedicated to local connections.
- TCP/IP - it deals with user connections which are done through the network.
- Named Pipes – it lets to network access to an instance of SQL Server.

The special objects called *endpoints* are created for each network protocol. The endpoints have the following names [BOL12]:


Server Objects
Backup Devices
Endpoints
System Endpoints
Database Mirroring
Service Broker
SOAP
TSQL
Dedicated Admin Connection
TSQL Default TCP
TSQL Default VIA
TSQL Local Machine
TSQL Named Pipes
Database Mirroring
Service Broker
SOAP
TSQL

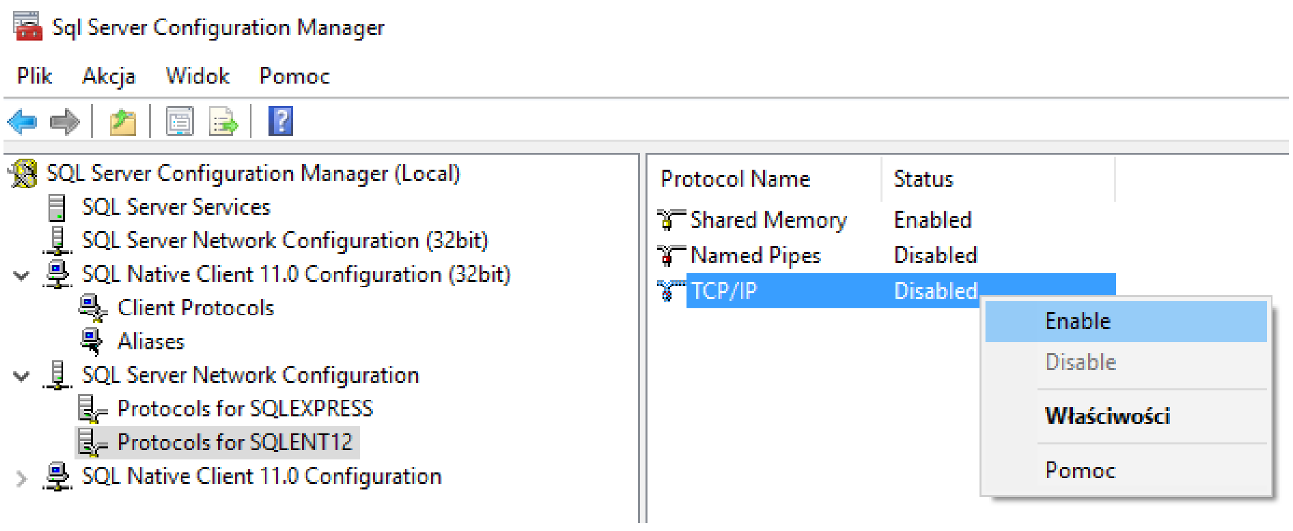
- TSQL Local Machine (for the Shared Memory protocol),
- TSQL Named Pipes,

- TSQL Default TCP/IP,
- TSQL Default VIA.

Configuring network protocols

The following steps are required to activate the TCP/IP protocol for the SQLENT12 instance:

- Expand *SQL Server Network Configuration* in the left pane.
- Click *Protocols for SQLENT12.*
- Right-click the TCP/IP protocol and then click Enable.

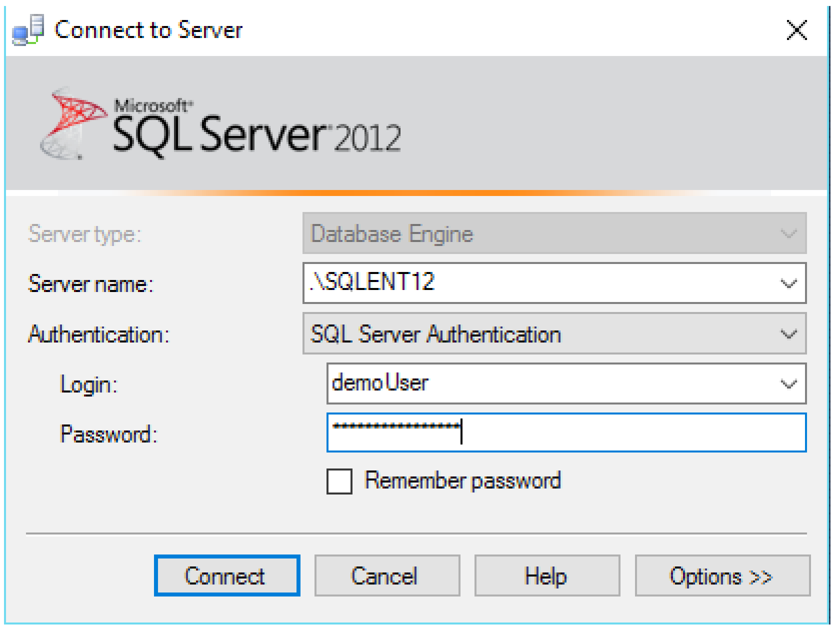

Sql Server Configuration Manager
Plik
Akcja
Widok
Pomoc
SQL Server Configuration Manager (Local)
SQL Server Services
SQL Server Network Configuration (32bit)
SQL Native Client 11.0 Configuration (32bit)
Client Protocols
Aliases
SQL Server Network Configuration
Protocols for SQLEXPRESS
Protocols for SQLENT12
SQL Native Client 11.0 Configuration
Enable
Disable
Właściwości
Pomoc

Protocol Name	Status
Shared Memory	Enabled
Named Pipes	Disabled
TCP/IP	Disabled

Figure 2.4. Configuring network protocols

Connecting to the SQL Server Database Engine with SSMS.

1. Launch SSMS.
2. Provide server name (e.g. .\SQLENT12 – it allows to connect to the named instance *SQLENT12* which resides on the local computer).
3. Select the authentication mode.
4. Provide user name and password.



Server type: Database Engine
Server name: .\SQLENT12
Authentication: SQL Server Authentication
Login: demoUser
Password: ****************
Remember password

Figure 2.5. Connecting to SQL Server

3 Database architecture

3.1 Database files

Each SQL Server database includes at least two files:

- *primary database file* – it stores data and has the extension **.mdf*,
- *transaction log file* – it records all modifications performed by database transactions.

A database can have several data files. Additional data files (secondary data files) have the extension **.ndf*. Applying multiple data files allows to:

- increase performance,
- use more CPU cores,
- perform smaller backups.

The following figure presents the structure of a SQL Server database:

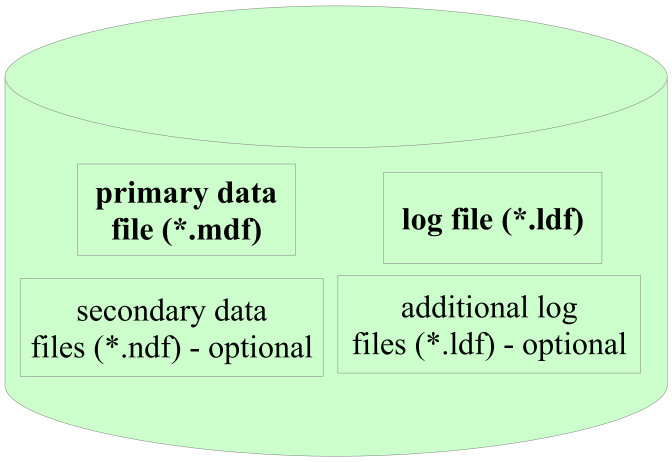

primary data file (*.mdf)
log file (*.ldf)
secondary data files (*.ndf) - optional
additional log files (*.ldf) - optional

Figure 3.1. SQL Server database structure

3.1.1 Pages and extents

A *page* [BOL12] is the basic unit of data storage. It has the size of 8kB in SQL Server. All read/write operations are preformed on the page level. A page consists of:

- header,
- area in which data is stored – it has the size of 8060 bytes.

An extent consists of eight pages. Extents are introduced to simplify space management.

3.1.2 Filegroups

Filegroups [BOL12] are containers for databases files. They allow to deal with the following tasks:

- controlling in which files database objects (e.g. tables, indexes) are stored,
- simplify and speed up backup and restore procedures.

Each SQL Server database has so called *primary* filegroup. The primary database file is a member of this group. Other data files can belong to the primary filegroup or to other groups.

The following figure presents the structure of a SQL Server database which has six files: four data files and two log files.

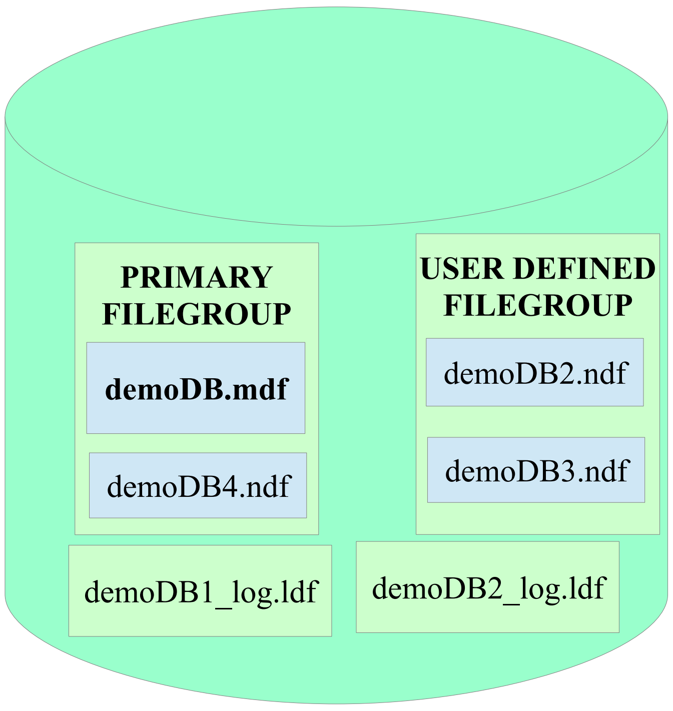


Figure 3.2. Database structure and filegroups

3.1.3 System databases

SQL Server provides the following system databases [BOL12]:

system	It stores the instance metadata and configuration information.
model	It is the template for user-defined databases.
msdb	It stores backup and restore history, SQL Server Agent objects.
resource	This is read only database which stores all the system objects.
tempdb	It stores temporary objects and intermediate query results.

3.1.4 User-defined databases

SQL Server allows to create user-defined databases and manage them. The basic syntax of CREATE DATABASE statement is as follows [BOL12]:

```
CREATE DATABASE database_name
[ON
     [ PRIMARY ] [ files_defintion [ ,...n ] ]
     [ , FILEGROUP filegroup_name file_defintion [ ,...n ] ]
LOG ON
     files_defintion [ ,...n ]
]

file_defintion :=
    (NAME = logical_file_name ,
      FILENAME = 'physical_file_name'
      [ , SIZE = file_size [ KB | MB | GB | TB ] ]
      [ , MAXSIZE = { max_size [ KB | MB | GB | TB ] | UNLIMITED ]
      [ , FILEGROWTH = growth_value [ KB | MB | GB | TB | % ] ]
     ) [ ,...n ]
```

database_name – it represents the name of a new database.

ON – it lets to explicitly define database files.

PRIMARY – it indicates that the first file presented in *file_definition* becomes the primary data file.

LOG ON – it lets to explicitly define log files.

file_definition – it specifies list of files and their properties.

NAME = logical_file_name – it specifies the logical file name.

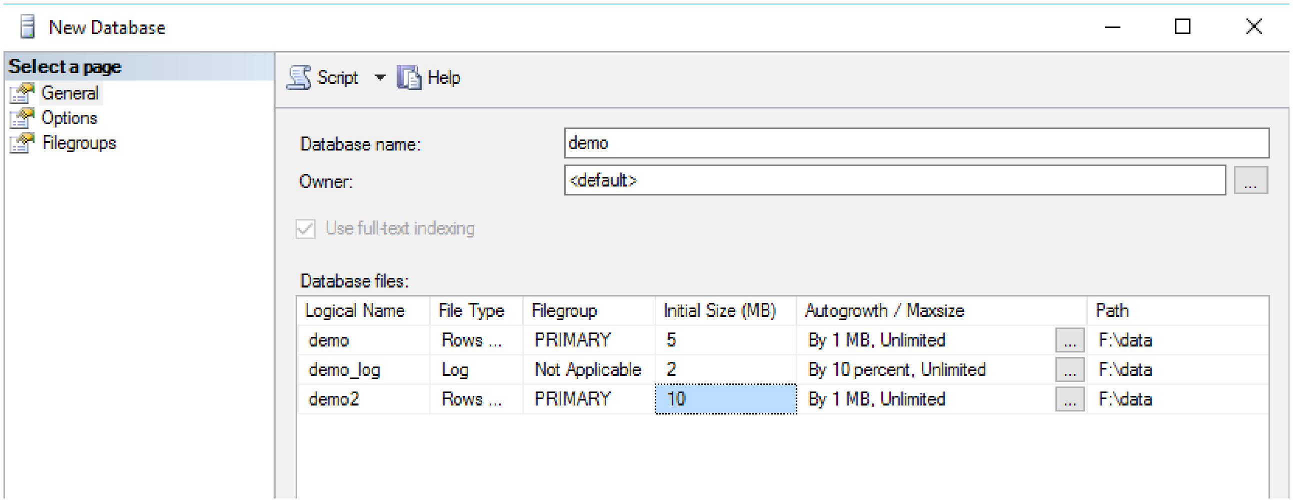
FILENAME = 'physical_file_name' – it specifies the physical file name.

SIZE = file_szie – it specifies the size of the file.

MAXSIZE = max_szie – it specifies the maximum size of the file. The file can grow until it reaches *max_size*.

FILEGROWTH = growth_value – it automatically increases the file when there is no enough space

Example 3.1: Creating a sample database using SSMS and T-SQL.



Database name: demo

Owner: <default>

Use full-text indexing

Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize	Path
demo	Rows ...	PRIMARY	5	By 1 MB, Unlimited	F:\data
demo_log	Log	Not Applicable	2	By 10 percent, Unlimited	F:\data
demo2	Rows ...	PRIMARY	10	By 1 MB, Unlimited	F:\data

```
CREATE DATABASE demo
ON  PRIMARY
( NAME = N'demo', FILENAME = 'f:\data\demo.mdf' ,
      SIZE = 5120KB , FILEGROWTH = 1024KB ),
 FILEGROUP SECONDONARY
( NAME = N'demo2', FILENAME = 'f:\data\demo2.ndf' ,
      SIZE = 5120KB , FILEGROWTH = 1024KB )
 LOG ON
( NAME = N'demoDB_log', FILENAME = 'f:\data\demo_log.ldf' ,
      SIZE = 1024KB , FILEGROWTH = 10%)
```

Example 3.2: Adding a file to the database demo.

```
ALTER DATABASE [demo] ADD FILE (
NAME = N'demo3',
FILENAME = N'f:\DATA\demo3.ndf' ,
 SIZE = 5120KB ,
 FILEGROWTH = 1024KB )
 TO FILEGROUP [PRIMARY]
```

Example 3.3: Adding a filegroup called *fgDemo*.

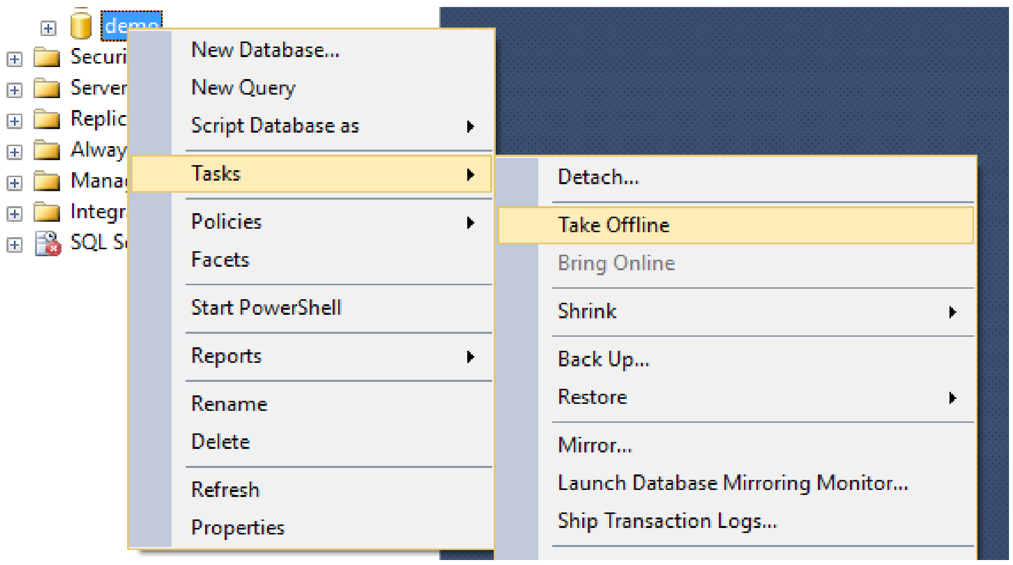
```
ALTER DATABASE [demo] ADD FILEGROUP [fgDemo]
```

Example 3.4: Adding a new file to *fgDemo*

```
ALTER DATABASE [demo] ADD FILE (
NAME = N'demo4',
FILENAME = N'f:\DATA\demo4.ndf' ,
 SIZE = 5120KB ,
 FILEGROWTH = 1024KB )
 TO FILEGROUP [fgDemo]
```

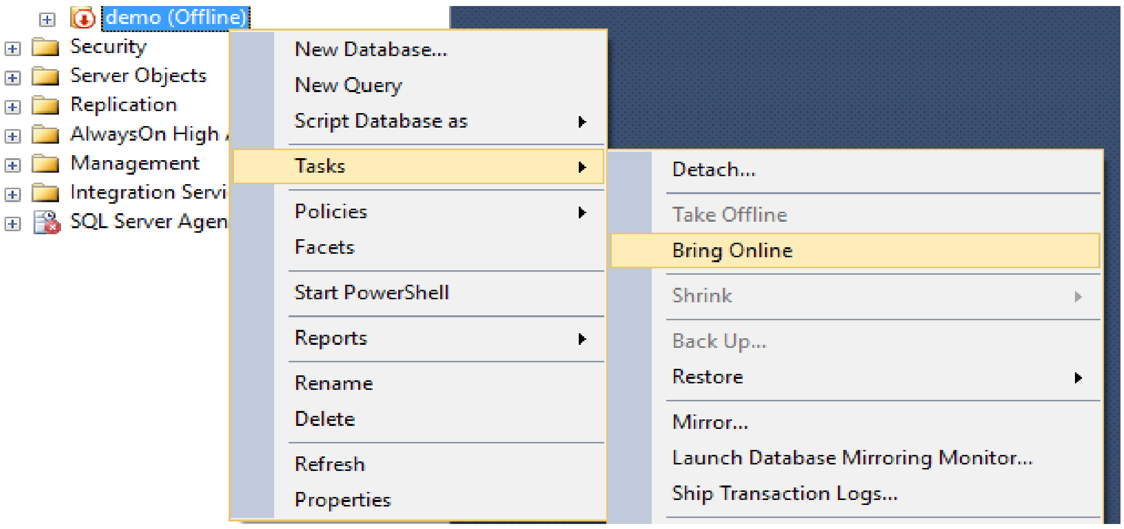
Example 3.5: Take the database to the offline state.

```
ALTER DATABASE [demo] SET OFFLINE WITH ROLLBACK IMMEDIATE
```



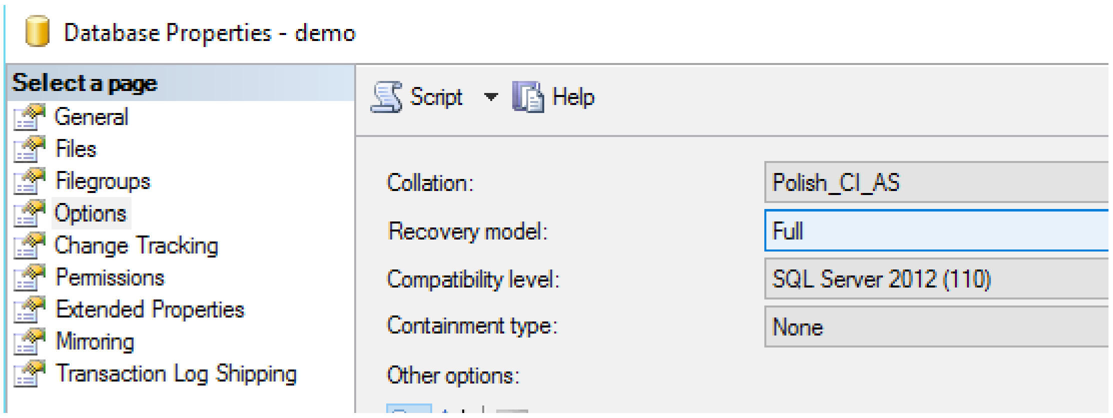
Example 3.6: Take the database to the online state

```
ALTER DATABASE [demo] SET ONLINE
```



Example 3.7: Set the recovery model to *full*.

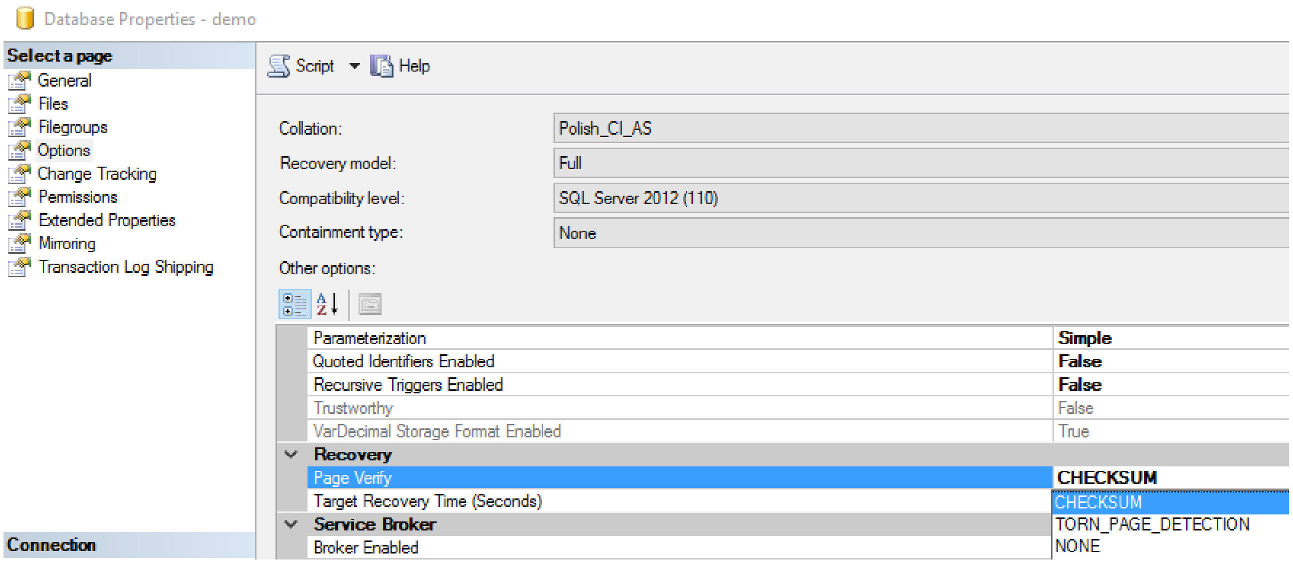
```
ALTER DATABASE [demo] SET RECOVERY FULL
```



Collation: Polish_CI_AS
Recovery model: Full
Compatibility level: SQL Server 2012 (110)
Containment type: None
Other options:

Example 3.8: Set the option *page verify* to *checksum*.

```
ALTER DATABASE [demo] SET PAGE_VERIFY CHECKSUM
```


Database Properties - demo

Select a page
- General
- Files
- Filegroups
- Options
- Change Tracking
- Permissions
- Extended Properties
- Mirroring
- Transaction Log Shipping

Connection

Collation:	Polish_CI_AS
Recovery model:	Full
Compatibility level:	SQL Server 2012 (110)
Containment type:	None

Other options:

Parameterization	Simple
Quoted Identifiers Enabled	False
Recursive Triggers Enabled	False
Trustworthy	False
VarDecimal Storage Format Enabled	True
Recovery	
Page Verify	CHECKSUM
Target Recovery Time (Seconds)	CHECKSUM
Service Broker	TORN_PAGE_DETECTION
Broker Enabled	NONE

4 Creating database objects

4.1 Creating tables

The *CREATE TABLE* statement is applied to create a table in a database [BOL12]:

```
CREATE TABLE [schema.]table_name
(
     column1 data_type [column_properties][column constraints]
   [,column2 data_type [column_properties][column constraints]
    ...
   [, columnN data_type [column_properties][column constraints]
)
```

Example 4.1: Create a table called *Student* which involves seven columns: *stud_id*, *first_name*, *last_name*, *birth_date*, *gender*, *email, supervisor_id*.

```
CREATE TABLE Student
(
   stud_id int IDENTITY CONSTRAINT PK_Stud_id PRIMARY KEY,
   first_name varchar(25) NOT NULL,
   last_name varchar(30) NOT NULL,
   birth_date date NOT NULL,
   gender char(1) CONSTRAINT CH_gen
                          CHECK(gender='M' OR gender='F'),
   email varchar(40) CONSTRAINT U_Email UNIQUE,
   supervisor_id int,
   CONSTRAINT FK_sup_id FOREIGN KEY(supervisor_id)
                              REFERENCES Student(stud_id)
)
```

Column properties

The most common column properties are:

- *IDENTITY*(*start_with*, *increment_by*) – it automatically generates unique sequential numbers.
- *NULL* – it allows to *null* values in the column. *NULL* is default.
- *NOT NULL* – it prohibits *null* values in the column.

Column constraints

- *PRIMARY KEY* – it enforces uniqueness for the column and prohibits *NULL* values in it.
- *FOREIGN KEY* – it requires only the values which occurs in the corresponding referenced column.
- *UNIQUE* - it guarantees that values in the column are distinct.
- *CHECK* – it restricts the inserted values to the values which comply with the specified condition.
- *DEFAULT* – it allows to define a default value for the column.

4.2 Main SQL Server data types

SQL Server provides the following data types [BOL12]:

Data type	Description
`char[(size)]`	Fixed-length, non-Unicode character data type of length *size* bytes (*size*∈<*1*, *8000*>). It always has the storage size equal to *size* bytes (even if the inserted value is shorter). The default length is 1 byte, MAX denotes the maximum storage size equal to 2^31 – 1 bytes.
`varchar[(size) | MAX]`	Variable-length, non-Unicode character data type of length *size* bytes (*size*∈<1 *8000*>). It has the storage size equal to the actual length of the inserted value + 2 bytes.
`nchar[(size)]`	Fixed-length, non-Unicode character data type of length *size* bytes (*size*∈<*1*, *4000*>). It always has the storage size equal to two times *size* bytes.
`nvarchar[(size) | MAX]`	Variable-length, non-Unicode character data type of length *size* bytes (*size*∈<*1*, *4000*>). It has the storage size equal to two times the actal length of the inserted value + 2 bytes.
`int`	Integer data type with the range: 2^31 to 2^31-1 (storage: 4B).
`smallint`	Integer data type with the range 2^15 to 2^15-1 (storage: 2B).
`tinyint`	Integer data type with the range 0 to 255 (storage: 1B).

`decimal(p, [s])` `numeric(p, [s])`	Decimal data types having fixed precision *p* (maximum number of decimal digits both before and after the decimal point) and scale (maximum number of decimal digits after the decimal point); ($p \in <1, 38>$, $s \in <0, p>$). The storage size ranges from 5 to 17 depending on *p*.
`float[(m)]`	Approximate-number data type with the range: - 1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308. The optional parameter *m* denotes mantissa ($m \in <1, 53>$). The storage size depends on m and it is equal to 4B or 8B.
`real`	Approximate-number data type with the range: - 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38 (storage 4B).
`money`	Monetary data type with the range: -922,337,203,685,477.5808 to 922,337,203,685,477.5807 (storage: 8B).
`smallmoney`	Monetary data type with the range:- 214,748.3648 to 214,748.3647 (storage: 4B).
`date`	Date data type with the range: 01.01.0001 to 31.12.9999 (accuracy: 1 day, storage: 3B).
`time`	Time data type with the range: 00:00:00.0000000 through 23:59:59.9999999 (accuracy: 100 nanoseconds, storage: from 3B to 5B).
`datetime`	Date data type connected with the time. It ranges from 01.01.1753 to 31.12.9999 (time range: 00:00:00 through 23:59:59.997). The storage size is equal to 8B.
`binary[(n)]`	Fixed-length binary data type of length *n* bytes ($n \in <1, 8000>$, storage: *n* bytes). The default length is 1B.
`varbinary[(n | MAX)]`	Variable-length binary data type of length *n* bytes ($n \in <1$ $8000>$). It has the storage size equal to the actual length of the inserted value + 2 bytes. MAX denotes the maximum storage size equal to 2^31 – 1 bytes.

4.3 Altering tables

The *ALTER TABLE* statement is mainly used to add, alter, or drop columns/constraints in a database table. The basic syntax of this statement is as follows [BOL12]:

```
ALTER TABLE [schema.]table_name
   alter_column_clause
```

```
    add_column_clause
    drop_column_clause
    add_constraint_clause
    drop_constraint_clause;
```

Each clause has the appropriate form (see [BOL12]). Consider the following SQL statements which create two tables:

```
CREATE TABLE Employee(
  emp_id int IDENTITY CONSTRAINT PK_Emp_id PRIMARY KEY,
  last_name varchar(40) NOT NULL)

CREATE TABLE Car(
  car_id int IDENTITY NOT NULL,
  vin varchar(40) NOT NULL)
```

Adding columns

Example 4.2: Add some columns to both tables.

```
ALTER TABLE Employee
ADD first_name varchar(20)

ALTER TABLE Employee
ADD birthDate date NOT NULL

ALTER TABLE Car
ADD reg_number varchar(20)

ALTER TABLE Car
ADD employee_id int
```

Altering columns

Example 4.3: Alter the column *Employee.last_name* (change the data type of *last_name* from *varchar*(40) to *varchar*(60)).

```
ALTER TABLE Employee
ALTER COLUMN last_name varchar(60)
```

Example 4.4: Set *NOT NULL* to *Car.employee_id.*

```
ALTER TABLE Car
ALTER COLUMN employee_id int NOT NULL
```

Adding new constraints

Example 4.5: Create a primary key on the column *Car.car_id.*

```
ALTER TABLE Car
ADD CONSTRAINT PK_Car_id PRIMARY KEY(car_id)
```

Example 4.6: Create a foreign key on *Car.employee_id* which refers to *Employee.emp_id.*

```
ALTER TABLE Car
```

```
ADD CONSTRAINT FK_Emp_id FOREIGN KEY(employee_id)
    REFERENCES Employee(emp_id)
```

Example 4.7: Create *UNIQUE* constraints on *Employee.last_name* and *Car.vin.*

```
ALTER TABLE Employee
ADD CONSTRAINT U_Emp_lName UNIQUE(last_name)

ALTER TABLE Car
ADD CONSTRAINT U_Car_vin UNIQUE(vin)
```

Dropping columns and constraints

Example 4.8: Remove the column *Employee.birthDate.*

```
ALTER TABLE Employee
DROP COLUMN birthDate
```

Example 4.9: Remove the *UNIQUE* constraint *U_Emp_lName.*

```
ALTER TABLE Employee
DROP CONSTRAINT U_Emp_lName
```

4.4 Dropping tables

The *DROP TABLE* statement is applied to drop database tables [BOL12]:

```
DROP TABLE [schema.]table_name
```

Example 4.10: Drop the table *Employee.*

```
DROP TABLE Employee
Could not drop object 'Employee' because it is referenced by a
FOREIGN KEY constraint.
```

Example 4.11: Drop the constraint *FK_Emp_id* and the table *Employee.*

```
ALTER TABLE Car
DROP CONSTRAINT FK_Emp_id

DROP TABLE Employee
```

Example 4.12: Drop the table *Car*

```
DROP TABLE Car
```

4.5 Views

The basic *CREATE VIEW* statement has the following syntax [BOL12]:

```
CREATE VIEW view_name
[ WITH SCHEMABINDING ]
AS select_statement
[ WITH CHECK OPTION ]
```

`WITH SCHEMABINDING` - it prevents modifications of the view tables which influence its definition.

`select_statement` – it defines a select statement on which the view is based.

`WITH CHECK OPTION` – it prohibits modifications which may produce records that are not included in the result set of *select_statement*.

Using views

Consider the following *Person* table:

```
CREATE TABLE Person
(
   [personId] int CONSTRAINT PK_Person_Id PRIMARY KEY,
   firstName varchar(25) NOT NULL,
   lastName varchar(30) NOT NULL,
   birthDate date NOT NULL,
   email varchar(40)
)
insert into Person values
    ('1','John','Norris','04-10-1982','jn@abc.com');
insert into Person values
    ('2','Emily','Parker','15-06-1980','ep@xyz.com');
insert into Person values
    ('3','Agnes','Smith','09-05-1960','as@cde.com');
```

Example 4.13: Create a view named *V_Person* which stores names and birth dates of people who are older than 50.

```
CREATE VIEW V_Person
AS SELECT firstName,lastName,birthDate
   FROM Person
   WHERE DATEDIFF(year, birthDate, GETDATE())> 50
WITH CHECK OPTION

SELECT * FROM V_Person   --the view contains only one row
```

Example 4.14: Working with *check option*.

```
UPDATE V_Person
SET birth_date='04-10-1999'
WHERE person_id=1
```

```
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.
```

The following *update* is valid:

```
UPDATE V_Person
SET birthDate='04-10-1953'
```

```
WHERE person_id=1
```

Example 4.15: Create a view with *schemabinding*

```
DROP VIEW V_Person

CREATE VIEW V_Person
WITH SCHEMABINDING
AS SELECT firstName,lastName,birthDate
   FROM dbo.Person
   WHERE DATEDIFF(year, birthDate, GETDATE())> 50
WITH CHECK OPTION
```

The *with schemabinding* clause protects from the following operation:

```
ALTER TABLE Person
DROP COLUMN birthDate
```

The following statement is valid (the column email does not occur in the view definition):

```
ALTER TABLE Person
DROP COLUMN email
```

4.6 Indices

Indices ([GMUW08, RG03]) are redundant database objects used to speed up select statements. An index can be considered as two columns table. The first column stores so called key values (e.g. last names) and the second one contains non-key values (e.g. row locators). Key values does not have to be unique and are ordered.

To increase the performance indices are organized as balanced trees (B-tree) [GMUW08, RG03]. The index has three levels. The first level is called *root*. The second one is called *intermediate level*. The last level is called *leaf level*.

SQL Server provides the following index types [BOL12]:

- clustered index – it stores the whole rows of the table (date pages) in the leaf nodes.
    - A table which does not have a clustered index is called *heap*.
    - A table which have a clustered index is called *clustered table*.
- nonclustered index – it stores key values and non-key values (row locators) in nodes on each level. The key values are ordered.
- unique index – it is a clustered or nonclustered index whose key values are unique.
- index with included columns – it is a nonclustered index whose leaf nodes store also values from other non-key columns.

- filtered index – it is a nonclustered index which involve the selection conditions.
- full-text index – it supports queries aimed at finding sophisticated phrases in various texts.
- spatial index – it is dedicated for spatial data.
- XML index – it supports operations performed on the XML binary large objects.

4.6.1 Creating indexes

The basic CREATE INDEX statement is as follows [BOL12]:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name

    ON table_name ( column_name [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <selection_predicate> ]
```

UNIQUE – it creates a unique index on a table.

CLUSTERD - it creates a clustered index on a table.

NONCLUSTERED - it creates a nonclustered index on a table (*NONCLSTERD* is the default option).

index_name – it is the index name.

table_name – it is the table name.

column_name – it is the column of table_name on which the index is created. ASC sorts the index keys (values stored in column_name) in ascending order. DESC sorts these values in descending order. ASC is the default sort order.

INCLUDE column_name – it specifies that values from the *column_name* column will be stored in the leaf nodes of the clustered index.

WHERE selection_predicate – it causes that only the rows which match the selection predicate will be used to form a nonclustered index.

Sample indexes

Consider the following *Student* table:

```
CREATE TABLE Student
(
   stud_id int IDENTITY CONSTRAINT PK_Stud_id
                        PRIMARY KEY NONCLUSTERED,
   first_name varchar(25) NOT NULL,
   last_name varchar(30) NOT NULL,
   birth_date date NOT NULL,
   gender char(1) CONSTRAINT CH_Stud_gen
                              CHECK(gender='M' OR gender='F'),
   email varchar(40) CONSTRAINT U_Stud_Email UNIQUE,
   year int not null
)
```

The table *Student* has two indexes:

- unique nonclustered index on the *stud_id* column
- unique nonclustered index on the *email* column

Example 4.16: Create a clustered index on the *last_name* column of *Student*.

```
CREATE CLUSTERED INDEX idx_Stud_lName

ON Student(last_name)
```

Example 4.17: Create an index with included columns which allow to covers the following query:

```
SELECT stud_id, first_name, last_name
FROM Student
WHERE year > some_value

CREATE NONCLUSTERED INDEX idx_Stud_year
ON Student(year)
INCLUDE (stud_id,first_name,last_name)
```

4.7 Altering indexes

Rebuilding indexes

Database administrators should periodically rebuild indexes to defragment data pages occupied by them.

Example 4.18: Rebuild the *idx_Stud_year* index from Example 4.17. Set *fillfactor* to 15%.

```
ALTER INDEX idx_Stud_year
ON Student
REBUILD
WITH (FILLFACTOR = 15)
```

Example 4.19: Disable the *idx_Stud_year* index

```
ALTER INDEX idx_Stud_year
ON Student
DISABLE
```

Example 4.20: Enable the *idx_Emp_lName_Sal* index

```
ALTER INDEX idx_Stud_year
ON Student
REBUILD
```

4.8 Working with indices

Perform the following steps:

1. Create the table *big1* in the *demoDB* database:

```
create table big1(id int identity,data char(1000) default 'big1')
set nocount on
insert into big1 default values
go 10000
```

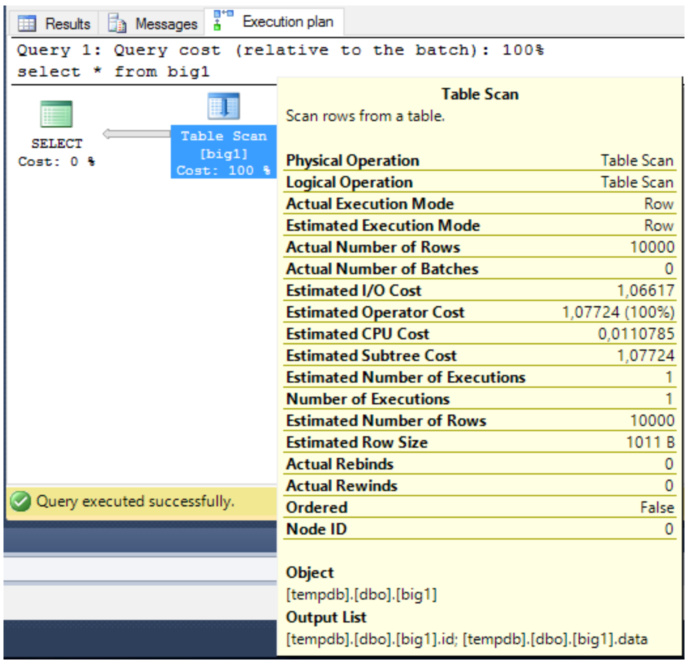
2. Enable Input/Output statistics:

```
set statistics io on
```

3. Include Actual Execution Plan (Ctrl+M) and run the query (1):

```
select * from big1 (1)
```

4. Examine the following query execution plan of (1) and the I/O statistics.



```
Query 1: Query cost (relative to the batch): 100%
select * from big1
```

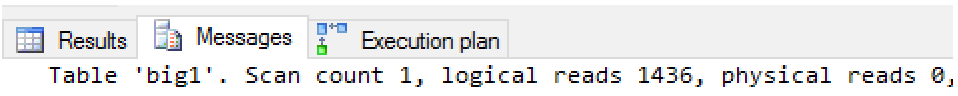
Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	10000
Actual Number of Batches	0
Estimated I/O Cost	1,06617
Estimated Operator Cost	1,07724 (100%)
Estimated CPU Cost	0,0110785
Estimated Subtree Cost	1,07724
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	10000
Estimated Row Size	1011 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object
[tempdb].[dbo].[big1]
Output List
[tempdb].[dbo].[big1].id; [tempdb].[dbo].[big1].data

Query executed successfully.

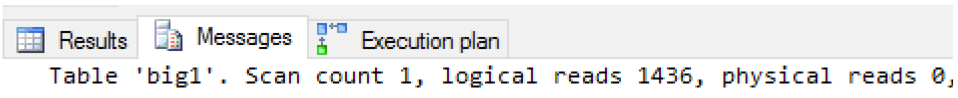


```
Table 'big1'. Scan count 1, logical reads 1436, physical reads 0,
```

5. Run the query (2):

```
select * from big1 (2)
where id=30
```

6. Examine the following I/O statistics of (2).



```
Table 'big1'. Scan count 1, logical reads 1436, physical reads 0,
```

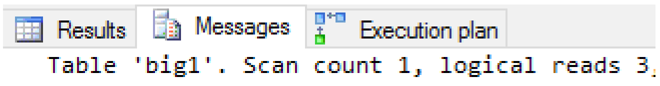
7. Compare execution plans of (1) and (2) and their I/O statistics.

8. Create a non-clustered index on the *id* column of *big1*.

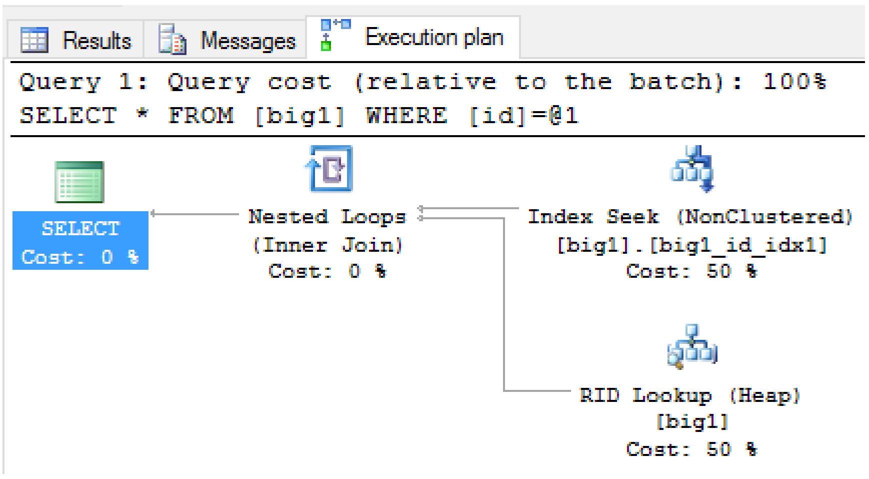
```
create index big1_id_idx1 on big1(id)
```

9. Run the queries (1) and (2).

10. The execution plan of (1) should not change. The plan and I/O statistics for the query (2) should be the following:



```
Table 'big1'. Scan count 1, logical reads 3,
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [big1] WHERE [id]=@1
```

11. Exclude Actual Execution Plan (Ctrl+M) and disable I/O statistics (*set statistics io off*).

12. Run the following statements:

```
create table big2(id int identity,number int, data char(1000) default 'big2')
set nocount on
insert into big2 values(convert(int,rand()*10),'introDB')
go 10000
select number,count(*)
from big2
group by number
```

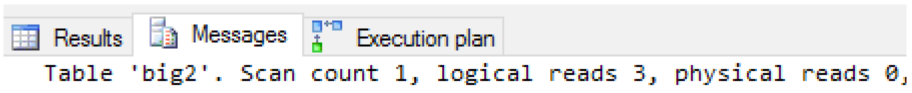
13. Create the following indices:

```
create index big2_idx1 on big2(id)
create index big_idx2 on big2(number)
```

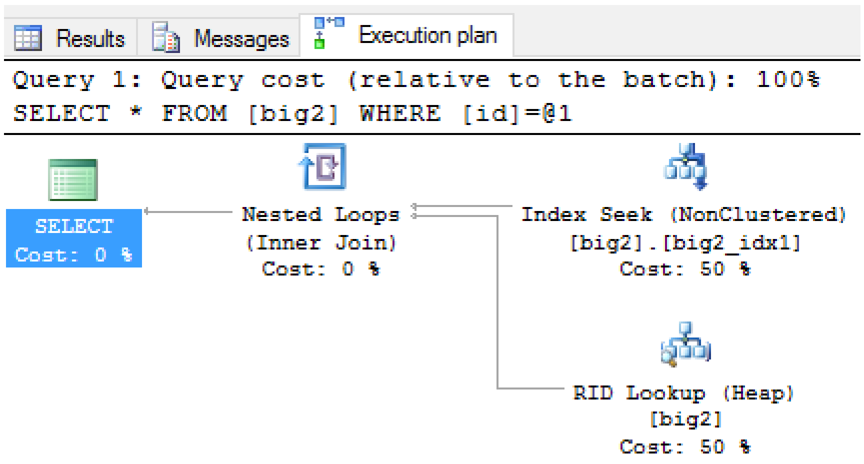
14. Include Actual Execution Plan (Ctrl+M) and enable I/O statistics.

15. The execution plan of the query (3) and its I/O statistics should be the following:

```
select * from big2 (3)
where id=3
```



```
Table 'big2'. Scan count 1, logical reads 3, physical reads 0,
```

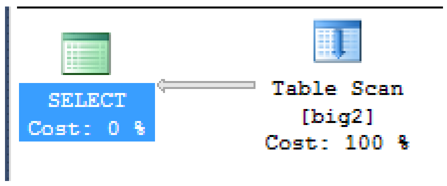


```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [big2] WHERE [id]=@1
```

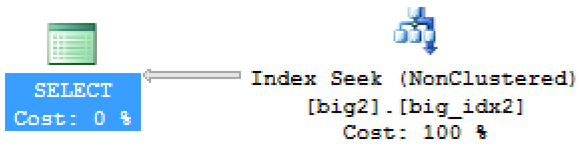
15. Why does the query (4).

```
select * from big2 (4)
where number=8
```

has the following execution plan?



16. Change the query (4) to obtain the following execution plan.



17. Exclude Actual Execution Plan (Ctrl+M) and disable I/O statistics (*set statistics io off*).

18. Run the following statements:

```
create table big3(id int identity,number1 int,number2 int, data char(1000) default 'big3')
set nocount on
insert into big3 values(convert(int,rand()*500),convert(int,rand()*10),'introDB')
go 10000
```

19. Create the following indexes:

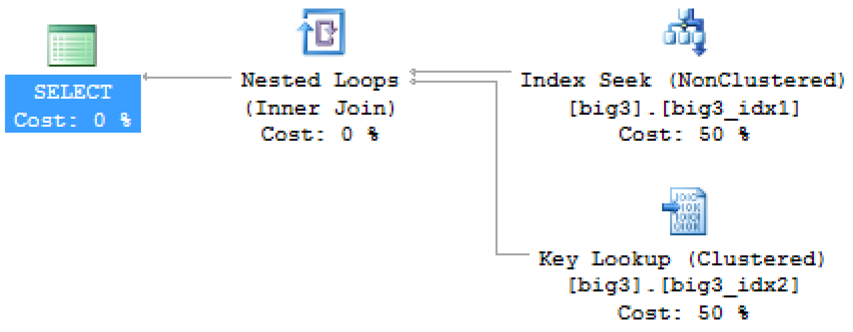
```
create nonclustered index big3_idx1 on big3(id)
create clustered index big3_idx2 on big3(number1)
```

20. Include Actual Execution Plan (Ctrl+M) and enable I/O statistics (*set statistics io on*).

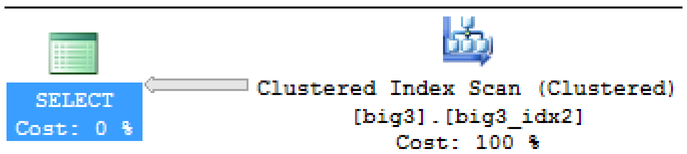
21. Run the following query (5):

```
select * from big3 (5)
where id=10
```

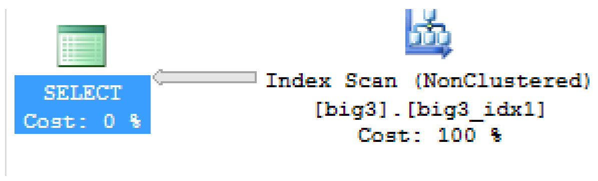
22. Interpret an execution plan of (5). It should have the following form:



23. Write a query involving the table *big3* which can have the following execution plan.



24. Write a query involving the table *big3* which can have the following execution plan.



25. Display information about indices of *big3*.

```
select * from sys.indexes
where object_id=object_id('big3','U');
```

26. Display detailed information about the indexes of *big3*.

```
select * from sys.dm_db_index_physical_stats(db_id('demoDB'),
object_id('big3','U'),null,null,'detailed')
```

27. Note the values stored in the *avg_fragmentation_in_percent* column of *sys.dm_db_index_physical_stats* for the the *big3_idx2* index.

28. Run the the following update statement and check the fragmentation percent (It should increase):

```
update *big3
set number1=number1*2-50
```

29. Rebuild the *big3_idx2* index. It should decrease the fragmentation percent.

5 Query optimization

There are two common approaches to query optimization: rule-based optimization and cost-based optimization. The first approach is based on query rewriting. Query optimizer tries to rewrite a given query to the semantically equivalent form which should guarantees better performance. For instance the following nested query

```
select jh.step_id,jh.step_name
from msdb.dbo.sysjobhistory jh
where jh.run_status=1 and jh.job_id in
                             (select j.job_id from msdb.dbo.sysjobs j)
```

can be rewritten to the following semantically equivalent form

```
select jh.step_id,jh.step_name
from msdb.dbo.sysjobhistory jh join msdb.dbo.sysjobs j
     on j.job_id=jh.job_id
where jh.run_status=1
```

In the second query the join operator allows to eliminate nested query. Contemporary database management systems are designed to efficiently process queries with joins rather than complex nested queries. Query optimizer usually tries to rewrite nested queries.

The cost-based optimization deals among others with the following issues:

- Choose the best indexes.
- Choose the best join order.
- Choose the best join operation (e.g. nested loops, merge join, hash join).

The query optimizer makes a decision based on statistics. Database administrators should regularly rebuild the database statistics. If they are out-of date the query optimizer can select inefficient execution plan for a given query. It is also important to regularly rebuild indexes.

DBA can create so called maintenance plan in SQL Server Management Studio which allows to rebuild database statistics and indexes:

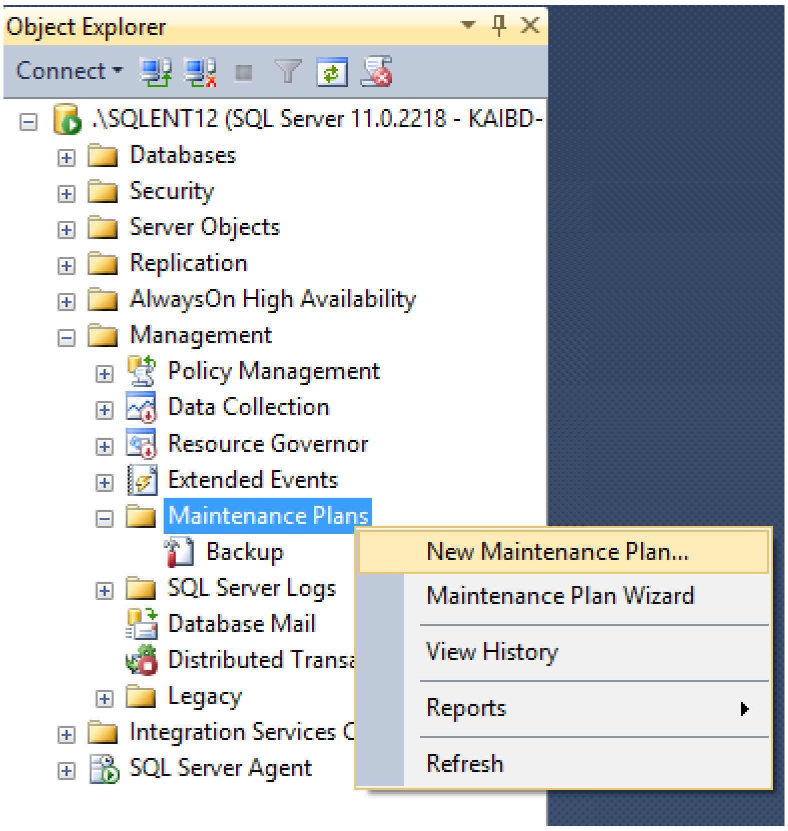
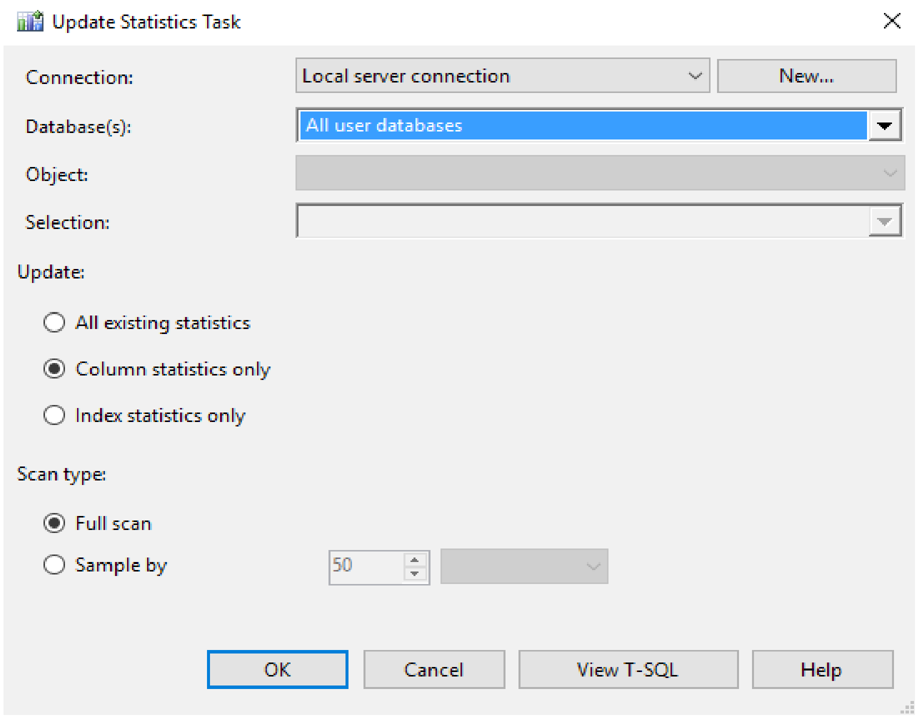


Fig. 5.1. Creating a new maintenance plan


Update Statistics Task

Connection: Local server connection

Database(s): All user databases

Object:

Selection:

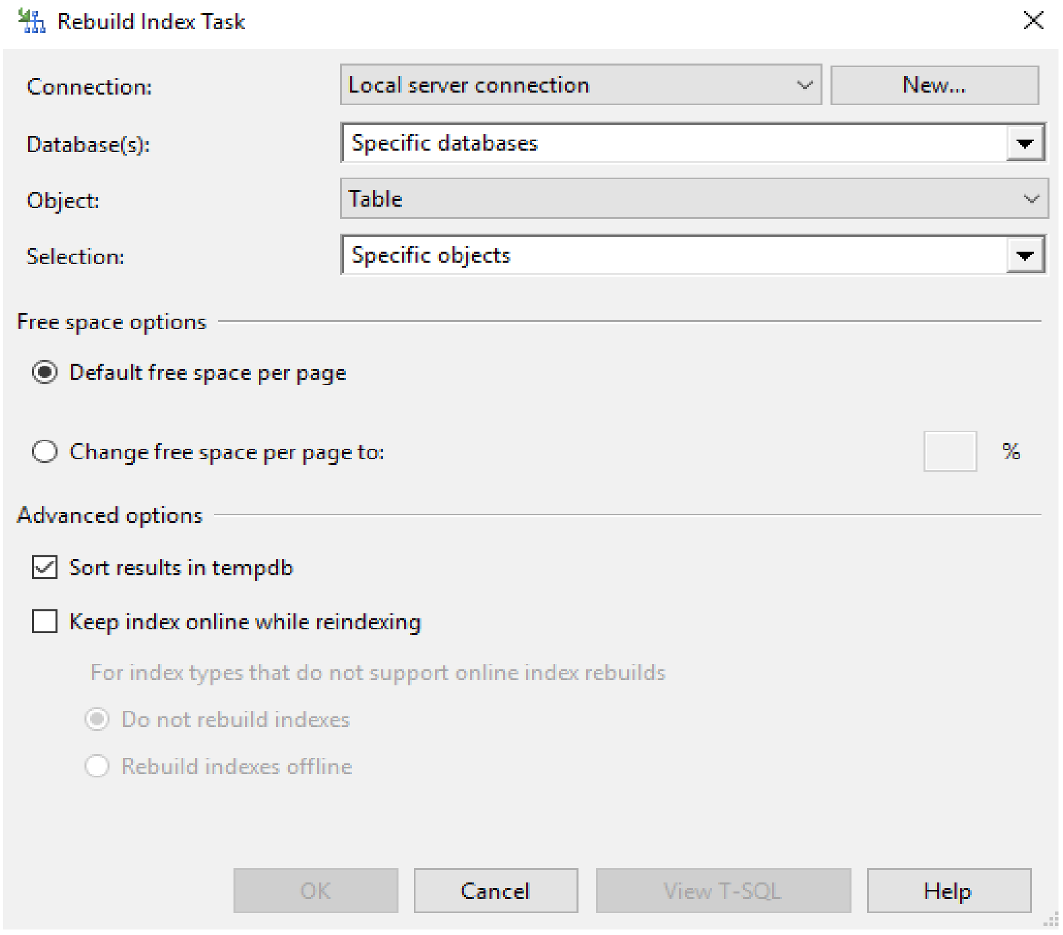
Update:

- ○ All existing statistics
- ● Column statistics only
- ○ Index statistics only

Scan type:

- ● Full scan
- ○ Sample by 50

Fig. 5.2. Configuring the update statistics task


Rebuild Index Task

Connection: Local server connection

Database(s): Specific databases

Object: Table

Selection: Specific objects

Free space options

- ● Default free space per page
- ○ Change free space per page to: %

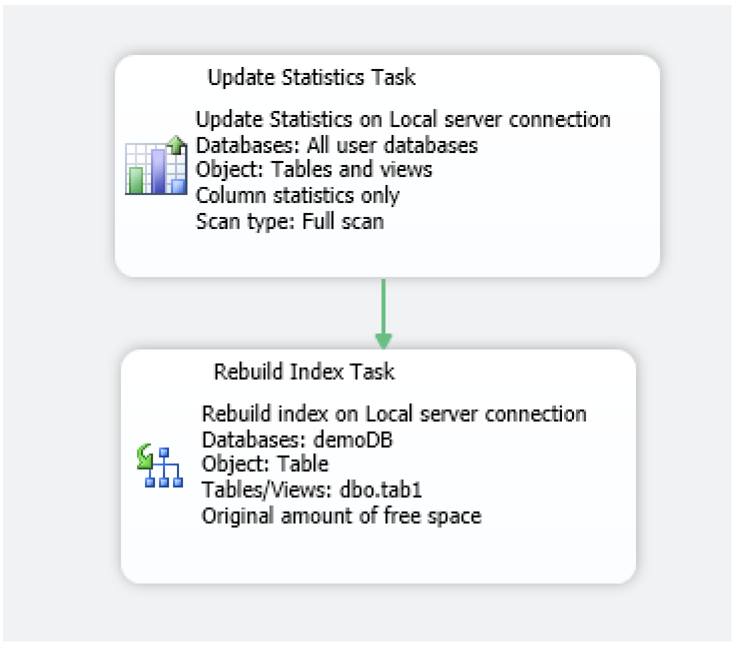
Advanced options

- ☑ Sort results in tempdb
- ☐ Keep index online while reindexing

For index types that do not support online index rebuilds

- ● Do not rebuild indexes
- ○ Rebuild indexes offline

Fig. 5.3. Configuring the rebuild index task


Update Statistics Task

Update Statistics on Local server connection
Databases: All user databases
Object: Tables and views
Column statistics only
Scan type: Full scan

Rebuild Index Task

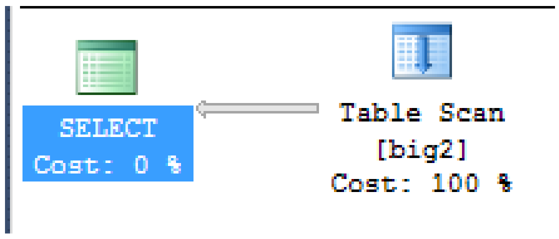
Rebuild index on Local server connection
Databases: demoDB
Object: Table
Tables/Views: dbo.tab1
Original amount of free space

Fig. 5.4. Final maintenance plan

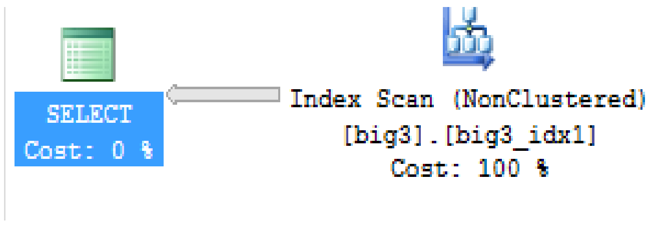
5.1 Main elements of SQL Server execution plans

SSMS allows to display execution plans for user queries. Such plans can have the following elements:

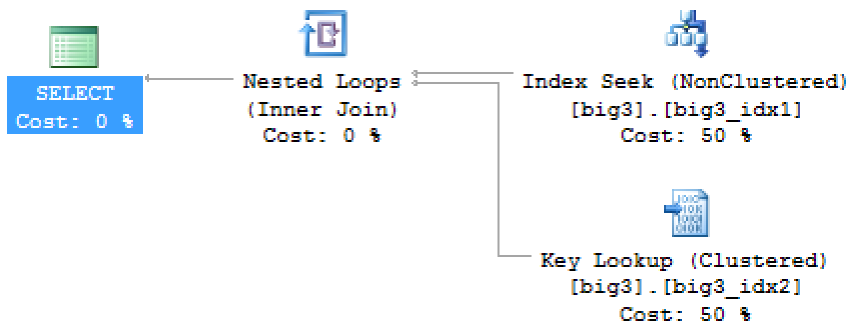
- *Table Scan* – it performs a full table scan.



- *Index Scan* - it performs a full index scan.

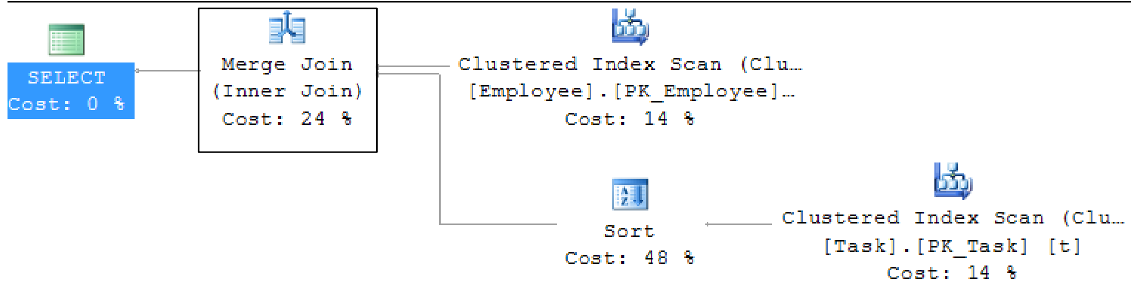


- *Nested Loops* – it performs the join operation using a nested loops join.



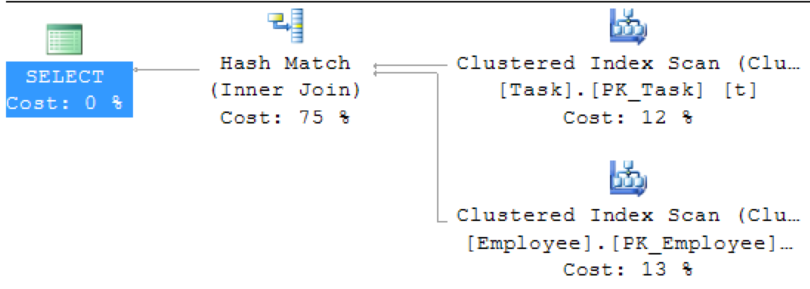
- *Merge Join* – it performs the join operation using a sort merge join.

```
SELECT * FROM Employee e JOIN Task t ON e.emp_id=t.e_id OPTION (MERGE JOIN);
```

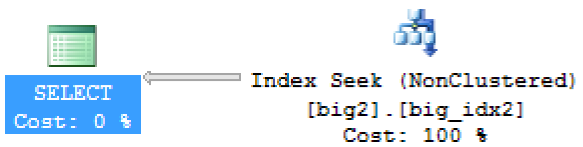


- *Hash Join* – it performs the join operation using a hash join.

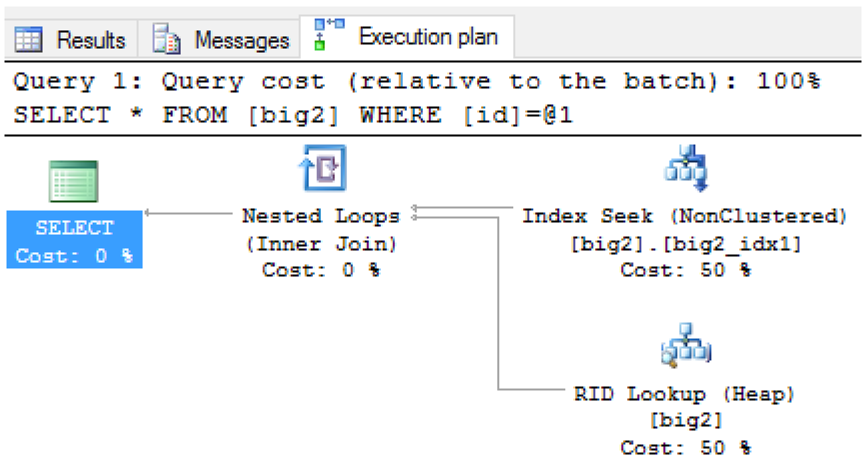
```
SELECT * FROM Employee e JOIN Task t ON e.emp_id=t.e_id OPTION (HASH JOIN);
```



- *Nonclustered index seek* – it searches a non-clustered index and returns the proper row locator (RID).

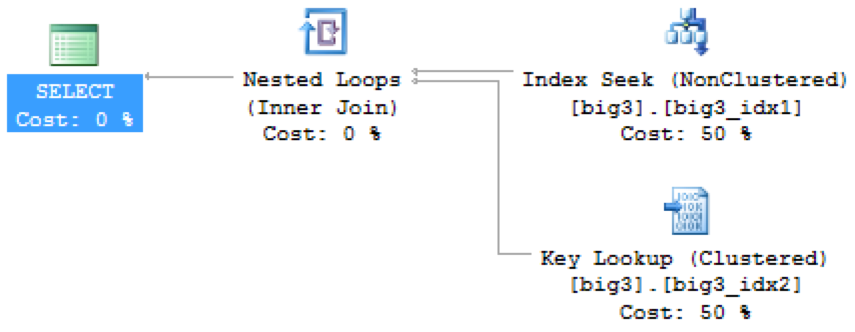


- *RID Lookup* (*Heap*) – it takes a row locator (RID) and searches the heap to get the row having this RID. RID has three elements: file identifier, page number, the row number on the page.



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [big2] WHERE [id]=@1
```

- *Key Lookup* (*Clustered*) – it is assumed that the *Tab3* table has a clustered index. The *Key Lookup* acts on a row locator returned by *Index Seek*. Next it searches the clustered index to get the row having this locator.



6 Stored procedures, functions and triggers

6.1 Conditional statements

SQL Server provides two conditional statements: *if..else* and *case*.

IF..ELSE statement [BOL12]:

```
IF boolean_expression
     { statement1 }
[ ELSE
     { statement2 }]
```

Example 6.1: Sample *if..else* statements.

```
DECLARE @year int;
SELECT @year = DATEPART(year,GETDATE())
IF(@year=2015)
```

```
   PRINT 'We have 2015 '
ELSE
  IF(@year=2016)
    PRINT 'We have 2016'
  ELSE
    IF(@quarter=2017)
      PRINT 'We have 2017'
    ELSE PRINT 'We have year less then 2015 or greater than 2017';
```

The case statements has two forms. The first forms is as follows [BOL12]:

```
CASE input_expression
     WHEN expr1 THEN result1
      [WHEN expr2 THEN result2
      ...
      WHEN exprN THEN resultN
     ELSE expr]
END
```

Example 6.2: Sample *case* statement (the first form).

```
SELECT CASE DATEPART(year,GETDATE())
     WHEN  2015 THEN 'We have 2015'
      WHEN 2016 THEN 'We have 2016'
      WHEN 2017 THEN 'We have 2017'
      ELSE 'We have year less then 2015 or greater than 2017'
END
```

The second form of the case statement is as follows [BOL12]:

```
CASE

     WHEN boolean_expr1 THEN result1
      [WHEN boolean_expr1expr2 THEN result2
      ...
      WHEN boolean_expr1exprN THEN resultN
     ELSE expr]
END
```

6.2 Loops

A loop enables to execute a block of statements a given number of times. SQL Server provides the classical *while* loop statement [BOL12]:

```
WHILE boolean_expression
BEGIN
    statements
END
```

Example 6.3: Sample while statement.

```
declare @cnt int=1

while @cnt<20
begin
```

```
      print cast(@cnt as  varchar(10))
      set @cnt=@cnt+1
end
```

6.3 Cursors

A cursor is applied to read an area in which SQL Server stores the result of a given query. The basic syntax to declare a cursor is as follows:

```
DECLARE cursor_name CURSOR
FOR select_statement
```

Example 6.4: Sample cursor

The following cursor returns information about the last backups of the *demoDB* database.

```
declare @row varchar(800)
declare @type char(1)
declare demoDB_backup_cursor cursor for
  select bmf.physical_device_name , bs.type
    from msdb.dbo.backupset bs join msdb.dbo.backupmediaset bms on
bs.media_set_id=bms.media_set_id
       join msdb.dbo.backupmediafamily bmf on
bmf.media_set_id=bms.media_set_id
    where bs.database_name='demoDB' and bs.backup_finish_date >= (select
max(backup_finish_date) from msdb.dbo.backupset where type='D')
    order by bs.backup_set_id asc

OPEN demoDB_backup_cursor;
FETCH NEXT FROM demoDB_backup_cursor INTO @row,@type;
WHILE @@FETCH_STATUS=0
    BEGIN
     IF (@type='D')
     print 'Full backup -'+@row
    ELSE IF(@type='I')
      print  'Differential backup -'+@row
    ELSE
      print 'Log backup -'+@row
    FETCH NEXT FROM demoDB_backup_cursor INTO @row,@type;
    END
CLOSE  demoDB_backup_cursor
DEALLOCATE  demoDB_backup_cursor
```

6.4 Stored procedures

A stored procedure is a sequence of T-SQL statements stored on the server.

The basic syntax of the *CREATE PROCEDURE* statement is as follows [BOL12]:

```
CREATE { PROC | PROCEDURE } procedure_name
[ @parameter1 data_type, @parameter2 data_type,... ]
AS sql_statements
```

The *ALTER PROCEDURE* and *DROP PROCEDURE* statements allow to modify and remove procedures, correspondingly [BOL12].

```
ALTER{ PROC | PROCEDURE } procedure_name
[ @parameter1 data_type, @parameter2 data_type,... ]
AS sql_statements

DROP{ PROC | PROCEDURE } procedure_name
```

Example 6.5. Creating a procedure without parameters.

```
CREATE PROCEDURE sp_current_date
AS SELECT getdate()
```

Example 6.6. Invoking the procedure.

```
EXEC sp_current_date
```

System stored procedures

SQL Server provides many system stored procedures. They are mainly applied for administrative purposes.

Example 6.7. Invoking the system procedures.

```
--returns information about current users, sessions, and
--processes in a server instance
EXEC sp_who;

--provides information about all databases
EXEC sys.sp_helpdb;
```

Procedures with parameters

Example 6.8. Creating and invoking a procedure with an input parameter.

```
CREATE PROCEDURE getDbFiles @dbName varchar(40)
AS
BEGIN
   SELECT *
   FROM sys.master_files
   WHERE database_id=db_id(@dbName)
END

exec getDbFiles 'demoDB'
```

Example 6.9. Creating and invoking a procedure with an input and output parameter.

```
CREATE PROCEDURE getDbFilesAndTotalSize @dbName varchar(40), @sizeInMB
float out
AS
BEGIN
   SELECT *
   FROM sys.master_files
   WHERE database_id=db_id(@dbName)
   select @sizeInMB = sum(size)/126.0 from sys.master_files where
database_id=db_id(@dbName)

END

declare @sizeInMB float
exec getDbFilesAndTotalSize 'demoDB',@sizeInMB out
select @sizeInMB size
```

6.5 Functions

A function is a sequence of T-SQL statements kept on the server. It can take parameters and must return some value.

Scalar functions

The basic syntax is as follows [BOL12]:

```
CREATE FUNCTION function_name
   ( [ @parameter1 data_type, @parameter2 data_type,... ] )
RETURNS data_type
[ AS ]
BEGIN
       T-SQL_statements
       RETURN scalar_expression
END
```

The ALTER FUNCTION and DROP FUNCTION statements allow to modify and remove functions.

Example 6.10. Sample scalar function which takes one parameter.

```
CREATE FUNCTION GetTotalDbSize(@dbName varchar(30)) RETURNS FLOAT
BEGIN
DECLARE @size float;
SELECT @size = SUM(size)/126.0
FROM sys.master_files WHERE database_id=db_id(@dbName)
RETURN @size;
END
--Invoking the function
SELECT dbo.GetTotalDbSize('demoDB')
```

Table-valued functions

There are two types of these functions [BOL12]:

- inline table-valued functions
- multi-statement table-valued functions

Inline table-valued functions

The basic syntax is as follows [BOL12]:

```
CREATE FUNCTION function_name
   ( [ @parameter1 data_type, @parameter2 data_type,... ] )
RETURNS TABLE
[ AS ]
       RETURN select_statement
```

Example 6.11. Sample inline table-valued function with one parameter.

```
CREATE FUNCTION GetDeptsByAvgSal(@recovery_model varchar(50))
RETURNS TABLE
AS
RETURN
      select name,create_date from sys.databases
       where recovery_model_desc=@recovery_model

--invoking the function
select * from GetDeptsByAvgSal('FULL')
```

Multi-statement table-valued functions

The basic syntax is as follows [BOL12]:

```
CREATE FUNCTION function_name

   ( [ @parameter1 data_type, @parameter2 data_type,... ] )
RETURNS @table_variable TABLE table_defintion
[ AS ]
BEGIN
       T-SQL statements
       RETURN
END
```

Example 6.12. Sample multi-statement table-valued function with one parameter.

```
CREATE FUNCTION GetDBs()

RETURNS @dbs TABLE
        (db_id int,
       name varchar(80),
       create_date datetime,
       recovery_model varchar(50))
AS
BEGIN
  INSERT INTO @dbs
  SELECT database_id,name,create_date,recovery_model_desc
  FROM sys.databases
  RETURN
END
--Invoking the function
SELECT * FROM GetDBs()
```

6.6 Triggers

A trigger is a kind of stored procedure which SQL Server runs automatically when the proper even occurs. The following triggers are supported:

- DML triggers – they are run when users perform DML statements (*INSERT*, *UPDATE*, *DELETE*) on tables or views.
- DDL triggers - they are run when users perform various DDL statements (e.g. *CREATE*, *ALTER*, *DROP*)
- Logon triggers – they are run when users logon to the SQL Server instance

DML triggers

The syntax is as follows [BOL12]:

```
CREATE TRIGGER trigger_name
ON { table_name | view_name }
{ AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS T-SQL_statements
```

Example 6.13. Using a DML trigger to ensure that only positive numbers can be inserted to a table called *positiveNumbers*.

```
create table positiveNumbers(number int)

CREATE TRIGGER TR_Positive_Numbers
ON positiveNumbers
AFTER INSERT AS BEGIN
  IF EXISTS( SELECT 1
             FROM inserted AS i
            WHERE i.number<=0)
  BEGIN
     PRINT 'Only positive numbers are allowed'
     ROLLBACK
  END;
END;

insert into positiveNumbers values(1)
insert into positiveNumbers values(-1)
```

DDL triggers

The basic syntax is as follows [BOL12]:

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
{ AFTER } { event_type } [ ,...n ]
AS
T-SQL_statements
```

ALL SERVER – the trigger scope involves the current instance.

DATABASE – the trigger scope involves the current database.

event_type – it defines DDL events (e.g. CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE PROCEDURE, ALTER PROCEDURE, CREATE INDEX, ALTER INDEX, etc.

Example 6.14. Using a DDL trigger to prevent modification or deletion of indexes.

create table tab1(id int primary key)

```
create procedure getDate as

begin
select getdate()
end


CREATE TRIGGER trgPreventAlter_Drop_Procedures
ON DATABASE
FOR ALTER_PROCEDURE, DROP_PROCEDURE
AS
BEGIN
   PRINT 'You cannot change or delete procedures';
   ROLLBACK;
END;


drop procedure getDate
```

Logon triggers

The basic syntax is as follows [BOL12]:

```
CREATE TRIGGER nazwa_wyzwalacza
 ON ALL SERVER
 AFTER LOGON AS
 T-SQL_statements
```

Example 6.15. Using a LOGON trigger to prevent logon as *sa*.

```
CREATE TRIGGER trgPreventLogonSA
ON ALL SERVER
FOR LOGON AS
 BEGIN
   IF
    (ORIGINAL_LOGIN()= 'sa'
        AND APP_NAME() = 'Microsoft SQL Server Management Studio')
   ROLLBACK;
END;
```

7 Security and user authorization in SQL Server

SQL Server introduces the following entities to provide security and access control mechanism [BOL12]:

- *securables* – they represent entities to which access can be controlled,

- *principals* – they represent entities which can gain access to *securables*.

Securables occur on three levels:

- instance level – databases, logins, server roles, endpoints.
- database level – database users, schemas, database roles, certificates, keys, etc.
- schema level – tables, views, procedures, functions, types, etc.

The following levels are adequate for *principals*:

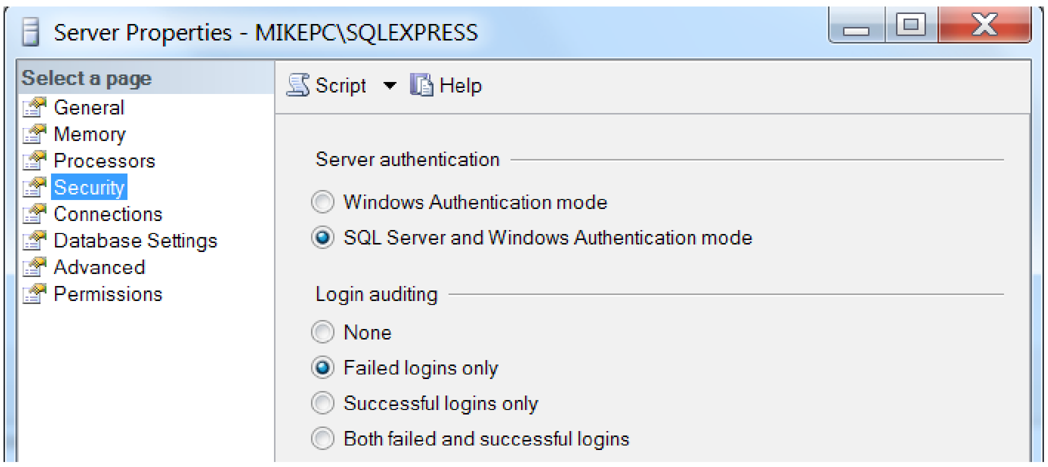
- Windows level – Windows logins and groups.
- instance level – SQL Server logins, server roles.
- database level – database users, database and application roles.

7.1 SQL Server authentication

Authentication is aimed at checking an identity. During this process the system verifies whether a user is who he claims he is. SQL Server provides two authentication modes [BOL12]:

- Windows authentication mode – in this mode SQL Server accepts credentials obtained from Windows.
- Mixed mode (Windows and SQL Server authentication mode) – in this mode a user identity can be checked either by SQL Server or by Windows operating systems.

SQL Server Management Studio (SSMS) also allows to manipulate the authentication mode (Right-click on the server instance – Select *Properties* and then select *Security*):



Server authentication
- Windows Authentication mode
- [✓ SQL Server and Windows Authentication mode]

Login auditing
- None
- [✓ Failed logins only]
- Successful logins only
- Both failed and successful logins

Figure 7.1: Setting SQL Server authentication mode.

Taking the above authentication modes into account, there are two main types of logins applied to connect to a SQL Server instance:

- Windows logins – they are based on Windows accounts or groups.
- SQL Server logins – they are managed directly by SQL Server.

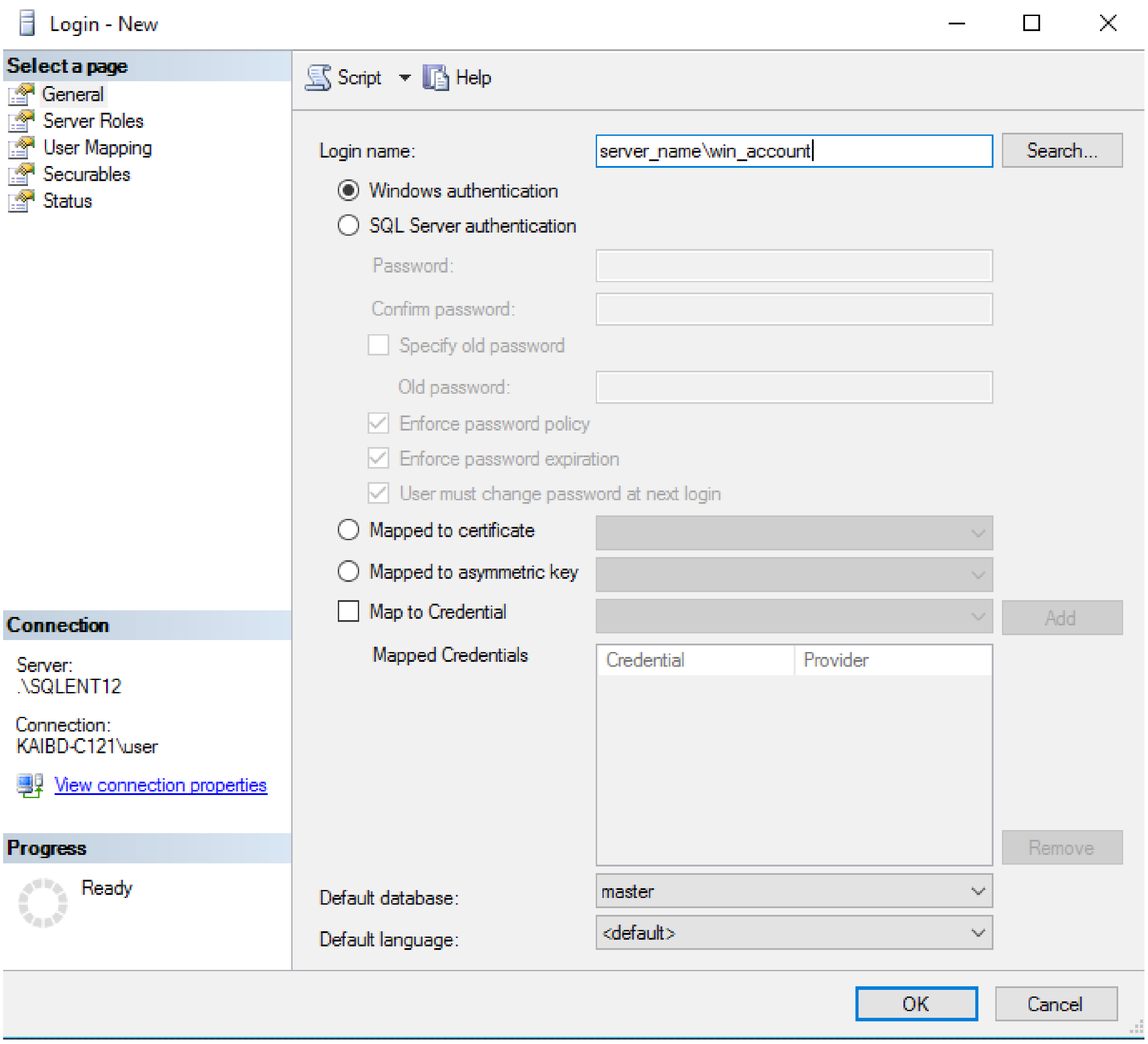
Windows logins

The following syntax is used to create a Windows login [BOL12]:

```
CREATE LOGIN [host\login_name] FROM WINDOWS
[WITH DEFAULT_DATABASE=database_name]
```

Example 7.1: Creating Windows logins.

```
-- win_account is a Windows account name
CREATE LOGIN [server_name\win_account] FROM WINDOWS
```



Login name: server_name\win_account

(● Windows authentication) ○ SQL Server authentication

Password:

Confirm password:

☐ Specify old password

Old password:

☑ Enforce password policy

☑ Enforce password expiration

☑ User must change password at next login

○ Mapped to certificate

○ Mapped to asymmetric key

☐ Map to Credential

Mapped Credentials

Credential	Provider

Server:
.\SQLENT12

Connection:
KAIBD-C121\user

Ready

Default database: master

Default language: <default>

```
-- domain_name is a Windows domain name,helpdesk is a Windows
group name
CREATE LOGIN [domain_name\helpdesk] FROM WINDOWS
```

SQL Server logins

The basic syntax used to create SQL Server logins is as follows [BOL12]:

```
CREATE LOGIN login_name
WITH PASSWORD='l_password' [MUST_CHANGE]
[ , CHECK_EXPIRATION=[ON | OFF]]
[ , CHECK_POLICY=[ON | OFF]]
[ , DEFAULT_DATABASE=database_name]
[ , SID=some_guid]
```

login_name – it represents the name of the SQL Server login.

l_password – it is the password for the login.

MUST_CHANGE – it forces a password change during the first login.

CHECK_EXPIRATION – it forces a password expiration policy for the login.

CHECK_POLICY – it forces the Windows password policy.

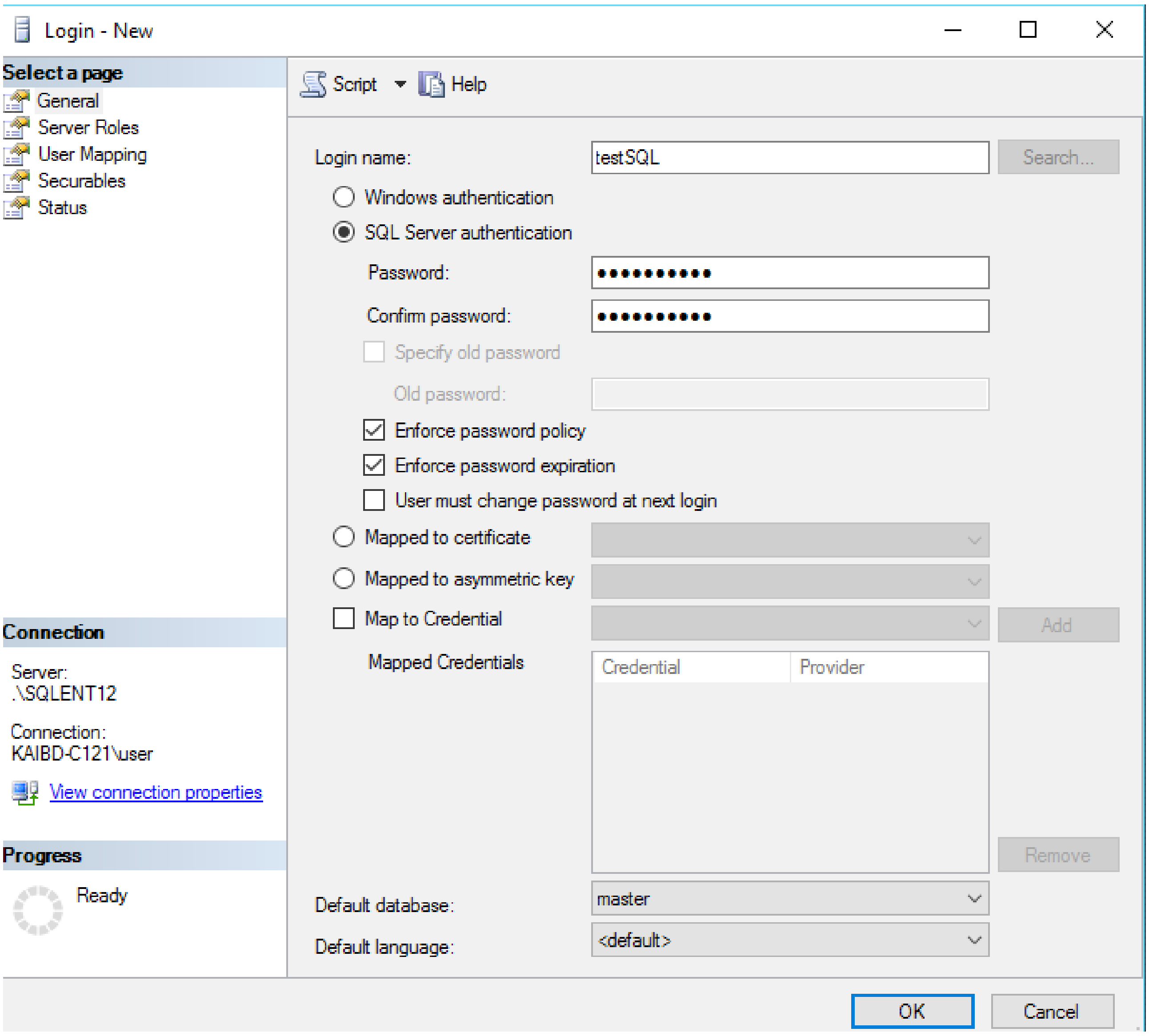
SID some_guid – it specifies the globally unique identifier (GUID) for the login.

Example 7.2: Creating sample SQL Server login.

```
CREATE LOGIN testSQL
WITH PASSWORD='fhfh838GG_A!',
CHECK_EXPIRATION= ON, CHECK_POLICY=ON
```

Example 7.3: Creating logins in SSMS.

SSMS allows to create both Windows logins and SQL Server logins. Right-clik on the *Security* folder in *Object Explorer* – Select *New* and then select Login – Fill the *General* page.



Login - New

Select a page: General, Server Roles, User Mapping, Securables, Status

Login name: testSQL

( ) Windows authentication
(●) SQL Server authentication

Password: ●●●●●●●●●●
Confirm password: ●●●●●●●●●●
☐ Specify old password
Old password:
☑ Enforce password policy
☑ Enforce password expiration
☐ User must change password at next login
( ) Mapped to certificate
( ) Mapped to asymmetric key
☐ Map to Credential
Mapped Credentials: Credential | Provider

Default database: master
Default language: <default>

Connection
Server: .\SQLENT12
Connection: KAIBD-C121\user
View connection properties

Progress
Ready

Altering logins

The ALTER LOGIN statement is used to modify both Windows logins and SQL Server logins [BOL12]:

```
ALTER LOGIN login_name
ENABLE | DISABLE
| WITH [NAME=new_login_name]
[ , DEFAULT_DATABASE=database_name]
[ , PASSWORD='new_password' [MUST_CHANGE]
[ , CHECK_EXPIRATION=[ON | OFF]]
[ , CHECK_POLICY=[ON | OFF]]
```

These clauses are valid for SQL Server logins

Example 7.4: Changing the login name from *loginOLD* to *loginNEW*.

```
ALTER LOGIN loginOLD NAME=loginNEW
```

Example 7.5: Changing the login password.

```
ALTER LOGIN loginNEW WITH PASSWORD='uuTRK80_!'
```

Example 7.6: Disabling a login.

```
ALTER LOGIN loginNEW DISABLE
```

Removing logins

The DROP LOGIN statement is used to remove logins:

```
DROP LOGIN login_name
```

7.2 SQL Server authorization

It is required to assign permissions on securables to principals. This process is called *authorization*. SQL Server provides several methods to implement authorization [BOL12]:

- assign logins to roles which involve the proper permissions,
- assign permissions to logins using the GRANT statement,
- create database users and associate them with logins.

The permission can be granted on the following levels:

- instance level (it is the highest level),
- database level (second level),
- schema level (third level),
- schema object level (fourth level).

Roles

Roles can be understood as containers for permissions. SQL Server supports four types of roles:

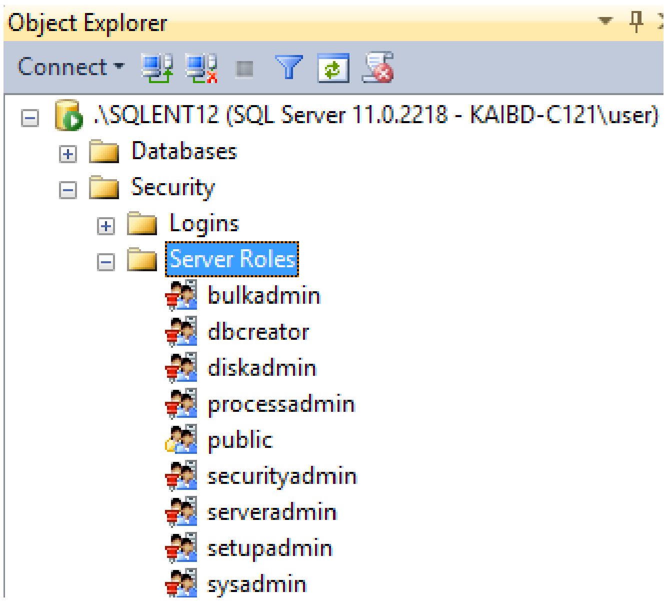
- fixed server roles,
- user-defined server roles,
- fixed database roles,
- user-defined database roles.

Fixed server roles

SQL Server provides the set of server-scoped roles. The following table presents the fixed server roles and their permissions [BOL12]:

Role	**Permission**	**Description**
sysadmin	CONTROL SERVER	It lets to perform any operations in the instance
securityadmin	ALTER ANY LOGIN	It lets to manage logins and their permissions
dbcreator	ALTER ANY DATABASE	It lets to modify (create, alter,

		drop) and restore any database
processadmin	ALTER ANY CONNECTION ALTER SERVER STATE	It lets to terminate existing processes
setupadmin	ALTER ANY LINKED SERVER	It lets to manipulate linked servers
serveradmin	ALTER ANY ENDPOINT, ALTER RESOURCES, VIEW SERVER STATE, ALTER SERVER STATE, ALTER SETTINGS, SHUTDOWN	It lets to manipulate server configuration options and shutdown the server
diskadmin	ALTER RESOURCES	It lets to manage disk files
bulkadmin	ADMINISTER BULK OPERATIONS	It lets to manage bulk operations



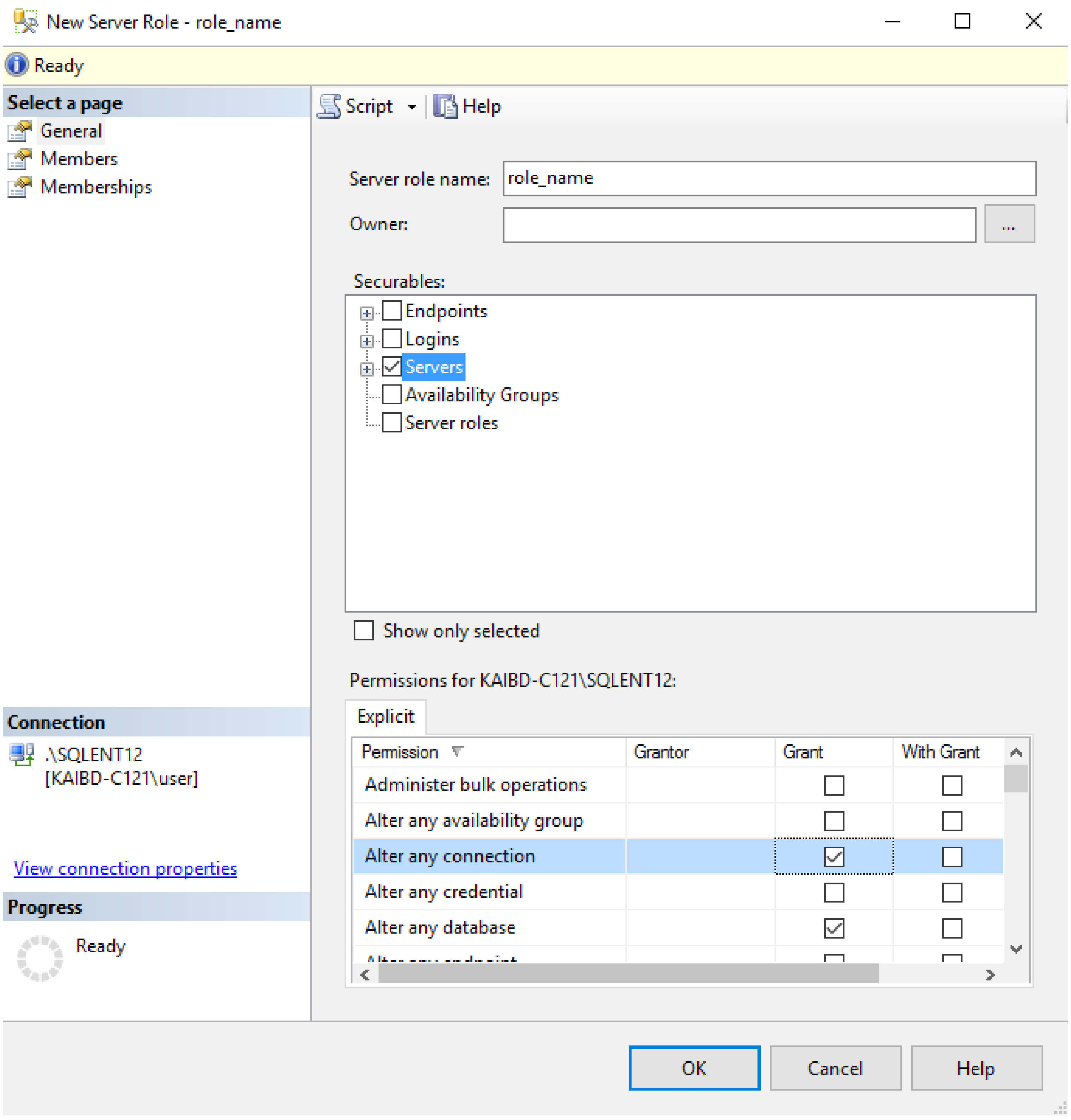
The following procedures are dedicated to fixed server roles [BOL12]:

- *sp_addsrvrolemember* '*login_name*', '*role_name*' – it causes that a login be the member of a role.
- *sp_dropsrvrolemember* '*login_name*', '*role_name*' – it removes a login from a role.
- *sp_helpsrvrolemember* '*role_name*' – it displays the members of a role.
- *sp_srvrolepermission* '*role_name*' – it displays permissions which belong to a role.
- *IS_SRVROLEMEMBER*('*role_name*' ,'*login_name*') – this function returns *true* if a login is the member of a role; otherwise returns *false*.

User-defined server roles

The CREATE SERVER ROLE statement is applied to create server roles defined by users:

```
CREATE SERVER ROLE role_name [ AUTHORIZATION role_owner ]
```



Server role name: role_name

Owner:

Securables:

- [ ] Endpoints
- [ ] Logins
- [✓] Servers
- [ ] Availability Groups
- [ ] Server roles

[ ] Show only selected

Permissions for KAIBD-C121\SQLENT12:

Explicit

Permission	Grantor	Grant	With Grant
Administer bulk operations		☐	☐
Alter any availability group		☐	☐
Alter any connection		✓	☐
Alter any credential		☐	☐
Alter any database		✓	☐

Connection: .\SQLENT12 [KAIBD-C121\user]

The following statement lets to remove user-defined server roles:

```
DROP SERVER ROLE role_name
```

Server-scoped permissions

The most common permissions at the server level are [BOL12]:

- CONTROL SERVER – it lets to perform any operations within the instance.
- ALTER ANY DATABASE – it lets to manage databases in the instance.
- CREATE ANY DATABASE – it lets to create databases.

- ALTER ANY LOGIN – it lets to manage logins.
- VIEW ANY DEFINTION – it lets to view metadata of any objects within the instance.
- ALTER TRACE – it lets to run traces against the instance.
- CONNECT SQL – it lets to connect to the instance using the proper endpoint.

The GRANT statement allows to grant permissions at the server level [BOL12]:

```
GRANT permission_name [ ,...n ] TO principal_name [ ,...n ]
[ WITH GRANT OPTION ]
```

permission – it represents a server-scoped permission name.

principal – it represents a principal name (Windows or SQL Server login, server role), which grants the permission.

WITH GRANT OPTION – this clause allows the principal to grant the permission to others.

The DENY statement denies previously granted permissions at the server level [BOL12]:

```
DENY permission_name [ ,...n ] TO principal_name [ ,...n ]
[ CASCADE ]
```

CASCADE – this option denies the permission *permission_name* from principals to which *principal_name* granted it.

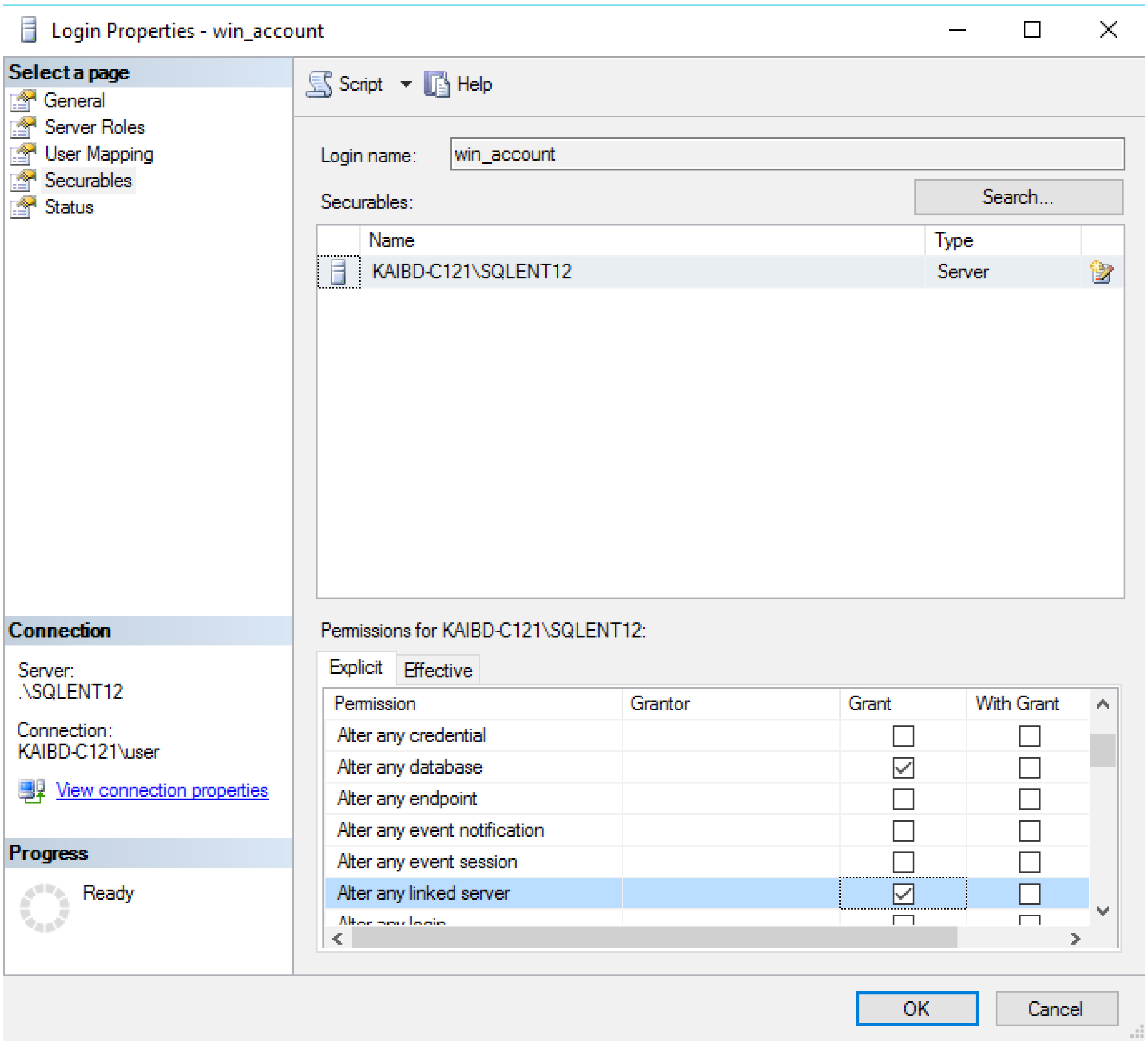
The REVOKE statement removes previously granted or denied permissions at the server level [BOL12]:

```
REVOKE permission_name [ ,...n ] TO principal_name [ ,...n ]
[ CASCADE ]
```

CASCADE – this option removes the permission *permission_name* from other principals to which *principal_name* granted or denied it.

<u>Example 7.7</u>: Assigning server-scoped permissions to logins.

```
-- win_account is a Windows login
GRANT ALTER ANY DATABASE, ALTER ANY LINKED SERVER
                              TO [win_account]
```


Login Properties - win_account

Select a page: General, Server Roles, User Mapping, Securables, Status

Login name: win_account

Securables:

Name	Type
KAIBD-C121\SQLENT12	Server

Connection

Server: .\SQLENT12

Connection: KAIBD-C121\user

View connection properties

Progress: Ready

Permissions for KAIBD-C121\SQLENT12:

Explicit | Effective

Permission	Grantor	Grant	With Grant
Alter any credential		☐	☐
Alter any database		✓	☐
Alter any endpoint		☐	☐
Alter any event notification		☐	☐
Alter any event session		☐	☐
Alter any linked server		✓	☐

```
-- sql1 and sql2 are SQL Server logins
GRANT ALTER TRACE TO sql1, sql2
```

Example 7.8: Creating a user-defined server role and assigning permissions to it.

```
USE master
create server role server_role1
GRANT CONNECT SQL, VIEW ANY DEFINITION TO server_role1
```

Example 7.9: Removing server-scoped permissions from a login.

```
REVOKE ALTER TRACE TO sql1
```

Example 7.10: Removing server-scoped permissions from a user-defined server role.

```
REVOKE VIEW ANY DEFINITION TO server_role1
```

Creating database users

The most common method to allows logins access to database is to create database users for them. The following T-SQL statement is applied to create database users [BOL12]:

```
CREATE USER db_user_name
FOR LOGIN login_name
[ WITH DEFAULT_SCHEMA = schema_name ]
```

db_user_name – it represents the name of the database user.

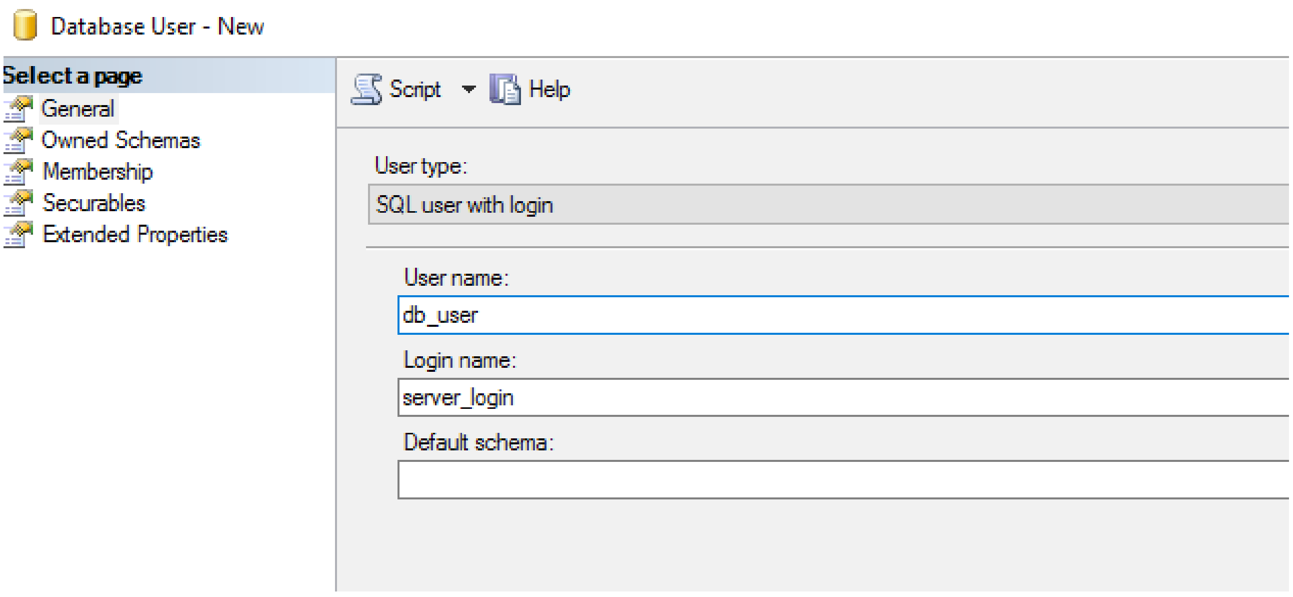
login_name – it represents the name of the login for which the database user is *db_user_name.*

DEFAULT_SCHEMA schema_name - it sets the default schema (container for objects). By default the *dbo* schema is assigned to the user.

Example 7.11: Creating a database user.

The following script creates a database user named *db_user* for a login named *server_login* in a database named *demoDB*:

```
USE demoDB
CREATE USER db_user FOR LOGIN [server_login]
```



User type: SQL user with login

User name: db_user

Login name: server_login

Default schema:

Altering database users

The ALTER USER statement is applied to modify database users [BOL12]:

```
ALTER USER database_user
WITH [NAME=new_db_user_name]
[ , DEFAULT_DATABASE=database_name]
[ , LOGIN=new_login_name]
```

NAME=new_db_user_name – it sets the user name to *new_db_user_name.*

DEFAULT_DATABASE=database_name - it sets the default database to database*_name.*

LOGIN=new_login_name – it associates the user with a login named *new_login_name.*

Example 7.12: Changing the user name from *userOLD* to *userNEW* in the *demoDB* database.

```
USE demoDB
ALTER USER userOLD WITH NAME = userNEW
```

Creating user based on certificates

Examine the following code and run it inside the instance server.

```
USE [demoDB]
GO
create master key encryption by password='1qaz@WSX'
create certificate CertDB with subject ='CertDB'
create user userC for certificate CertDB
execute as user = ' userC '

create procedure s1.sp1 as
select id,tekst from s1.tab1

exec s1.sp1

create procedure s2.sp2 as
exec s1.sp1

exec s2.sp2

grant execute on s2.sp2 to user3
execute as user='user3'
exec s2.sp2
revert
add signature to s2.sp2 by certificate CertDB
grant execute on s1.sp1 to userC
execute as user='user3'
exec s2.sp2
exec s1.sp1
```

7.3 Ownership chains

Examine the following code and run it inside the instance server.

```
USE [master]
GO
CREATE LOGIN [user1] WITH PASSWORD=N'user1',
DEFAULT_DATABASE=[master],
CHECK_EXPIRATION=OFF,CHECK_POLICY=OFF
GO
USE[master]
GO
CREATE LOGIN [user2] WITH PASSWORD=N'user2',
DEFAULT_DATABASE=[master],
CHECK_EXPIRATION=OFF,CHECK_POLICY=OFF
GO
USE[master]
GO
CREATE LOGIN [user3] WITH PASSWORD=N'user3',
DEFAULT_DATABASE=[master],
CHECK_EXPIRATION=OFF,CHECK_POLICY=OFF
GO
USE[demoDB]
GO
CREATE USER user1 FOR LOGIN user1
CREATE USER user2 FOR LOGIN user2
CREATE USER user3 FOR LOGIN user3
GO
CREATE SCHEMA s1 AUTHORIZATION user1
CREATE SCHEMA s2 AUTHORIZATION user2
CREATE SCHEMA s3 AUTHORIZATION user3
create tables2.tab2(id int identity primary key ,tekst varchar(30));
insert into s2.tab2 values('aaa'),('bbb')
create view s1.view2 as
select id,tekst
```

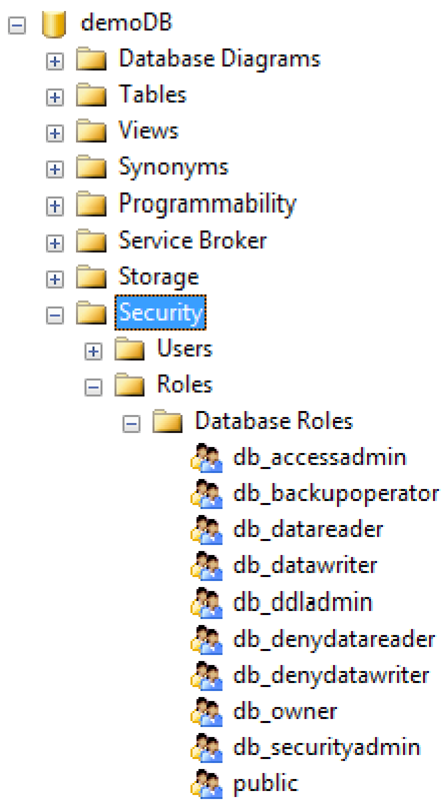
```
froms2.tab2
create view s1.view1 as
select id*2 id2,tekst tekst2
from s1.view2
grant select on s1.view1 to user3
select *from s1.view1
execute as user='user3'
select USER_NAME()
select * from s1.view1
revert
selec USER_NAME()
grant select on s2.tab2 to user3

execute as user='user3'
select USER_NAME()
select * from s1.view1
```

Fixed database roles

SQL Server provides the set of database-level roles. The following predefined roles exist in each database [BOL12]:

Role	**Permission**	**Description**
db_owner	CONTROL	It allows to manage the whole database.
db_securityadmin	ALTER ANY APPLICATION ROLE, ALTER ANY ROLE, CREATE SCHEMA, VIEW DEFINITION	It lets to manage permissions.
db_accessadmin	ALTER ANY USER CREATE SCHEMA, CONNECT	It allows to manage access to the database.
db_backupoperator	BACKUP DATABASE, BACKUP LOG, CHECKPOINT	It lets to backup the database
db_ddladmin	e.g. CREATE TABLE, CREATE VIEW, CREATE PROCEDURE, CREATE FUNCTION, etc.	It lets to perform any DDL (create, alter, drop) statement in a database
db_datawriter	DELETE, INSERT, UPDATE	It allows to perform any DML (insert, update, delete) statement.
db_datareader	SELECT	It lets to perform select statement

		on any database objects (tables, views).
db_denydatawriter	*Denied*: SELECT	It prohibits DML operations within a database.
db_denydatareader	*Denied*: DELETE, INSERT, UPDATE	It prohibits select operations within a database.



The following procedures are dedicated to fixed database roles [BOL12]:

- *sp_addrolemember* '*db_user_name*', '*db_role_name*' – it causes that a user be the member of a role.
- *sp_droprolemember* '*db_user_name*', '*db_role_name*' – it removes a user from a role.
- *sp_helprolemember* '*db_role_name*' – it displays the members of a role.
- *sp_dbfixedrolepermission* '*db_role_name*' – it displays permissions which belong to a role.
- *IS_ROLEMEMBER*('*db_role_name*' ,'*db_user_name*') – this functions returns *true* if a user is the member of a role; otherwise returns *false*.

User-defined database roles

The CREATE ROLE statement [BOL12] is applied to create database roles defined by users:

```
CREATE ROLE db_role_name [ AUTHORIZATION db_role_owner ]
```

The following statement lets to remove user-defined database roles:

```
DROP ROLE role_name
```

Assigning permissions to database users

The GRANT statement allows to grant permissions on database objects to database users or roles [BOL12]:

```
GRANT permission_name [ ( column_name [ ,...n ] ) ] [ ,...n ]
 [ ON [ s_type :: ] securable_name ] TO principal_name [ ,...n ]
  [ WITH GRANT OPTION ]
```

permission_name – it represents the name of a permission.

column_name – it represents the name of a column

s_type – it represents objects such as schemas, tables, procedures, etc.)

WITH GRANT OPTION – this clause lets the principal to grant the permission to other principals.

The REVOKE statement removes previously granted or denied permissions [BOL12]:

```
REVOKE permission_name [ ( column_name [ ,...n ] ) ] [ ,...n ]
 [ ON [ s_type :: ] securable_name ] TO principal_name [ ,...n ]
  [ CASCADE ]
```

Example 7.14: Granting permission (with grant option) to create tables in the *demoDB* database.

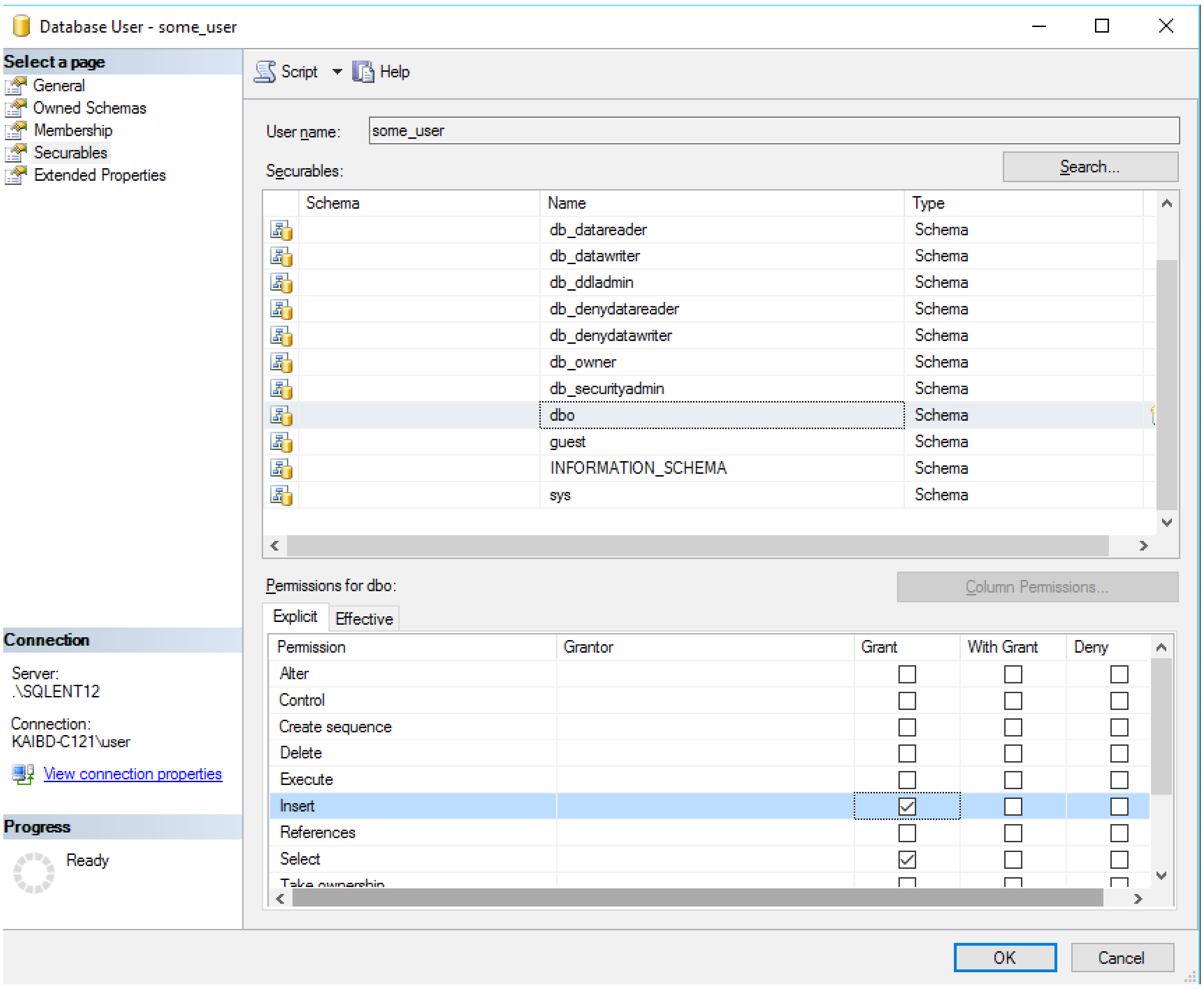
```
USE demoDB
GRANT CREATE TABLE TO some_db_user
WITH GRANT OPTION
```

Example 7.15: Granting permission to alter views in the *demoDB* database.

```
USE demoDB
GRANT ALTER VIEW TO some_db_user
```

Example 7.16: Granting SELECT and INSERT permissions on the *dbo* schema to *user4*.

```
USE demoDB
GRANT SELECT,INSERT ON SCHEMA::dbo TO some_user
```


Database User - some_user

Select a page
- General
- Owned Schemas
- Membership
- Securables
- Extended Properties

Script ▾ Help

User name: some_user

Securables:

Schema	Name	Type
	db_datareader	Schema
	db_datawriter	Schema
	db_ddladmin	Schema
	db_denydatareader	Schema
	db_denydatawriter	Schema
	db_owner	Schema
	db_securityadmin	Schema
	dbo	Schema
	guest	Schema
	INFORMATION_SCHEMA	Schema
	sys	Schema

Permissions for dbo:

Explicit | Effective

Permission	Grantor	Grant	With Grant	Deny
Alter		☐	☐	☐
Control		☐	☐	☐
Create sequence		☐	☐	☐
Delete		☐	☐	☐
Execute		☐	☐	☐
Insert		✓	☐	☐
References		☐	☐	☐
Select		✓	☐	☐
Take ownership		☐	☐	☐

Connection

Server:
.\SQLENT12

Connection:
KAIBD-C121\user

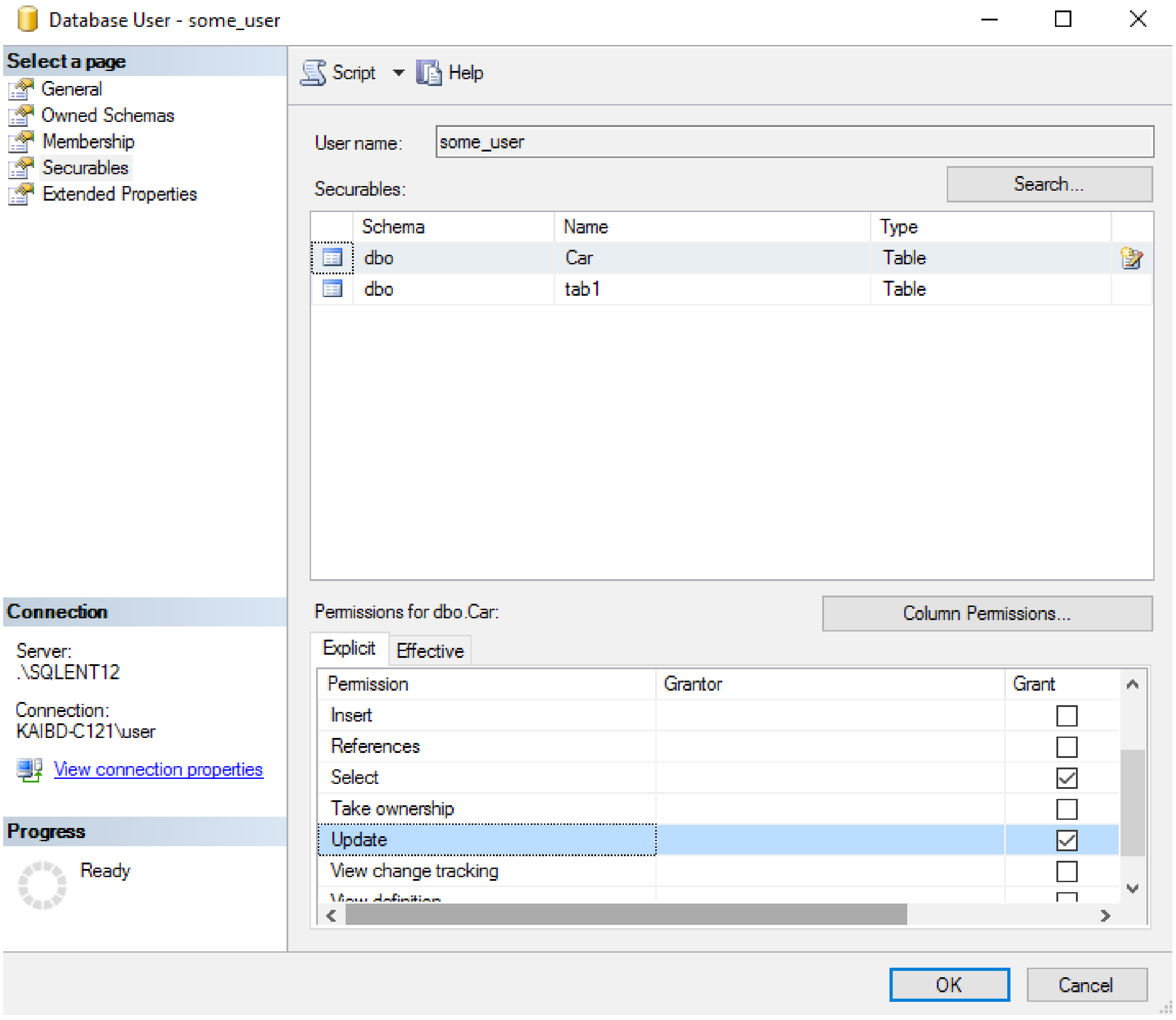
View connection properties

Progress

Ready

Example 7.17: Granting SELECT and UPDATE permission on a table named *Car*.

```
USE demoDB

GRANT SELECT,UPDATE ON OBJECT::dbo.Car TO some_user
```


Database User - some_user

Select a page
- General
- Owned Schemas
- Membership
- Securables
- Extended Properties

Script ▾ Help

User name: some_user

Securables: Search...

	Schema	Name	Type
	dbo	Car	Table
	dbo	tab1	Table

Connection

Server:
.\SQLENT12

Connection:
KAIBD-C121\user

View connection properties

Progress

Ready

Permissions for dbo.Car: Column Permissions...

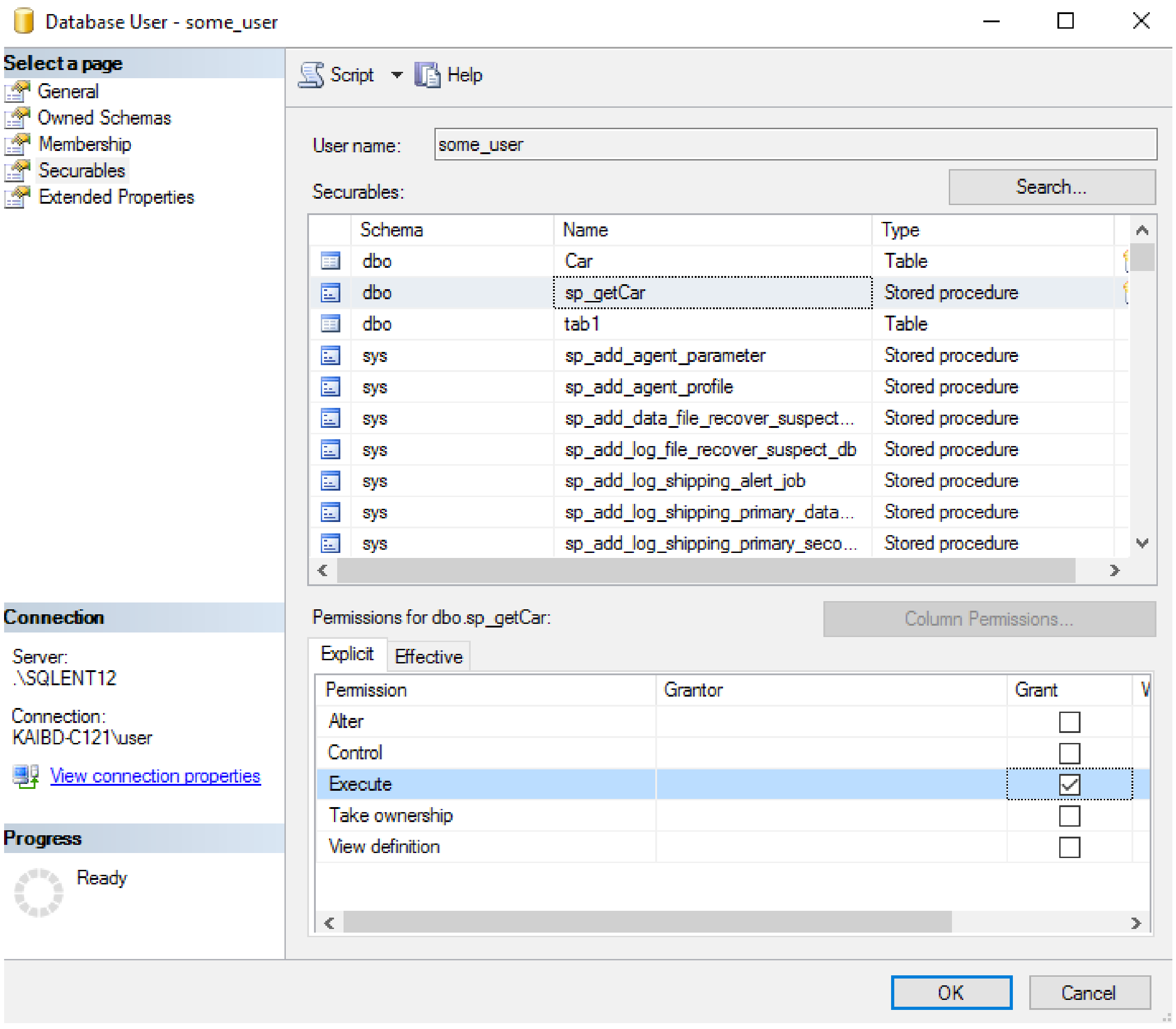
Explicit | Effective

Permission	Grantor	Grant
Insert		☐
References		☐
Select		✓
Take ownership		☐
Update		✓
View change tracking		☐
View definition		☐

OK Cancel

Example 7.18: Granting EXECUTE permission on a procedure named *sp_getCars*.

```
USE demoDB
GRANT EXECUTE ON OBJECT::dbo.sp_getCar TO some_user
```


Database User - some_user

Connection

Server:
.\SQLENT12

Connection:
KAIBD-C121\user

View connection properties

Progress

Ready

User name: some_user

Securables:

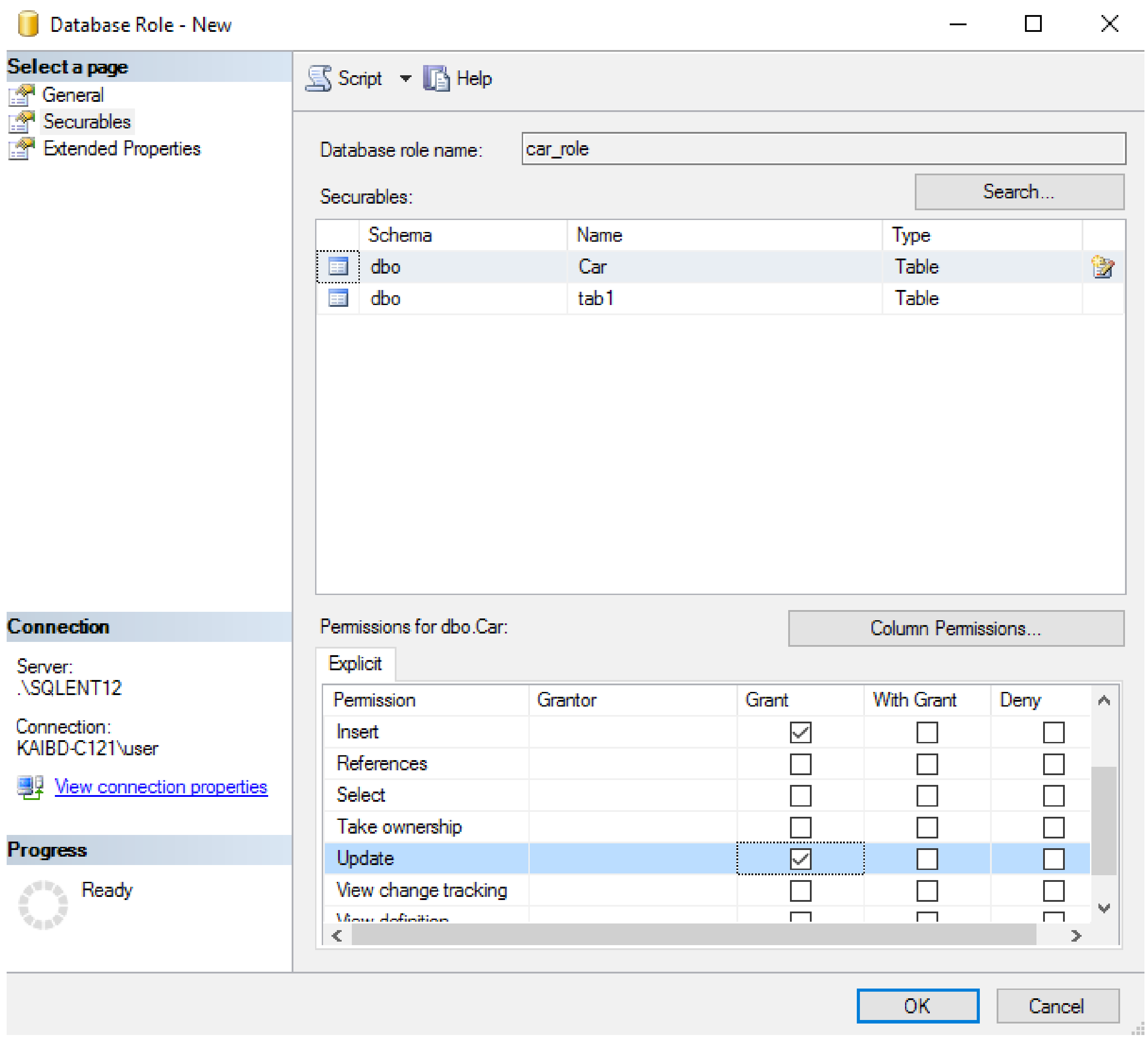
Schema	Name	Type
dbo	Car	Table
dbo	sp_getCar	Stored procedure
dbo	tab1	Table
sys	sp_add_agent_parameter	Stored procedure
sys	sp_add_agent_profile	Stored procedure
sys	sp_add_data_file_recover_suspect...	Stored procedure
sys	sp_add_log_file_recover_suspect_db	Stored procedure
sys	sp_add_log_shipping_alert_job	Stored procedure
sys	sp_add_log_shipping_primary_data...	Stored procedure
sys	sp_add_log_shipping_primary_seco...	Stored procedure

Permissions for dbo.sp_getCar:

Permission	Grantor	Grant
Alter		☐
Control		☐
Execute		✓
Take ownership		☐
View definition		☐

Example 7.19: Granting INSERT and UPDATE permissions on a table named *Car* to a role named *car_role.*

```
USE demoDB
CREATE ROLE car_role ;
GRANT INSERT, UPDATE ON OBJECT::dbo.Car TO car_role
```



Database Role - New

Select a page
- General
- Securables
- Extended Properties

Script ▾ Help

Database role name: car_role

Securables: Search...

Schema	Name	Type
dbo	Car	Table
dbo	tab1	Table

Connection

Server:
.\SQLENT12

Connection:
KAIBD-C121\user

View connection properties

Progress

Ready

Permissions for dbo.Car: Column Permissions...

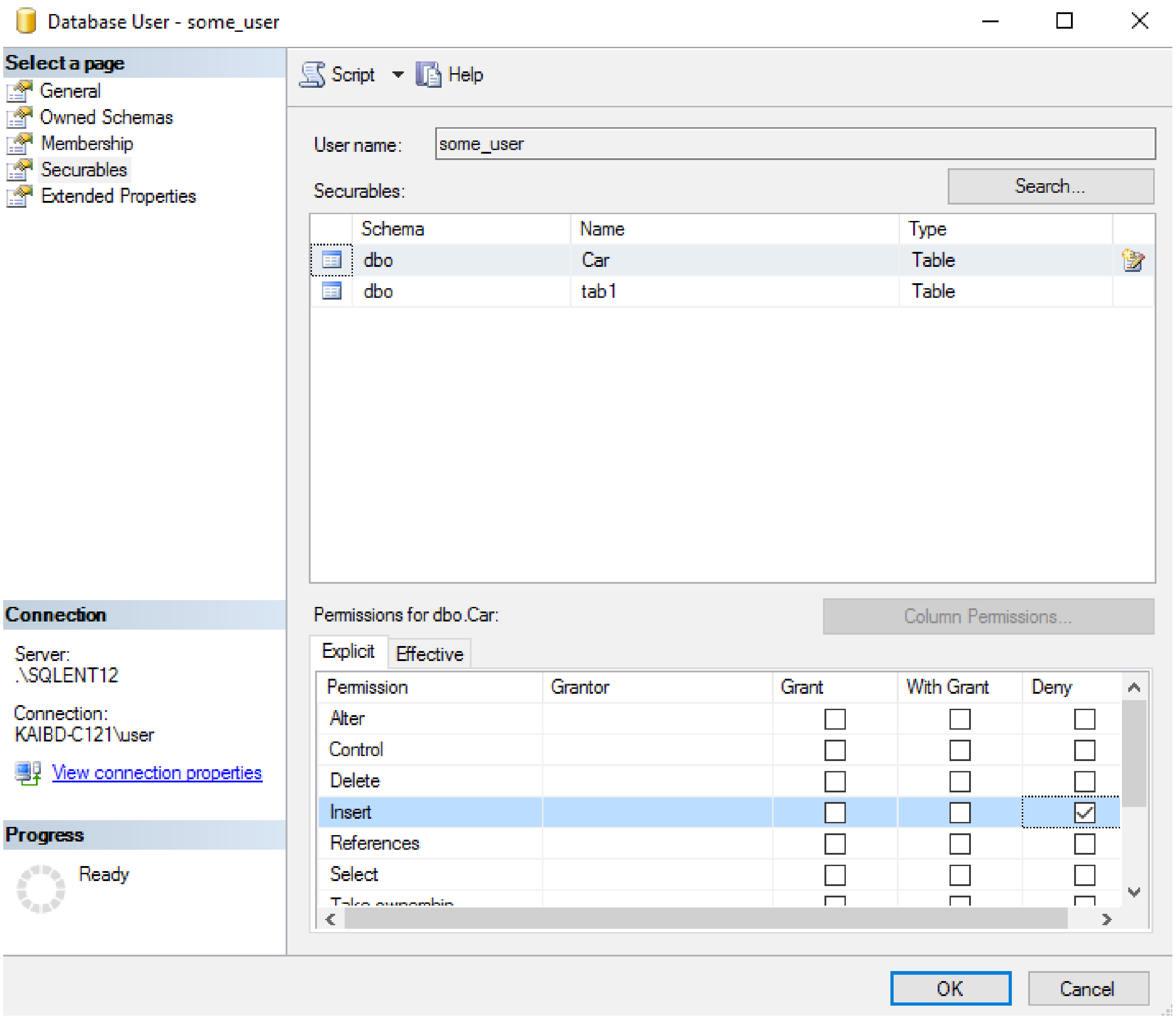
Explicit

Permission	Grantor	Grant	With Grant	Deny
Insert		✓	☐	☐
References		☐	☐	☐
Select		☐	☐	☐
Take ownership		☐	☐	☐
Update		✓	☐	☐
View change tracking		☐	☐	☐
View definition		☐	☐	☐

OK Cancel

<u>Example 7.20:</u> Denying INSERT permission on *Car* to *some_user* .

```
USE demoDB
DENY INSERT ON OBJECT::dbo.Car TO some_user
```


Database User - some_user

Select a page: General, Owned Schemas, Membership, Securables, Extended Properties

User name: some_user

Securables:

	Schema	Name	Type
	dbo	Car	Table
	dbo	tab1	Table

Permissions for dbo.Car: (Explicit tab)

Permission	Grantor	Grant	With Grant	Deny
Alter		☐	☐	☐
Control		☐	☐	☐
Delete		☐	☐	☐
Insert		☐	☐	✓
References		☐	☐	☐
Select		☐	☐	☐

Connection — Server: .\SQLENT12; Connection: KAIBD-C121\user

Progress: Ready

Example 7.21: Revoking UPDATE permission on the *dbo* schema to *some_user.*

```
USE demoDB
REVOKE UPDATE ON SCHEMA::dbo TO some_user
```

Assignment

1. Verify the instance authentication mode.
2. Set the authentication mode to the mixed mode.
3. Acitvate the *sa* login.
4. Change the name of *sa* to *sqlLogAdmin* and set the strong password for it. Logon as *sqlLogAdmin*.
5. Create a new Windows account called *sql* and login for it in SSMS.

6. Logon to the instance as *sql* and check whether this login has access to the instance databases.

7. In the *demoDB* database create a database user called *demo* for the *sql* login.

8. Check whether the *demo* user can display data from the tables stored in *demoDB*.

9. Grant select, insert, update, delete permissions on all tables stored in the *dbo* schema for the user *demo*.

10. Revoke the insert permission on the *dbo* schema from *demo*.

11. Add the user *demo* to the *db_owner* role in the *demoDB* database.

12. Create a new role called *test_role* in the *demoDB* database.

13. Add some permission to this role.

14. Remove *demo* from the *db_owner* role . Add *demo* to the *test_role* role.

8 Backups in SQL Server

SQL Server supports the following backups [BOL12]:

- Full database backup – it involves all data pages and the active part of the log file.
- Differential database backup – it involves only pages which have been changed since the last full backup.
- Partial backup – it contains files from the PRIMARY filegroup, any filegroups having the READ/WRITE status, READ ONLY filegroups if they will be specified.
- Filegroup backup.
- Filegroup differential backup.
- Transaction log backup – it contains the active log part and the part which does not occur in the previous log backup (The database recovery model must be set to full or bulk-logged).
- Copy only backup (full or log backup).

8.1 Full database backup

Perform full database backup and display information about it.

```
BACKUP DATABASE demoDB TO DISK = 'd:\data\demoDB_Full.bak'
SELECT FROM msdb.dbo.backupset

SELECT * FROM msdb.dbo.backupmediaset

SELECT * FROM msdb.dbo.backupmediafamily

SELECT * FROM  msdb.dbo.backupfile
```

Sample backup sequence

- Full backup – once a week (on Sunday at twelve o'clock at night).
- Differential backup – once a day from Monday to Saturday (at 8 p.m.).
- Transaction log backup – every 5 minutes between 9 a.m. and 7 p.m.
- The above strategy guarantees that in the case of failure we lose the transactions from maximum 5 minutes.

Backup media

- Backup set – it represents a single backup.
- Backup device – disk files or tapes.
- Media set – it can include up to 64 backup devices.

Backup validation

- Backup statement can count the backup checksum. It is stored in the backup header.
- The CHECKSUM option implies the following operations.
    - Verifying each data page if the page verify database option is enabled.
    - Generating checksum for the whole backup.
- NO_CHECKSUM is the default.
- The backup statement with the *checksum* option results in failure if corrupt data pages are detected.
- The CONTINUE_AFTER_ERROR option does not stop the backup operation.

```
BACKUP DATABASE demoDB
TO DISK = 'd:\data\demoDB_Full.bak'
WITH INIT, CHECKSUM
```

- The RESTORE VERIFYONLY statement allows to check whether the backup is complete and readable.
- The CHECKSUM option of RESTORE VERIFYONLY verifies the checksum of each page and the checksum of the whole backup.

```
RESTORE VERIFYONLY
FROM DISK = 'd:\data\demoDB_Full.bak'
WITH CHECKSUM
```

Information about backups

- RESTORE LABELONLY – it returns information about media sets.
- RESOTRE HEADERONLY – it returns headers of backup sets from backup devices.
- RESTORE FILELISTONLY – it returns list of data files and log files stored in the backup set.

```
RESTORE LABELONLY
FROM DISK='d:\data\demoDB_Full.bak'

RESTORE HEADERONLY
FROM DISK='d:\data\demoDB_Full.bak'

RESTORE FILELISTONLY
FROM DISK='d:\data\demoDB_Full.bak'
```

Sample backup options

INIT – it overwrites the existing backup sets by a new backup set.

NOINIT – it appends a new backup set to the specified media set. It is the default.

```
BACKUP DATABASE  demoDB TO DISK = 'd:\data\demoDB_Full.bak'
WITH NOINIT

BACKUP DATABASE demoDB TO DISK = 'd:\data\demoDB_Full.bak'
WITH INIT
```

Full backup on medium involving two data files

```
BACKUP DATABASE demoDB
TO DISK = 'd:\data\demoDB_Full.bak',
DISK = 'd:\data\demoDB2_Full.bak'
WITH INIT
```

The above statement returns the following error:

```
The media loaded on "d:\data\demoDB_Full.bak" is formatted to support 1
media families, but 2 media families are expected according to the backup
device specification.
```

Solution:

The FORMAT option is required to modify the structure of the existing media. It should be based on two files rather than one file.

```
BACKUP DATABASE demoDB
TO DISK = 'd:\data\demoDB_Full.bak',
DISK = 'd:\data\demoDB2_Full.bak'
WITH INIT, FORMAT,
MEDIANAME = 'demoMedia'
```

The MIRROR TO option

- The *MIRROR TO* allows to perform several backups of the same database using one statement.
- It allows to specify up to three secondary backup devices.

```
BACKUP DATABASE demoDB
TO DISK = 'd:\data\demoDB_Full.bak',
DISK = 'd:\data\demoDB2_Full.bak'
MIRROR TO DISK = 'd:\data2\demoDBM_Full.bak',
DISK = 'd:\data2\demoDBM2_Full.bak'
WITH FORMAT
```

Remark: The FORMAT option is required to create mirrored backup set

Compressed backups

- The *COMPRESSION* option allows to create compressed backup.
- It requires at least Standard Edition of SQL Server 2008 R2.
- It requires Enterprise Edition of SQL Server 2008.

```
BACKUP DATABASE demoDB
TO DISK = 'd:\data\demoDB_Full.bak',
DISK = 'd:\data\demoDB2_Full.bak'
MIRROR TO DISK = 'd:\data2\demoDBM_Full.bak',
DISK = 'd:\data2\demoDBM2_Full.bak'
WITH FORMAT, COMPRESSION
```

Partial backups, filegroup backups, file backups

```
BACKUP DATABASE demoDB
READ_WRITE_FILEGROUPS
TO DISK = 'd:\data\demoDB_Partial.bak'
```

```
WITH MEDIANAME = 'demoDBPartial'

BACKUP DATABASE demoDB
FILEGROUP = 'PRIMARY'
TO DISK = 'd:\data\demoDB_Partial.bak'
WITH MEDIANAME = 'demoDBPartial'

BACKUP DATABASE demoDB
FILE = 'demoDB'
TO DISK = 'd:\data\demoDB_Partial.bak'
WITH MEDIANAME = 'demoDBPartial'
```

8.2 *Differential backups*

- The *DIFFERENTIAL* option is applied to perform differential backups.
- Differential backup involves only extents which have been changed since the last full backup.
- Full backup must be done before making differential backup.

```
BACKUP DATABASE demoDB
TO DISK = 'd:\data\demoDB_Full.bak',
DISK = 'd:\data\demoDB2_Full.bak'
WITH DIFFERENTIAL, FORMAT,
MEDIANAME = 'demoDBDiff'
```

8.3 *Transaction log backups*

- The BACKUP LOG statement allows to do transaction log backup.
- The recovery model must be set to *FULL* or *BULK-LOGGED* to perform log backups.
- If the recovery model is set to simple it is impossible to perform log backups.
- After performing log backup SQL Server truncates inactive part of the log file.

```
BACKUP LOG demoDB
TO DISK = 'd:\data\demoDB_Log.bak'
WITH MEDIANAME = 'demoDBLog'
```

Tail-log backup

- Tail-log backup is log backup which involves the log records that have not yet been backed up.
- **At first, in the case of failure database administrator should perform tail-log backup.**
- After performing tail-log backup DBA can start restore procedure.
- The CONTINUE_AFTER_ERROR option is required to perform tail-log backup if data files are damaged.

```
BACKUP LOG demoDB
TO DISK = 'd:\data\demoDB_Log.bak'
WITH CONTINUE_AFTER_ERROR
```

9 Restoring SQL Server Databases

9.1 Restoring process

The *RESTORE* statement allows to:

- restore the whole database from a full backup (complete restore). It can be followed by restoring differential and log backups.
- restore specific filegroups or files (partial restore).
- restore specific pages (page restore).
- restore a transaction log (transaction log restore).

Database restore process involves the following phases:

1. *Data copy phase* - Creating database files and copying data into them.
2. *Redo phase* – applying transactions from log files to the data to roll forward it to the given recovery point.
3. *Undo phase* – Rolling back uncommitted transactions (they cab be introduced in the second phase).

The redo and undo phase are called *database recovery.*

Sample restore procedure

9.2 Restoring to the point of failure

Consider the following scenario:

The *demoDB* database has a table called *tab1*(*id* int identity primary key, *desc* varchar(30)) which stores two rows

- Perform full database backup.
  ```
  BACKUP DATABASE demoDB TO DISK = 'd:\data\demoDB_Full.bak'
  ```
- Insert two rows to the *tab1* table.
- Perform differential backup.
  ```
  BACKUP DATABASE demoDB TO DISK = 'd:\data\demoDB_Diff.bak'
  ```
- Insert two rows to the *tab1* table.
- Perform transaction log backup.
  ```
  BACKUP LOG demoDB TO DISK = 'd:\data\demoDB_Log.bak'
  ```
- Insert three rows to the *tab1* table.

- Shutdown SQL Server and remove only data files (it is assumed that the log file is not corrupted).
- Start SQL Server and restore *demoDB* to the point of failure.
- Perform tail-log backup (the *norecovery* option – it leaves the database in the restoring state).

  ```
  BACKUP LOG demoDB TO DISK = 'd:\data\demoDB_TailLog.bak'
  WITH CONTINUE_AFTER_ERROR, NORECOVERY
  ```
- Restore the full database backup (it allows to continue the restore procedure – the undo phase is not performed, subsequent backups can be restored).

  ```
  RESTORE DATABASE demoDB FROM DISK = 'd:\data\demoDB_Full.bak'
  WITH NORECOVERY
  ```
- Restore the differential backup.

  ```
  RESTORE DATABASE demoDB FROM DISK = 'd:\data\demoDB_Diff.bak'
  WITH NORECOVERY
  ```
- Restore the log backup.

  ```
  RESTORE LOG demoDB FROM DISK = 'd:\data\demoDB_Log.bak'
  WITH NORECOVERY
  ```
- Restore the tail-log backup (RECOVERY is the default, it brings the database from the *recovering* state to the *online* state).

  ```
  RESTORE LOG demoDB FROM DISK = 'd:\data\demoDB_TailLog.bak'
  WITH RECOVERY
  ```

9.3 Restoring to the point in time

The *STOPAT date_time_expression* option of the RESTORE LOG statement allows to restore a database to the specified point in time.

Using the *stopat* option (If the specified time is in the log backup having number *k* then the restore process will stop in k-th step):

```
RESTORE DATABASE demoDB
FROM DISK = 'd:\data\demoDB_Full.bak'
WITH NORECOVERY

RESTORE DATABASE demoDB
FROM DISK = 'd:\data\demoDB_Diff.bak'
```

```
WITH NORECOVERY

RESTORE LOG demoDB
FROM DISK = 'd:\data\demoDB_Log1.bak'
WITH STOPAT = '2013-10-21 10:00:00.000', RECOVERY
…….
…….
…….
RESTORE LOG demoDB
FROM DISK = 'd:\data\demoDB_LogK.bak'
WITH STOPAT = '2013-10-21 10:00:00.000', RECOVERY
```

9.4 Restoring to the marked transactions

The STOPATMARK *transaction_name* option of the RESTORE LOG allows to restore a database to the marked transaction (BEFOREMARK - it restores without the marked transaction).

Using the *STOPATMARK* option:

```
BEGIN TRANSACTION transaction_name WITH MARK 'some description'
--some operations
COMMIT TRANSACTION transaction_name

RESTORE DATABASE demoDB
FROM DISK = 'd:\data\demoDB_Full.bak'
WITH NORECOVERY

RESTORE DATABASE demoDB
FROM DISK = 'd:\data\demoDB_Diff.bak'
WITH NORECOVERY

RESTORE LOG demoDB
FROM DISK = 'd:\data\demoDB_Log1.bak'
WITH STOPATMARK = 'transaction_name', RECOVERY
……….
……….
……….
RESTORE LOG demoDB
```

```
FROM DISK = 'd:\data\demoDB_LogK.bak'
WITH STOPATMARK = 'transaction_name', RECOVERY
```

9.5 Restoring system databases

Restoring the master database

- DBA must have some data and log files of the *master* database (the *master* database files are required to start the instance).
- Alternatively DBA can generate *master* using *SQL Server Setup* (this process overwrites another system databases).
- DBA must start the SQL Server instance in single user mode (add *-m* in the startup options).
- DBA must run the following command in *sqlcmd*.
  ```
  RESTORE DATABASE master
  FROM 'd:\data\master_Full.bak' WITH REPLACE
  ```
- The *REPLACE* option allows to restore the database if a database of the same name already exists.
- Remove the *-m* parameter from the startup option.
- Run SQL Server instance.

Restoring the model database

- DBA must have some data and log files of the *model* database.
- DBA can generate *model* using SQL Server Setup.
- DBA must start the SQL Server instance with the T3608 parameter (it causes that only the *master* database is started).
- DBA must run the following command.
  ```
  RESTORE DATABASE model
  FROM 'd:\data\model_Full.bak' WITH REPLACE
  ```

Restoring other types of databases

- *Tempdb* – it is recreated during the instance startup (the backup and restore statements are not available).
- *Resource* (read-only database which stores all the system objects) – it can be restored using operating system backups of its files or *SQL Server Setup* (the backup and restore statements are not available).
- *Msdb* – restoring this database is analogous to restoring user-defined databases.

9.6 Backup Recovery Assignment

1. Create a database called *demoDB* with one table *Students*(*id int identity primary key*, *fName varchar(80), lName varchar(80))*.
2. Perform full database backup of *demoDB* to the medium involving one file called *demoDB1.bak*.
3. Insert one row to the *Students* table. Perform differential backup of *demoDB* to the file *demoDB1.bak*. The full database backup stored in this should not be overwritten.
4. Perform differential backup of *demoDB* to a new medium involving two files *demoDBdf1.bak* and *demoDBdf2.bak*.
5. Set the value of the *recovery model* parameter to *full* if it has different value. Perform full database backup of *demoDB* to *demoDBdf1.bak* and *demoDBdf2.bak* with the *init* option. Perform transaction log backup to a new medium involving one file called *demoDBlb1.bak.*
6. Perform full compressed backup of *demoDB* to *demoDBdf1.bak* and *demoDBdf2.bak.* Is the *format* option required?
7. Perform full mirrored backup of *demoDB* to (*demoDBm11.bak, demoDBm12.bak*) and (*demoDBm21.bak,demoDBm22.bak*)
8. Add a new filegroup called *fg2* to the *demoDB* database. Add a new file called *demoDB2.ndf* to *demoDB* and assign it to *fg2*.
9. Create a table called *tab2*(id int) in the file *demoDB2.ndf.*
10. Perform full backup of *fg2*.
11. Insert some rows to the *tab2* table. Perform differential backup of *fg2*.
12. Perform full, differential and transaction log backup of the *master* database. Is it possible?
13. Perform full, differential and transaction log backup of the *msdb* database. Is it possible?
14. Perform full, differential and transaction log backup of the *model* database. Is it possible?
15. Perform full, differential and transaction log backup of the *tempdb* database. Is it possible?

9.7 Sample recovery procedure

The following procedure can be used to automate restore process. Examine the code of the procedure and try to invoke it.

```
create procedure [dbo].[getRestoreSequence] @dbName varchar(200) ,
@backupLoc varchar(200) as
begin


declare @row varchar(400)
```

```
set @row = 'backup log '+@dbName+ ' to disk='''+@backupLoc+ ''' with
continue_after_error'
print @row
exec(@row)

declare bk cursor for
  select 'restore database '+ bs.database_name + ' from disk = '''+
bmf.physical_device_name + ''' with norecovery'
    from msdb.dbo.backupset bs join msdb.dbo.backupmediaset bms on
bs.media_set_id=bms.media_set_id
       join msdb.dbo.backupmediafamily bmf on
bmf.media_set_id=bms.media_set_id
    where bs.database_name=@dbName and bs.backup_finish_date >= (select
max(backup_finish_date) from msdb.dbo.backupset where type='D')
    order by bs.backup_set_id asc

OPEN bk;
FETCH NEXT FROM bk INTO @row;
WHILE @@FETCH_STATUS=0
    BEGIN
    print @row
        exec(@row)
    FETCH NEXT FROM bk INTO @row;
    END
CLOSE bk
DEALLOCATE bk

print 'restore database '+ @dbName+ ' with recovery'

exec('restore database '+ @dbName+ ' with recovery')

end

GO


exec [dbo].[getRestoreSequence] 'demo','F:\backup\log\demo_ogonek.bak'
```

10 High Availability

High availability mechanisms are applied to reduce downtime in the case of system failure. MS SQL Server provides the following high availability mechanisms:

- Log Shipping.
- Database mirroring.
- Always On High Availability Groups (It is supported from version 2012).
- Always On Failover Cluster Instances (or SQL Server Failover Clustering prior to 2012 version).

10.1 Log Shipping

Log shipping provides high availability on the database level. It allows to maintain so called standby database for a given primary database on the remote server. The following steps are required to implement log shipping [BOL12]:

- Set the recovery model to *full* for a primary database.
- Perform full backup of the primary database.
- Restore the above backup on the remote server.
- Run the procedure *sp_add_log_shipping_primary_database* on the server which stores the primary database. The procedure creates a job which performs transaction log backup of the primary database according to the given schedule.
- Run the procedure *sp_add_log_shipping_secondary_database* on the remote server. This procedure creates two jobs. The first one is responsible for copying transaction log backups of the primary database to the remote server. The second job will restore these backups on the remote server.
- Run *sp_add_log_shipping_primary_secondary_database* on the server which stores the primary database.

The T-SQL code which allows to implement log shipping can be generated using SSMS. Right-click the database which will act as the primary database and then click *Properties*. In the panel *Select a page* choose *Transaction Log shipping* and provide the proper values. Finally press the button called *Script Configuration*.

10.2 Database Mirroring

Database Mirroring also provides high availability at the database level. A database mirroring session usually has two participants:

- Principal server (It stores a database which accepts user transactions).
- Mirror server (It stores a database which accepts the transaction previously performed on the database hosted by the principal server. Transaction log records are regularly sent from the principal server to the mirror server in synchronous or asynchronous mode depending on the configuration).
- Witness server (It is optional component which can be implemented using server with SQL

Server Express Edition. It is required only if DBA wants to implement automatic failover).

The following steps are required to implement log shipping [BOL12]:

- Set the recovery model to *full* for a database which is hosted on the principal server.
- Perform full backup of the principal database.
- Restore the above backup on the mirror server.
- Configure endpoints on each server instance.
- Configure security mechanism for database mirroring session (Windows authentication or certificate authentication).

10.3 AlwaysOn Availability Groups

AlwaysOn Availability Groups have been implemented in SQL Server 2012. They can be treated as extension of the database mirroring mechanism. An availability group involves two main elements: [BOL12]:

- Primary replica – set of user-defined databases.
- One to four secondary replica (each secondary replica contains databases which have their counterparts in the primary replica).

AlwaysOn Availability Groups are based on the Windows Server Failover Clustering cluster. Replicas must be stored on the cluster nodes. DBA must perform the following steps to configure AlwaysOn Availability Groups [BOL12]:

- Create and configure Windows Server Failover Clustering cluster.
- Activate the AlwaysOn Availability Group mechanism on instances which will host replicas.
- Configure an availability group listener.
- Create an availability group on the instance which hosts the primary replica.
- Add each secondary replica to the above availability group.
- Restore backups of the databases which belongs to the primary replica to prepare secondary databases.
- Add secondary databases to the availability group.

10.4 AlwaysOn Failover Cluster Instances

AlwaysOn Failover Cluster Instances (FCI) have been introduced in MS SQL Server 2012. In contrast to other solutions (Log Shipping, Database Mirroring, AlwaysOn Availability Groups) they provide high availability on the instance level. Windows Server Failover Clustering cluster is essential to implement FCI. Cluster nodes can be physical or virtual computers. The following steps are required to implement FCI in the virtual environment:

- Install hypervisor (e.g. Hyper-V, VirtualBox).
- Create three virtual machines with Windows Server 2012.
- Network adapters for virtual machines can be configured as follows:

Server name	**Production network**	**ISCSI network**	**Heartbeat network**
server1.xyz.com	162.30.0.10	162.30.1.10	
node1.xyz.com	162.30.0.20	162.30.1.20	162.30.2.20
node2.xyz.com	162.30.0.30	162.30.1.30	162.30.2.30

- Install the *Active Directory Domain Services* role on *server1*.
- Promote *server1* to a domain controller (The root domain name is set to *xyz.com*).
- Create two domain accounts for MS SQL Server instance (*xyz\sql_service*, *xyz\sql_agent*).
- Join the servers *node1* and *node2* to the *xyz.com* domain.
- Install the *Target Server* role on *server1.xyz.com.*
- Create an iSCSI target with one virtual hard disk which will store MS SQL Server database files.
- Run iSCSI Initiator on *node1.xyz.com* and configure the connection to the above iSCSI Target (Perform this step also on *node2.xyz.com*).
- Initialize the shared disk on *node1.xyz.com*.
- Create a shared folder called *cluster01-witness* on *server1.xyz.com*. (The servers *node1.xyz.com* and *node2.xyz.com* should have read/write permissions to this folder).
- Install the *Failover Clustering* feature on *node1.xyz.com* and *node2.xyz.com*.
- Run Failover Cluster Manager on *node1.xyz.com* an create a cluster named *cluster01-sql* (162.30.0.40).
- Create
- Set quorum configuration to File Share Majority (By default it is set to *Node and Disk Majority*).
- Set the file share witness to the shared folder *cluster01-witness.*
- Run SQL Server Installation Center on *node1.xyz.com* and type *New SQL Server failover cluster installation.*
- Install a default MS SQL Server instance on *node1.xyz.com.*

- Run SQL Server Installation Center on *node2.xyz.com* and type *Add node to a SQL Server failover cluster.*
- Install a default MS SQL Server instance on *node2.xyz.com.*

11 Bibliography

[BOL12]	Books Online for SQL Server 2012, http://technet.microsoft.com/en-us/library/ms130214.aspx
[EN04]	R.Elmasri, S.B. Navathe. Fundamentals of Database Systems. 4th Edition, Addison-Wesley Publishing Company, 2004
[GMUW08]	Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom. Database Systems: The Complete Book. Prentice Hall, 2008.
[RG03]	Raghu Ramakrishnan, Johannes Gehrke. Database Management Systems, Third Edition (Published by McGraw-Hili)