Contents

Database and Table Creation 2
Constraints 3
ALTER TABLE 8
The IDENTITY Property 9
Indexes 9

CMPE2400 – Databases – Notes V

Database and Table Creation

This section will explore the syntax of database and table creation using SQL Data Definition Language (DDL) syntax. Implementation of DDL will differ slightly across Server Platforms depending on the extensions added by the vendor. For example, most vendors supply a server managed numeric auto generated key value for fields – Microsoft calls this identity, whereas mySQL calls it auto_increment.

Microsoft SQL Server provides the facility for authorized users to create, modify and destroy databases. This is entirely dependent on the user's privilege set within the server context.

The simplest form of database creation is the CREATE DATABASE keywords along with the name of the database you wish to create, followed by a GO command. Many defaults are provided for you, some of which include database and log file name and location, database properties, default privileges, etc.

In this course, by convention, all our user created databases you create must start with your user name:

```
create database fredf_Sample
go
```

Bender will use a set of defaults to create all of the necessary files to back this new database. As the owner of the database, you will have rights to do everything you need to do with the database.

To delete the database, you DROP it:

```
drop database fredf_Sample
go
```

You will typically want to ensure that a database is present before you drop it, particularly in a script that creates it:

```
if exists
(
	select *
	from sysdatabases
	where [name] = 'fredf_Sample'
)
drop database fredf_Sample
go
```

NOTE: You can't drop the current database! Switch to a different database if you are attempting to drop the one you are currently using.

Once your database is created, you are free to add tables to it. Tables are created with the CREATE TABLE keywords, followed by the table name, and then the definitions for the columns in the table. Unless you name resolve otherwise, tables will be added to the current database, so ensure that you are using the appropriate database when you are adding a table (or fully resolve the name).

```
create table Monkey
(
	[Name] nvarchar (50) not null,
	Age tinyint null
)
go
```

As the above example shows, each column in the table has a name, a type, and nullable specification. Remember that this statement has not created any data, just the container for it. Columns that are marked as null will be allowed NULL as a value.

Constraints

The tables you create may have constraints added to them. A constraint is a named rule that is used by the database engine to automatically enforce integrity.

Any constraints that are specified as a part of a column definition are called column constraints, and apply only to that column. If a constraint is specified outside of a column definition, but within the table definition, it is knows as a table constraint.

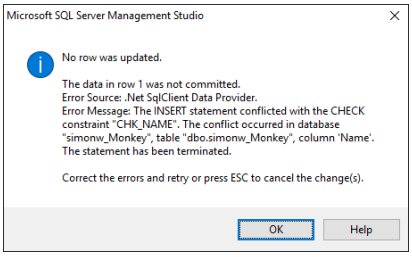
You have already seen one *sort of* constraint – the NOT NULL specification. The database engine will not allow NULL values to be inserted into a NOT NULL column. If the value is required, then this column constraint prevents having a row of data with NULL in this column.

With the exception of the null specification, a constraint is named, typically with a prefix for the type of constraint it is, followed by the name of the column or columns the constraint applies to. Constraint names must be unique within the database, so pick quality names that aren't likely to be repeated.

The CHECK column constraint will ensure that a Boolean expression evaluates to *true* before a value is put in that column:

```
create table Monkey
(
	[Name] nvarchar (50) not null
		constraint chk_Monkey_Name check (len ([Name]) >= 3),
	Age tinyint null
)
```

Failure to meet the requirements of the CHECK constraint will cause errors:



The UNIQUE column constraint will ensure that no two rows have the same value. NULL is considered a unique value in the case of UNIQUE.

```
create table Monkey
(
	[Name] nvarchar (50) not null
		constraint chk_Monkey_Name check (len ([Name]) >= 3)
		constraint unique_Monkey_Name unique,
	Age tinyint null
)
go
```

You may specify a default value for a column with a DEFAULT constraint:

```
create table fredf_Sample.dbo.Monkey
(
	[Name] nvarchar (50) not null
		constraint chk_Monkey_Name check (len ([Name]) >= 3)
		constraint unique_Monkey_Name unique,
	Age tinyint not null
		constraint def_Monkey_Age default 1
)
go
```

If the value is not specified for the column, then the default will be used.

The PRIMARY KEY constraint identifies the column or columns that have values that uniquely identify each row in the table. Every table should have a primary key. They PRIMARY KEY constraint implies the UNIQUE constraint, but unlike UNIQUE, NULL is not allowed.

```
create table fredf_Sample.dbo.Monkey
(
	MonkeyID int not null constraint pk_Monkey_MonkeyID primary key,
	[Name] nvarchar (50) not null
		constraint chk_Monkey_Name check (len ([Name]) >= 3)
		constraint unique_Monkey_Name unique,
	Age tinyint not null
		constraint def_Monkey_Age default 1
)
go
```

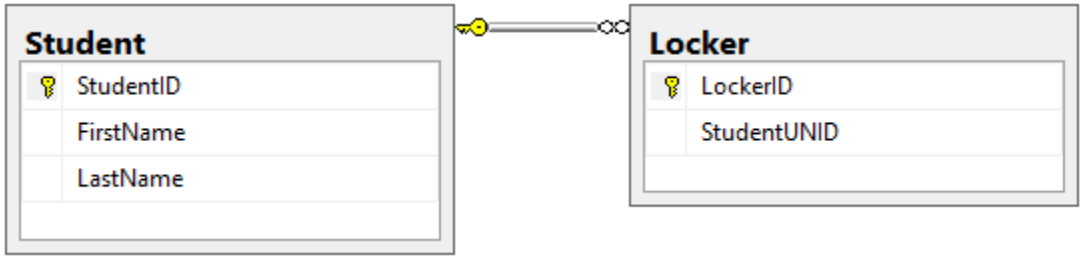
If a primary key constraint applies to more than one column, then it must be created as a table constraint. Multi-column table PRIMARY KEY constraints may only include NOT NULL columns:

```
create table Monkey
(
	[Name] nvarchar (50) not null
		constraint CHK_NAME check (len ([Name]) >= 3)
		constraint UNIQ_NAME unique,
	Age tinyint not null

	constraint pk_Monkey_Name_Age primary key ([Name], Age)
)
```

FOREIGN KEY constraints identify and enforce the relationships between tables, although a FOREIGN KEY can refer to the same table, or columns that are not primary keys. Typically a FOREIGN KEY constraint references a PRIMARY KEY in a different table.

A FOREIGN KEY constraint serves *referential integrity* by ensuring that rows in the foreign table are not orphaned when referenced rows in the primary table are changed. Similarly, a row can't be added to a table with a FOREIGN KEY constraint if the supplied key does not exist in the referenced table.

Consider the following tables, with the following relationship:



The *Student* table contains a primary key *StudentID*, and the *Locker* table contains a FOREIGN KEY constraint on the *StudentUNID* column, binding it to the *StudentID* column in the *Student* table.

This implies, through database engine mechanics, that a locker could not be assigned to a student that does not exist, nor could a student with a locker be deleted. Let's see…

First, look at the existing data in the tables:

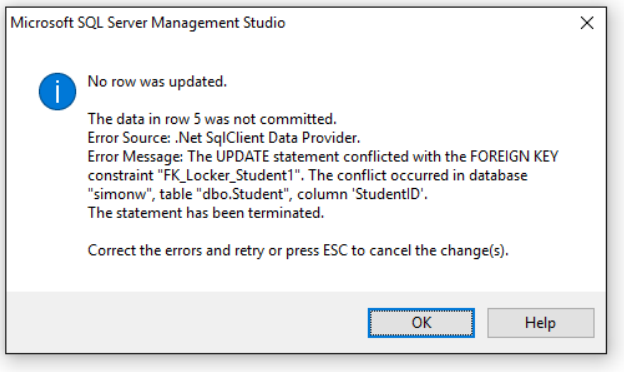
```
select * from Student as S full outer join Locker as L
       on S.StudentID = L.StudentUNID

StudentID  FirstName  LastName      LockerID  StudentUNID
---------  ---------  --------      --------  -----------
123456     Bork       Norgrand      W205-004  123456
589632     Mike       Winderthorpe  NULL      NULL
658412     Brent      MacDormand    W205-003  658412
785684     Bork       Naagland      NULL      NULL
845785     Mike       Dunbar        NULL      NULL
897456     Filbert    Smith         NULL      NULL
NULL       NULL       NULL          W205-001  NULL
NULL       NULL       NULL          W205-002  NULL
NULL       NULL       NULL          W205-005  NULL
```

From the existing data, we can see that

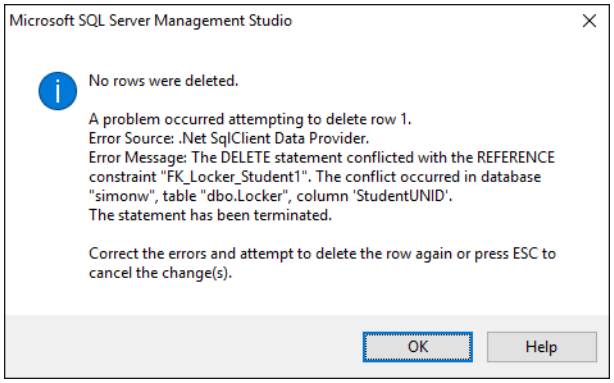
- Some students have a locker
- Some students do not have a locker
- Lockers exist if they are or are not in use

Test number 1: can a locker be assigned to a student that does not exist? Let's try to alter locker W205-005 to have a StudentUNID of 1:

LockerID	StudentUNID
W205-001	*NULL*
W205-002	*NULL*
W205-003	658412
W205-004	123456
W205-005	1
NULL	*NULL*



And what if we attempt to delete a student that has a locker assigned?



So there you have it: automatic referential integrity brought about by a FOREIGN KEY constraint.

What then, is the syntax for a FOREIGN KEY constraint? Creating the constraint will require another table to reference, so we will put the monkey in a zoo:

```
create table fredf_Sample.dbo.Zoo
(
	ZooID int constraint pk_Zoo_ZooID primary key,
	[Name] nvarchar (50) not null
)
go

create table fredf_Sample.dbo.Monkey
(
	MonkeyID int not null constraint pk_Monkey_MonkeyID primary key,
	[Name] nvarchar (50) not null
		constraint CHK_NAME check (len ([Name]) >= 3)
		constraint UNIQ_NAME unique,
	Age tinyint not null
		constraint DV_Age default 1,
	ZooID int null
		constraint fk_Monkey_ZooID foreign key
		references Zoo (ZooID) on delete no action
)
go
```

Note: the definition for the Zoo must come before the FOREIGN KEY constraint that references it. The reverse is also true when dropping the tables: you can't drop a table that a foreign key depends on; the table with the foreign key must be dropped first (or the foreign key constraint removed).

The ON DELETE clause determines what happens when an attempt is made to delete a row that is referenced by a foreign key. NO ACTION means no data will be changed, and an error will be generated. With the NO ACTION option enabled, you are responsible for disassembling the primary/foreign key dependencies in the rows in the correct order before you perform the deletion.

Unlike primary keys, foreign keys can be set to NULL.

ALTER TABLE

You may alter any column definition of an existing table with the ALTER TABLE statement. This is not necessarily practical for the scripting of a complete table definition, as you would instead modify the original table definition, not add corrections to the creation script. ALTER TABLE is particularly useful when you wish to modify an existing table without completely recreating the entire definition, or when dropping the table is not possible. If the table contains data, you may not be able to make table modifications.

The layout or flow of your creation script may benefit from adding constraints after the table definition. Consider the following table creation script (the same table as above, with constraints added after the table definition):

```
create table Monkey
(
	MonkeyID int,
	[Name] nvarchar (50) not null,
	Age tinyint not null,
	ZooID int null
)
go

-- not practical to alter null specifier - do it in original table definition
alter table Monkey alter column MonkeyID int not null
go

-- add primary key constraint:
alter table Monkey add constraint PK_MONKEYID primary key (MonkeyID)
go

-- add a check constraint:
alter table Monkey add constraint CHK_NAME check (len ([Name]) >= 3)
go

-- additions can be grouped into a single alter table statement:
alter table Monkey
	add
		-- add a unique constraint
		constraint UNIQ_NAME unique ([Name]),

		-- add a default constraint (note: unusual form for default)
		constraint DV_AGE default (1) for Age
go

-- add a foreign key constraint (note: round brackets required in both cases)
alter table Monkey add constraint FK_ZOOID foreign key (ZooID) references Zoo (ZooID)
go
```

For the most part, adding the constraint requires the ALTER TABLE statement, followed by the table name, then ADD CONSTRAINT with the constraint name, followed by the constraint type. What comes next depends on the constraint. Note the unusual form for the DEFAULT constraint, which requires the FOR keyword in the definition.

You will be expected to interchangeably use column or table constraint definitions where such use is possible.

The IDENTITY Property

You may designate a column as an identity. Identity columns have a seed and an increment value. New values are generated based on the seed and increment, meaning, you do not provide the value for an identity column - the database engine generates it for you.

An identity is particularly useful when the column is a primary key, as a primary key enforces uniqueness. This is an effective way to generate unique row values that are not related in any way to the data in the row (a surrogate key).

Note that identity values are not guaranteed to be consecutive, and there are many circumstances that can lead to missing values in the identity sequence. In other words, do not depend on sequential values, or contiguous runs of values when using an identity.

Creating a primary key that uses an identity has the following syntax:

```
create table fredf_Sample.dbo.Zoo
(
	ZooID int identity (1000, 1)
		constraint PK_ZOOID primary key,
	[Name] nvarchar (50) not null
)
go
```

Indexes

Simply put, an index is an on-disk data structure that is associated with a table that speeds retrieval of rows from that table. In Microsoft SQL Server, these structures are binary trees of keys that allow the database engine to locate rows associated with these keys very efficiently. An index is either clustered or non-clustered.

Clustered indexes sort and store the data rows based on their key values. There can only be one clustered index per table, as there can only be one ordering used. Effectively, the clustered index is the table data.

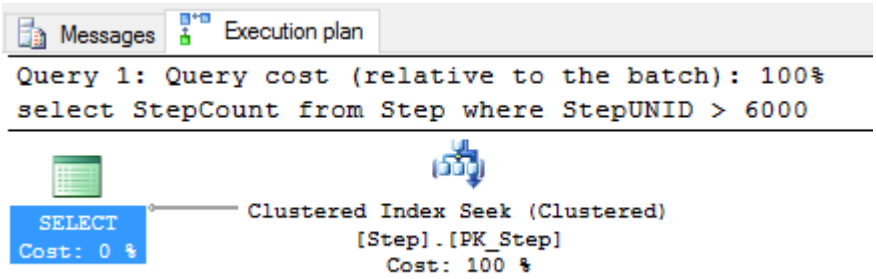
Non-clustered indexes have a structure that is separate from the data rows. In this case, the keys are associated with pointers (row locators) to the data rows.

An index is automatically created for primary keys (usually clustered) and unique constraints (non-clustered).

Indexes can radically improve database performance for searching, but indexes are maintained when data is altered, so performance on insert, update, and delete operations is usually reduced. In general, indexes should be avoided when:

- The table is small
- The table is used with frequent insert or update operations
- Indexed columns will contain a large number of NULL values

You can determine if indexes are being used in SQL Server by inspecting the execution plan for your query:



This information can be helpful in locating bottlenecks in your SELECT statements, and possibly indicating if you should be adding indexes to your columns.

An index may apply to more than one column. In the case of multiple column indexes, the sorting rules follow what you have seen with ORDER BY.

You may add an index to a column with the CREATE CLUSTERED INDEX or CREATE NONCLUSTERED INDEX statements. These statements come after the table has already been created, and have the following form:

```
create table SomeTable
(
	-- implied clustered index (primary key)
	PrimaryID int not null constraint PK_ID primary key,
	ItemName nvarchar (100) null
)
go

create nonclustered index NCI_ItemName on SomeTable (ItemName desc)
go

select ItemName from SomeTable
where ItemName = 'Goats'
```

Without an index, searching for column data may result in a linear search, which may be slow and very resource expensive. This is something to consider when designing your tables.

Execution plan with index on ItemName	Execution plan without index on ItemName