## Notes to accompany SQL course exercises…

With reference to: - Sams teach yourself SQL in 10 Minutes - Fourth Edition

Table of Contents

[Notes to accompany SQL course exercises… 1](#_Toc504928947)

[Loading Sample Files 2](#_Toc504928948)

[Chapter 15 – Inserting Data 3](#_Toc504928949)

[Chapter 16 – Updating & Deleting Data 4](#_Toc504928950)

[Chapter 17 – Creating & Manipulating Tables 5](#_Toc504928951)

[Chapter 18 – Using Views 7](#_Toc504928952)

[Chapter 19 – Stored Procedures 8](#_Toc504928953)

[Chapter 20 – Managing Transactions 10](#_Toc504928954)

[Chapter 21 – Using Cursors 11](#_Toc504928955)

[Example of a Cursor Statement 13](#_Toc504928956)

[Chapter 22 - Constraints 14](#_Toc504928957)

[Understanding Indexes 15](#_Toc504928958)

[Using Triggers 17](#_Toc504928959)

[Exercise: Using Cursors and Stored Procedures 20](#_Toc504928960)

## Loading Sample Files

Sample files are available from the Web page:

<http://forta.com/books/0672336073/> (as referenced Forth Edition 2015 – Appendix A).

Once the SQL Management Studio has been installed, a new database must be created e.g. **SQL\_Course**

For this course, the files are scripted into the created database (**SQL\_Course)** i.e. CREATE TABLE (s) and INSERT INTO. (These are more full covered in Chapters 17 for CREATE TABLE and Chapter 15 INSERT INTO).

The cursor **must** select the newly created database (**SQL\_Course)** in the Object Explorer before any file Importing to *ensure* the files will end up in the Exercise Database **SQL\_Course)**, NOT the Master.

The *alternative* to scripting the files in is to Import (using Tasks/Import Data) the files from another database.

Point to the Microsoft .mdb file and follow the prompts.

If the data is imported from the .mdb database, *no* relationships will be inherited. If needed, then they will have to be added ‘after the fact’ with reference to Chapter 22 – Understanding Constraints.

**NB**

There is a sample backup database in the Backup directory of this CD for continued exercises regarding Cursors and Stored Procedures… refer to Chapter ‘Exercise: Using Cursors and Stored Procedures ‘.

# Chapter 15 – Inserting Data

**Page 146 -** INSERT INTO

Contains two statements

1. **INSERT INTO**
2. **VALUES**

N.B. There are no Column references in the Customers table (poor practice).

**Page 147**- INSERT\_INTO by Column Reference

The INSERTED columns are **explicitly** stated after the table name – NO AMBIGUITY – Best Practice

**Page 148** – Re-Ordering Column references

The Columns have been **explicitly** defined for the INSERT; VALUES must match the Column Order.

**Page 149** – Partial Rows

Suggested that only the cust\_id, cust\_name be INSERTED; all other Columns allow NULL

**Page 150** – Inserting Retrieved Data

This exercise requires that a NEW TABLE be CREATED and POPULATED (INSERT INTO) with data in preparation for the Insertion of retrieved data.

* **NB Lesson 17** must now be addressed to CREATE TABLE in preparation for INSERT INTO Retrieved Data

**Page 163** – **CREATE TABLE**

**CREATE TABLE** CustNew

(

cust\_id CHAR(10) NOT NULL PRIMARY KEY,

cust\_name CHAR(50) NOT NULL,

cust\_address CHAR(50),

cust \_city CHAR(50)

)

Continuing from **CREATE** CustNew table

**INSERT** **INTO** to populate the table as the final step for Inserting Retrieved Data

Two rows have been INSERTED into the CustNew table in preparation for the exercise on **Page 150**

**Page 150 -** INSERT\_INTO\_SELECT

INSERT the rows from another table into this table – this APPENDS rows to an existing table

This IMPORTS data into another table i.e. APPEND

**Page 152 -** SELECT\_INTO\_FROM (Make a Copy of the Table).

CREATES a new table on the fly and appends all (or nominated rows into the new table using WHERE)

This EXPORTS data into a newly created table i.e. MAKE TABLE

# Chapter 16 – Updating & Deleting Data

**Page 156 -** 16\_UPDATE

**UPDATE** modifies row(s) in an existing table

* **UPDATE** the nominated table
* **SET** the Column and Value to be Updated – an Assignment (=)
* **WHERE** the Column to be Updated is referenced (usually the PK)

**Page 156 -** 16\_UPDATE\_Multiple\_Columns

**Page 158 -** 16\_DELETE \_All Rows

Deletes **ALL** rows from a table – NB **No** WHERE Clause

DELETE

FROM Table

**Page 158 -** 16\_DELETE \_WHERE

Deletes **specified** rows from a table conditional upon the WHERE Clause

**DELETE**

**FROM** Table

**WHERE** Row(s) are specified

# Chapter 17 – Creating & Manipulating Tables

**Page 164 -** 17 **CREATE TABLE**

Create a new table by specifying the

1. Table Name
2. Data Type
3. Allow or Not allow NULLS

**DATA TYPES**:

***Strings***:

1. CHAR Fixed Length String 1 to 255 characters
2. VARCHAR National Character (Unicode) Fixed width
3. NVARCHAR National Character (Unicode) Variable width

***Numeric Types:***

1. BIT Single Bit value – Boolean for Flags
2. DECIMAL (**p**, [**s**]) Fixed or Floating Point, varying levels of precision  
   **p** (precision) - The maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18.  
   **s** (scale) - The maximum number of decimal digits that can be stored to the right of the decimal point. Scale must be a value from 0 through p. Scale can be specified only if precision is specified. The default scale is 0; therefore, 0 <= s <= p. Maximum storage sizes vary, based on the precision  
     
   e.g. DECIMAL (10,4) is **10** places to the *right and left* of the decimal point - *Precision*  
   the **4** is the number of decimals to the *right* of the decimal point - *Scale*  
   10 - 4 = 6. **6** is the number of places which can be stored to the *left* of the decimal point
3. FLOAT Floating Point values
4. INTEGER Integer value (+ and –) 2 billion
5. REAL 4-byte Floating Point values
6. SMALLINT (+ and -) 32,000
7. TINYINT 0 – 255 (Positive Only)

***Date and Time***:

1. DATE Date only value
2. TIME Time Only
3. DATETIME Date and Time values

Chapter 17 – Creating & Manipulating Tables (cont.)

**Page 168 -** 17 DEFAULT VALUES

*Default values* can be assigned at table creation time

E.g. **Quantity INTEGER NOT NULL Default 1,**

(By using a Default Value instead of a NULL, Columns that may be required to calculate will not fall over.)

UPDATING TABLES **– ALTER TABLE**

Should be avoided... Best to have an accurate schema before database construction.

**Page 170 -** 17 ALTER TABLE

ALTER TABLE Table name

ADD new Column

ALTER TABLE Table name

DROP COLUMN Column name

**Page 172** - 17 DROP TABLE

Table name – (**NO** warning will be given, nor is there any UNDO!)

# Chapter 18 – Using Views

Views are virtual tables...*they are queries that dynamically retrieve data when used* ...they contain NO data themselves. The data is returned dynamically from all table(s) in the View.

Views are used to *simplify* and allow the *reuse* of SQL statements

**Page 179 - CREATE VIEW**

Both **CREATE VIEW** and **DROP VIEW** have been used.

Two alternative **SELECT** Statements have been used demonstrating simple table (equi) joins, as well as **INNER** Joins.

**Page 176 -** Using the **VIEW**

The VIEW is used as a *virtual table* and the use of the **WHERE** Clause filters the customers that have purchased a particular product

VIEWS can be used for reformatting data prior to selection, filtering unwanted data before selection, or as a precursor to calculating fields.

**Page 183 -** CREATE VIEW

VIEWS with Calculated Fields e.g.

USE SQL\_Course

CREATE VIEW v\_AllCustomerDetails

AS

SELECT C.cust\_id

, O.order\_num

, C.cust\_name

, C.cust\_city

, OI.quantity

, OI.item\_price

, OI.quantity \* OI.item\_price AS **LineItem**

, P.prod\_id

, P.prod\_name

, V.vend\_name

, V.vend\_address

FROM Customers AS C INNER JOIN Orders AS O

ON C.cust\_id = O.cust\_id

INNER JOIN OrderItems AS OI

ON O.order\_num = OI.order\_num

INNER JOIN Products as P

ON OI.prod\_id = P.prod\_id

INNER JOIN Vendors AS V

ON P.vend\_id = V.vend\_id

Calculated fields can be processed within the VIEW to be utilised by other SELECT statements

# Chapter 19 – Stored Procedures

The syntax for creating a STORED PROCEDURE is exactly that.

STORED PROCEDURE – AS

**N.B**. - CREATE PROCEDURE MUST be the FIRST/ONLY statement in this query batch

**Page 194 -** STORED PROCEDURE

Procedure has been modified to demonstrate

1. How a STORED PROCEDURE is CREATED.
2. How to DROP a STORED PROCEDURE.
3. How to DECLARE a variable.
4. How a STORED PROCEDURE uses the EXECUTE statement.

Code for a Stored Procedure to add a new Customer with an incremented cust\_id

SELECT \*

FROM CustNew

--DROP the NewCustomer Stored Procedure

--DROP PROCEDURE NewCustomer

--Create a new stored Procedure for inserting new customers with an incremented cust\_id

CREATE PROCEDURE NewCustomer @cust\_name CHAR (20), @cust\_address CHAR (50), @cust\_city CHAR (50)

AS

--Declare the variable for the new cust\_id

DECLARE @Newcust\_id INTEGER

--Get the current highest cust\_id

SELECT @Newcust\_id = MAX (cust\_id )+ 1

FROM CustNew

--INSERT the New Customer

INSERT INTO CustNew(cust\_id,cust\_name, cust\_address,cust\_city)

VALUES (@Newcust\_id,@cust\_name, @cust\_address, @cust\_city)

--Return the NewCustomer

RETURN @NewCust\_id

--Execute the Stored Procedure

EXECUTE NewCustomer 'Fish Shop', '11 Bean Street', 'Glebe'

Chapter 19 – Stored Procedures (Cont.)

User Defined Stored Procedure – for mathematical calculations.

**DECIMAL** (10, 4) is 10 places to the right and left of the decimal point…

the 4 is the number of decimals returned to the right of the decimal point.

Calculate the number of integer places:

10 - 4 = 6 6 is the number of places which can be stored to the left of the decimal point.

6 integer positions to the left of the decimal and 4 decimals add to 10.

CREATE PROCEDURE CalcVars @VarOne DECIMAL(10, 6), @VarTwo DECIMAL(10, 6)

AS

DECLARE @VarResult DECIMAL(10, 6)

SELECT @Varresult = @varOne \* @varTwo

PRINT @VarResult

RETURN @VarResult

EXEC dbo.CalcVars 122.85 ,81.4 -- returns four (4) decimal places – the maximum allowable

EXEC dbo.CalcVars 122.85 ,81.5 -- cannot return more than four (4) decimal places - ERROR

# Chapter 20 – Managing Transactions

To ensure that batches of SQL statements are executed *in entirety with integrity*, a transaction management regime is employed.

BEGIN TRANSACTION - the commencement of the transaction process

SAVE TRANSACTION - establish point in the database ‘record set’ where the process can return to the *Savepoint*

COMMIT TRANSACTION - all SQL statements between BEGIN and COMMIT will be executed if successful, OR NOT AT ALL if unsuccessful

ROLLBACK TRANSACTION - return to the *Savepoint* in the case that all SQL statements have not been executed  
 completely

**Page 203 -** Using *Savepoints*

SAVE TRANSACTION {Unique Identifier}

ROLLBACK TRANSACTION {Unique Identifier}

**If you commit the transaction, you can't then make a rollback. Do one or the other!**

* **NB Once you BEGIN a transaction, you MUST either ROLLBACK or COMMIT.**
* **The server will expect one of either of these two instructions, or will hang!**

# Chapter 21 – Using Cursors

Cursors are the result dataset of a query. The dataset is ‘loaded’ into the Cursor for processing on a record by record basis. Once the Cursor is stored, applications (SQL code) can scroll or browse up and down through the data.

**Page 207 -** CREATE CURSOR

A Cursor is DECLARED.

DECLARED {CursorName} CURSOR

FOR

SELECT {Fields}

FROM {Table}

DECLARE statements - Declare variables used in the code block

SET\SELECT statements - Initialize the variables to a specific value

DECLARE CURSOR statement - Populate the cursor with values that will be evaluated

OPEN statement - Open the cursor to begin data processing

FETCH NEXT statements - Assign the specific values from the cursor to the variables

NOTE - This logic is used for the initial population before the WHILE statement and then again during each loop in the process as a portion of the WHILE statement

WHILE statement - Condition to begin and continue data processing

BEGIN...END statement - Start and end of the code block

NOTE - Based on the data processing multiple BEGIN...END statements can be used could be just about any DML or administrative logic

CLOSE statement - Releases the current data and associated locks, but permits the cursor to be re-opened

DEALLOCATE statement - Destroys the cursor

**Page 210 -** USING CURSOR(s)

Once DECLARED, Cursors can be OPENED.

Variables are DECLARED to hold the contents of the Cursor.

Code executed against it.

Once the execution of the code has been completed, the Cursor must be CLOSED (each time).

Once a Cursor is CLOSED, it cannot be reused again.

It does not have to be ‘RE-DECLARED’ to be OPENED again, just OPEN it.

Once the Cursor is completely finished with, it is DEALLOCATED to return resources back to the application.

Once DEALOCATED, the Cursor must be DECLARED again before it can be OPENED.

Please note that cursors are the **SLOWEST** way to access data inside SQL Server. They should only be used when you truly need to access a row or object (e.g. a Database) one at a time.

**N.B.**

*There are additional sample files for Cursors in the Section of this Document - Exercise: Using Cursors and Stored Procedures.*

Chapter 21 – Using Cursors (cont.)

## Example of a Cursor Statement

*Copied from mssqltips.com*

This will backup all databases for this instance of SQL Server…

DECLARE @name VARCHAR (50) -- database name

DECLARE @path VARCHAR (256) -- path for backup files

DECLARE @fileName VARCHAR (256) -- filename for backup

DECLARE @fileDate VARCHAR (20) -- used for file name

-- specify database backup directory

SET @path = 'C:\Backup\'

-- specify filename format

SELECT @fileDate = CONVERT(VARCHAR (20), GETDATE(),112) -- returns in the format 20160725

DECLARE db\_cursor CURSOR FOR

SELECT name

FROM master.dbo.sysdatabases

WHERE name NOT IN ('master','model','msdb','tempdb') -- exclude these databases

OPEN db\_cursor

FETCH NEXT FROM db\_cursor INTO @name

WHILE @@FETCH\_STATUS = 0 -- continue until there are no more databases to backup

BEGIN

SET @fileName = @path + @name + '\_' + @fileDate + '.BAK'

BACKUP DATABASE @name TO DISK = @fileName

FETCH NEXT FROM db\_cursor INTO @name

END

CLOSE db\_cursor

DEALLOCATE db\_cursor

# Chapter 22 - Constraints

PRIMARY KEY – A unique identifier for a table; allows the ability to form a relationship with a linked table via a Foreign Key. One Primary Key in the parent table and many of those keys in the linking table. Once associated, Referential Integrity can be enforced to ensure that a row is present in the Primary Key table before any subsequent records (rows) can be added as a Child Record in the linking table.

* ADD Primary Key

-- ADD a Primary Key to a table not keyed

ALTER TABLE CustNew

ADD PRIMARY KEY (cust\_id)

For demonstration purposes, return the name of the Column containing the PK

* SELECT Primary Key

--Select the name of the Primary Key field

SELECT name

FROM sys.key\_constraints

WHERE type = 'PK' AND OBJECT\_NAME(parent\_object\_id) = N'CustNew';

GO

* ADD Foreign Key

--ALTER the table of the Foreign Key

ALTER TABLE Orders

ADD Constraint

--Describe the Relationship between the PK & FK

FOREIGN KEY (cust\_id) REFERENCES Customers (cust\_id)

* Check Constraints – CREATING TABLE

--Define the Column requiring the CHECK at Table creation

CREATE TABLE OrderItems

Quantity INTEGER NOT NULL CHECK (quantity > 0)

* Check Constraints – ALTERING TABLE

--ADD CONSTRAINT to the Column after Table built

ADD CONSTRAINT CHECK (quantity > 0)

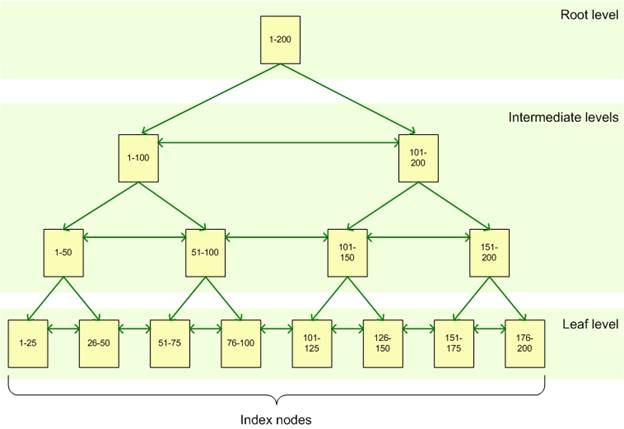
# Understanding Indexes

Indexes are used to sort data in a logical order.

A Primary Keyed Column is a **Clustered Index**.

*There can be only one clustered index per table*, because the data rows themselves can be sorted in **only one order**.

The returned Row is quickly found by the unique sorted Column (usually the PK).



Chapter 22 – Indexes (cont.)

The database also allows for **Non Clustered Indexes**.

Non Clustered Indexes are a separate Index file ordering the required Column alphabetically and associating the location of each Row with the position (Pointer) to the Table.

This is the Clustered Index column

|  |  |  |  |
| --- | --- | --- | --- |
| **id\_PK** | **Item** | **Value** | **Location** |
| 1 | Mon | 54 | South |
| 2 | Tue | 21 | West |
| 3 | Wed | 83 | South |
| 4 | Thu | 24 | North |
| 5 | Fri | 97 | West |
| 6 | Sat | 72 | West |
| 7 | Sun | 41 | North |
| 8 | Mon | 14 | South |
| 9 | Tue | 96 | East |
| 10 | Wed | 50 | South |
| 11 | Thu | 81 | West |
| 12 | Fri | 88 | South |
| 13 | Sat | 88 | North |
| 14 | Sun | 62 | West |
| 15 | Mon | 24 | West |
| 16 | Tue | 70 | North |
| 17 | Wed | 77 | South |
| 18 | Thu | 85 | East |
| 19 | Fri | 35 | South |
| 20 | Sat | 34 | West |
| 21 | Sun | 57 | South |
| 22 | Mon | 73 | North |
| 23 | Tue | 95 | West |
| 24 | Wed | 13 | West |
| 25 | Thu | 85 | North |

This table is ordered by the I**tem Column** for faster access.  
 **Index Table**

|  |  |
| --- | --- |
| **id\_PK** | **Pointer** |
| 5 | Fri |
| 12 | Fri |
| 19 | Fri |
| 1 | Mon |
| 8 | Mon |
| 15 | Mon |
| 22 | Mon |
| 6 | Sat |
| 13 | Sat |
| 20 | Sat |
| 7 | Sun |
| 14 | Sun |
| 21 | Sun |
| 4 | Thu |
| 11 | Thu |
| 18 | Thu |
| 25 | Thu |
| 2 | Tue |
| 9 | Tue |
| 16 | Tue |
| 23 | Tue |
| 3 | Wed |
| 10 | Wed |
| 17 | Wed |
| 24 | Wed |

# Using Triggers

Triggers are created to be fired (triggered) after any **action query**:

* INSERT
* UPDATE
* DELETE

Examples of the uses for triggers would be generating audit tables to record data movements as a result of an action query i.e. writing logging tables.

The following example creates both a Employee\_Test, and an Employee\_Test\_Audit tables for use in both the INSERT and UPDATE triggers. No DELETE trigger has been provided, but the trigger architecture is almost identical.

--CREATE TABLE Employee\_Test

--(

--emp\_ID INT Identity PRIMARY KEY,

--emp\_Name Varchar(100),

--emp\_Sal Decimal (10,2)

--)

--INSERT INTO Employee\_Test VALUES ('Anees',1000);

--INSERT INTO Employee\_Test VALUES ('Rick',1200);

--INSERT INTO Employee\_Test VALUES ('John',1100);

--INSERT INTO Employee\_Test VALUES ('Stephen',1300);

--INSERT INTO Employee\_Test VALUES ('Maria',1400);

--SELECT \*

--FROM Employee\_Test

--CREATE TABLE Employee\_Test\_Audit

--(

--emp\_AuditKey INT Identity PRIMARY KEY,

--emp\_ID int,

--emp\_name varchar(100),

--emp\_Sal decimal (10,2),

--audit\_Action varchar(100),

--audit\_Timestamp datetime

--)

--SELECT \*

--FROM Employee\_Test\_Audit

Chapter 22 – Triggers (cont.)

**This is an After INSERT Trigger**

==============================================================================

--CREATE TRIGGER trgAfterInsert ON [dbo].[Employee\_Test]

--FOR **INSERT**

--AS

-- DECLARE @empid INT;

-- DECLARE @empname VARCHAR(100);

-- DECLARE @empsal DECIMAL(10,2);

-- DECLARE @audit\_action VARCHAR(100);

-- SELECT @empid=i.Emp\_ID FROM inserted AS i;

-- SELECT @empname=i.Emp\_Name FROM inserted AS i;

-- SELECT @empsal=i.Emp\_Sal from inserted AS i;

-- SET @audit\_action='Inserted Record -- After Insert Trigger.';

-- INSERT INTO Employee\_Test\_Audit

-- (emp\_ID, emp\_Name, emp\_Sal, audit\_Action, audit\_Timestamp)

-- VALUES (@empid, @empname, @empsal, @audit\_action, GetDate());

-- PRINT 'AFTER INSERT trigger fired.'

--GO

--INSERT INTO Employee\_Test

--VALUES('Miles',2250);

--SELECT \*

--FROM Employee\_Test

--SELECT \*

--FROM Employee\_Test\_Audit

===============================================================================

Chapter 22 – Triggers (cont)

**This is an After UPDATE Trigger**

--CREATE TRIGGER trgAfterUpdate ON [dbo].[Employee\_Test]

--FOR **UPDATE**

--AS

-- DECLARE @empid INT;

-- DECLARE @empname VARCHAR(100);

-- DECLARE @empsal DECIMAL(10,2);

-- DECLARE @audit\_action VARCHAR(100);

-- SELECT @empid=i.Emp\_ID FROM inserted AS i;

-- SELECT @empname=i.Emp\_Name FROM inserted AS i;

-- SELECT @empsal=i.Emp\_Sal FROM inserted AS i;

-- IF UPDATE(Emp\_Name)

-- SET @audit\_action='Updated Employee Name -- After Update Trigger.';

-- IF UPDATE(Emp\_Sal)

-- SET @audit\_action='Updated Employee Salary -- After Update Trigger.';

-- INSERT INTO Employee\_Test\_Audit(Emp\_ID, Emp\_Name, Emp\_Sal, Audit\_Action, Audit\_Timestamp)

-- VALUES(@empid, @empname, @empsal, @audit\_action, GETDATE());

-- PRINT 'AFTER UPDATE Trigger fired.'

--GO

-- UPDATE Employee\_Test

-- SET Emp\_Sal=1550

-- WHERE Emp\_ID=5

# Exercise: Using Cursors and Stored Procedures

There is a sp\_Exercise file in the Backup directory.

This is a complete database containing a number of Stored Procedures and examples of Cursors using those Stored Procedures.

The Backup file can be Restored to the SQL Server by following the steps below:

1. Ensure you have SQL Management studio open
2. Highlight the Databases folder
3. Right mouse click and select Restore Database
4. In the General page
   1. Source – Select Device
   2. Click on the ellipsis …
   3. Select Backup Device
   4. Add path to the sp\_Exercise file in the Backup Directory
   5. Locate Backup File and Select All Files (\*) from the File name
   6. Select sp\_Exercise
   7. OK, then OK
   8. In the Destination, select the Database
   9. Add **a new database name** e.g. WorkingWithCursors
   10. Hit OK
5. A new database has been Restored to the SQL Server

Scripted exercise references are prefixed with Alpha characters e.g. AAA; BBB in the Project folder.