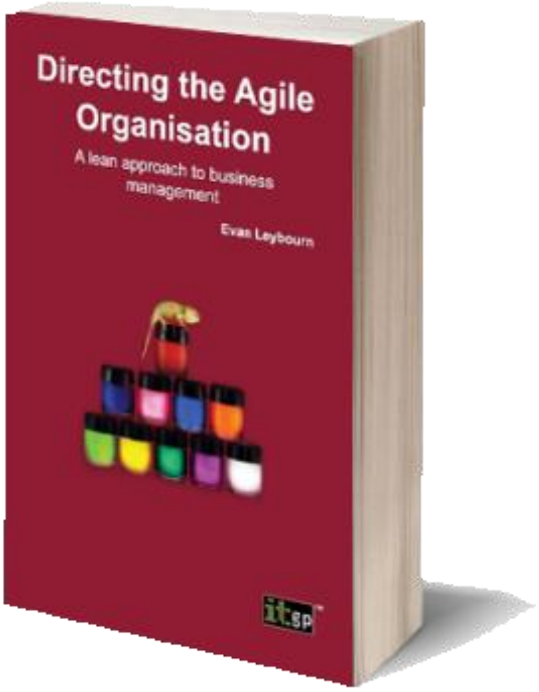
Advanced SQL Testing with SQLServer

Student Guide

Evan Leybourn
evan@theagiledirector.com
Twitter: @eleybourn



OTHER WORKS BY EVAN LEYBOURN

DIRECTING THE AGILE ORGANISATION – BY EVAN LEYBOURN

http://theagiledirector.com/book

- Embrace change and steal a march on your competitors
- Discover the exciting adaptive approach to management
- Become the Agile champion for your organisation

Business systems do not always end up the way that we first plan them. Requirements can change to accommodate a new strategy, a new target or a new competitor. In these circumstances, conventional business management methods often struggle and a different approach is required.

Agile business management is a series of concepts and processes for the day-to-day management of an organisation. As an Agile manager, you need to understand, embody and encourage these concepts. By embracing and shaping change within your organisation you can take advantage of new opportunities and outperform your competition.

Using a combination of first-hand research and in-depth case studies, Directing the Agile Organisation offers a fresh approach to business management, applying Agile processes pioneered In the IT and manufacturing industries.

TABLE OF CONTENTS

1: INTRODUCTION

> *'On two occasions I have been asked, "Pray, Mr Babbage, if you put into the machine wrong figures, will the right answers come out?" [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.'*
>
> *Charles Babbage, 1864*

Notes:

EXAMPLES USED IN THIS COURSE

All examples in this course use the imaginary “Community Bank” database. The Community Bank database stores information about the staff, customers, accounts, loans and branches used by a hypothetical bank. Branch details consist of the branch name and location. Staff details consist of the name, position within the organisation and what branch that a hypothetical employee of the Community Bank works at.

Customer details stored include the customer name, gender, date of birth and their local branch. Each customer can have one or more accounts – details of banking transactions are stored against accounts.

COMMUNITY BANK SCHEMA

```
CREATE TABLE Staff (
      staff_id INT IDENTITY(1,1) PRIMARY KEY,
      given_name VARCHAR(50),
      family_name VARCHAR(50),
      role VARCHAR(50)
)

CREATE TABLE Branch (
      name VARCHAR(50),
      location VARCHAR(50),
      manager INT REFERENCES Staff
            ON DELETE SET NULL
            ON UPDATE CASCADE,
      CONSTRAINT branchpk PRIMARY KEY (name)
)

CREATE TABLE WorksAt (
      branch_id VARCHAR(50) REFERENCES Branch
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      staff_id INT REFERENCES Staff
            ON UPDATE NO ACTION
            ON DELETE NO ACTION,
      CONSTRAINT worksatpk PRIMARY KEY (branch_id, staff_id)
)
```

Notes:

```
CREATE TABLE Customer (
      customer_id INT IDENTITY(1,1),
      given_name VARCHAR(50),
      family_name VARCHAR(50),
      gender CHAR(1),
      dob DATE,
      age INT,
      branch_id VARCHAR(50) REFERENCES Branch
            ON UPDATE CASCADE
            ON DELETE CASCADE,
      CONSTRAINT customerpk PRIMARY KEY (customer_id)
)

CREATE TABLE Account (
      account_id BIGINT IDENTITY(1,1) PRIMARY KEY,
      customer_id INT REFERENCES Customer
            ON UPDATE NO ACTION
            ON DELETE NO ACTION,
      balance NUMERIC CHECK (balance > 0),
      opened_by INT REFERENCES Staff
            ON UPDATE NO ACTION
            ON DELETE NO ACTION
)

CREATE TABLE Transactions (
      account_id BIGINT REFERENCES Account
            ON UPDATE NO ACTION
            ON DELETE NO ACTION,
      transaction_amt NUMERIC,
      date DATETIME
)
```

Notes:

2: PHASES OF DATABASE DESIGN & TESTING

'So Mr Edison, how did it feel to fail 10,000 times?'
'Young man, I didn't fail, I found 9,999 ways that didn't work'

Thomas Edison, anecdotal (on his invention of the incandescent light)

Notes:

TESTING PHASES

Depending on the complexity of the data and business rules involved, testing a database application can be a complicated process. Testing at each stage of the development process eases this complexity.

Six phases of database development commonly used in the industry are:

1. Requirements analysis to identify data and operations
2. Data modelling
3. Database schema design to design detailed tables and entities
4. Database implementation to create instance of schema
5. Build operations and interface (SQL, stored procedures, GUI)
6. Performance tuning

These map roughly to phases in many common project management approaches; however, they should not be mistaken as such – these are purely from the point of view of database developer.

DATA MODELLING

WHAT IS DATA MODELLING?

Data modelling is a process to describe the information that should be contained in the database and the relationships between this information. It is also used to describe any constraints and restrictions on the data.

As a banking database, the Community Bank stores information such as customer records and customer accounts. Each account record is related to a customer record, since each customer holds an account.

Constraints that may be placed on the data can include 7 digit customer ids and reasonable dates of birth.

MODELLING OUTCOMES

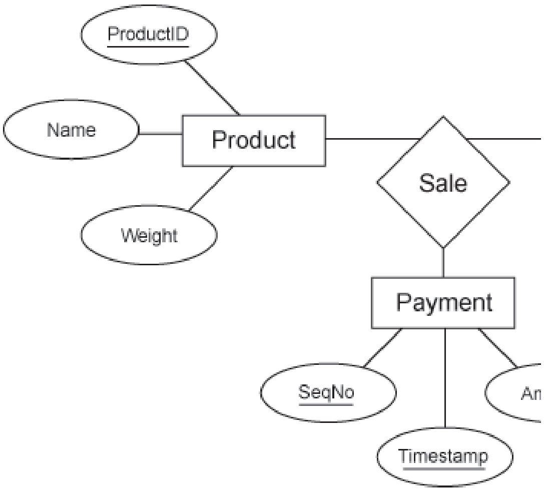
Agreed user and design requirements focus the modelling process. The aim of which is to create a (semi) formal description of the database structure. There are two types of data models, both of which need to be validated against the user requirements;

Notes:

1. Logical models which represent the conceptual structure of the data. These models are created from the user requirements and need to be validated against business need.
2. Physical models which represent the physical layout of data and need to be tested for performance and data integrity. Underlying defects identified at this stage will significantly improve development and testing in later stages.

ENTITY-RELATIONSHIP (ER) MODEL

The ER Model (or ER diagram) is the most common modelling method for database design. The ER model represents the data as a collection of interrelated entities. It is a notation for describing entities and their relationships.



It is also very simple to understand and validate, representing the information in an abstract and graphical form, which allows clients and developers to both use the model to understand the design.

This modelling method has existed for almost 40 years. The form of Entity Relationship Diagrams has never been standardised; though some loose attempts have been made (the class diagram in Unified Modelling Language, for example, and the ICAM Definition Languages). As a result, many variations do (unfortunately) exist. Common variations involve relationship cardinalities (which will be covered later) and Object Oriented (OO) extensions (which won't).

ER diagrams are a graphical tool for data modelling. An ER diagram consists of:

- The definitions of each entity set
- The definitions of each relationship set
- The attributes associated with each entity set and
- The connections between entity and relationship sets

Notes:

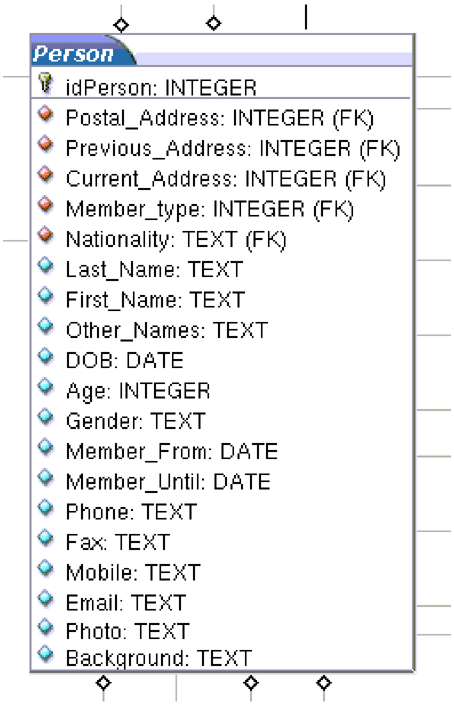
ENTITIES AND ENTITY SETS

Each entity is described by a collection of attributes that describe it.

```
The customer table.
	Family Name:		Smith
	Given Names:	Jenny
	DOB:		01-Jan-1970
	Gender:		Female


The branch table.
	Name:			Canberra City
	Location:		Canberra
```

An entity set is a collection of entities with the same attributes. It should be noted that 99% of the time we say "entity" when we mean "entity set" and we say "relationship" when we mean "relationship set".



Person

idPerson: INTEGER
Postal_Address: INTEGER (FK)
Previous_Address: INTEGER (FK)
Current_Address: INTEGER (FK)
Member_type: INTEGER (FK)
Nationality: TEXT (FK)
Last_Name: TEXT
First_Name: TEXT
Other_Names: TEXT
DOB: DATE
Age: INTEGER
Gender: TEXT
Member_From: DATE
Member_Until: DATE
Phone: TEXT
Fax: TEXT
Mobile: TEXT
Email: TEXT
Photo: TEXT
Background: TEXT

Notes:

ATTRIBUTES

Each attribute in an ER diagram has a name (which appears on the ER diagram) and is associated with an entity or relationship set. Attributes are assigned types (domains) which constrain the data allowed within the attribute.

NULL VALUES

Attributes may contain a null (or empty) value to indicate that the attribute is not relevant, or that the value of the attribute is unknown for a particular entity.

> *Renae is under 18 years old and will not have a credit card number. We also do not know her mobile phone number, so both of these values will be NULL.*

COMPUTED (OR DERIVED) ATTRIBUTES

Derived attributes contain values that are calculated from other attributes. The rules of which need to be thoroughly tested to ensure data integrity. It needs to be understood that persisted computed columns are calculated on insert, and non-persisted computed columns are calculated on select. This may have implications for testing, both in terms of accuracy and performance.

> *A person's age is be calculated by subtracting the year of their birth from the current year.*

Most databases have a large number of available functions for each data type which will be discussed later in this course.

PRIMARY KEYS

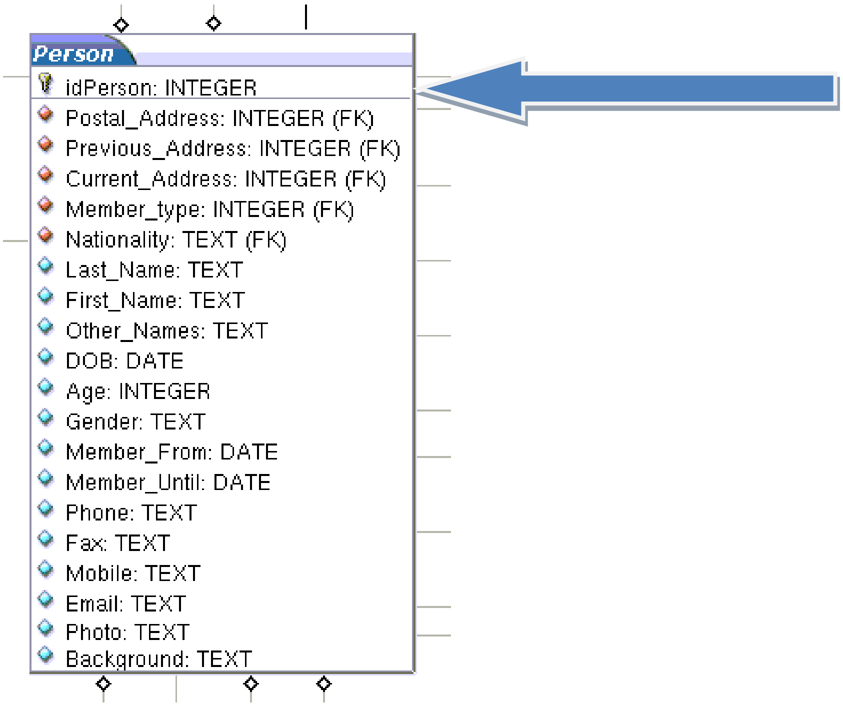
In order to identify an individual entity within an entity set, each entity must contain an attribute which is unique, or a series of attributes which, in combination, are unique. Sometimes the nature of the data enforces uniqueness across one or more attributes within an entity, and this/these can be used as a primary key. Unless there a very good reasons for the contrary, all tables must contain a primary key.

Another approach to ensuring this uniqueness is to create a new attribute which contains a guaranteed unique value for each entity in the entity set, and using this as the primary key. This approach is commonly used .

Notes:

> *The Community Bank branch entity uses the name of each Branch as a Primary Key, as no two branches can have identical names.*
>
> *However, as no attribute or combination of attributes in the Customer table can be guaranteed to always be unique, a unique customer id is given to each row.*

If the key is made from more than one attribute it is called a composite key. Keys are indicated in ER diagrams by underlining the key attributes; it is also common to denote a non-composite Primary Key with the symbol (PK).



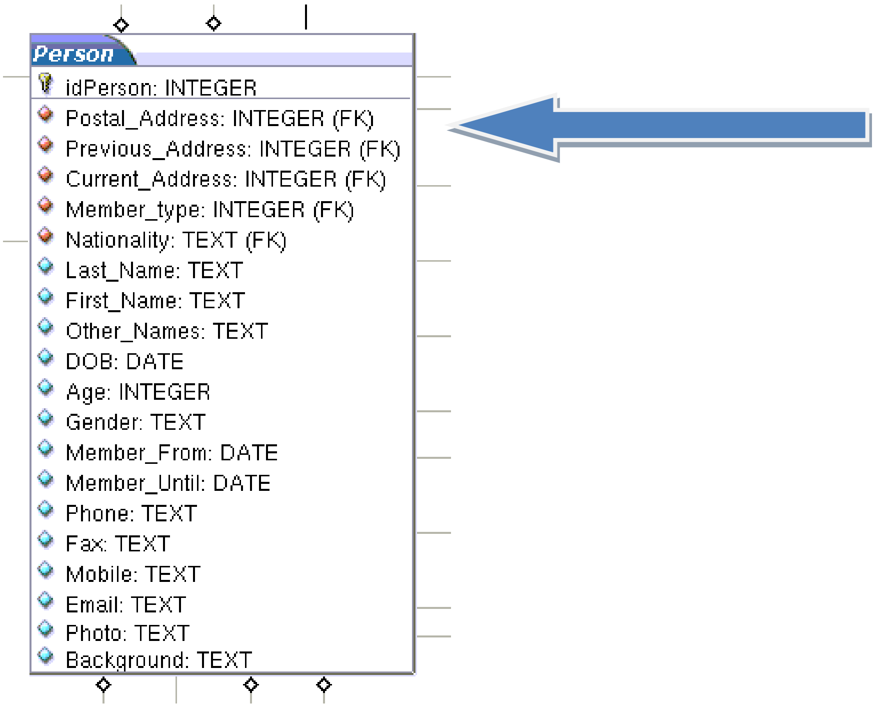
Notes:

FOREIGN KEYS

Given that a primary key is one or more attributes identifying a specific entity in an entity set, a foreign key is one or more attributes identifying an entity in another entity set. Foreign keys are critical in relational databases, they link individual relations into a cohesive database structure.

They are also very important when querying data as they provide the basis for connecting individual relations to assemble the results.

> *A customer's address is stored in a special Address table. There are Foreign Keys linking the Person table to the Address table.*



Person
idPerson: INTEGER
Postal_Address: INTEGER (FK)
Previous_Address: INTEGER (FK)
Current_Address: INTEGER (FK)
Member_type: INTEGER (FK)
Nationality: TEXT (FK)
Last_Name: TEXT
First_Name: TEXT
Other_Names: TEXT
DOB: DATE
Age: INTEGER
Gender: TEXT
Member_From: DATE
Member_Until: DATE
Phone: TEXT
Fax: TEXT
Mobile: TEXT
Email: TEXT
Photo: TEXT
Background: TEXT

Notes:

RELATIONS

A relationship is an association between two entities. Similarly, a relationship set is a collection of relationships of the same type.

> *Customer(9876) holds Account(12345)*
>
> *Customer(9877) holds Account(12346)*
>
> *Customer(9878) holds Account(12347)*
>
> *Customer(9878) also holds Account(12348)*
>
> *etc.*

CARDINALITIES

Cardinalities describe the number of entities that a given entity can be associated with through a relationship.

One-to-One (1-1): Each X is associated with at most one Y and each Y is associated with at most one X.

> *Every branch has one staff member as manager, and a staff member can only be a manager for one branch.*

One-to-Many (1-M): each X is associated with zero or more Y, each Y is associated with at most one Y.

> *A customer can hold many accounts, but an account belongs to only one customer.*

Many-to-Many (M-N): each X is associated with zero or more Y, each Y is associated with zero or more X. M-N relationships usually degrade into a weak entity between the two main entities. This will be covered later.

> *A staff member can work at many branches, and a branch has many staff working there.*

Notes:

FOREIGN KEY PARTICIPATION

Foreign Keys can be NULL, indicating that though a relationship is possible for a generalised entity set, a specific entity does not have one. This means that the level of participation in a relationship is another type of constraint. Participation can be total or partial;

Total: Every X must have at least 1 Y

> *A customer must have an account to have a customer record at the Bank.*

Partial: Every X can have 0 or more Y

> *An account may have no transactions on record, such as if it has just been opened.*

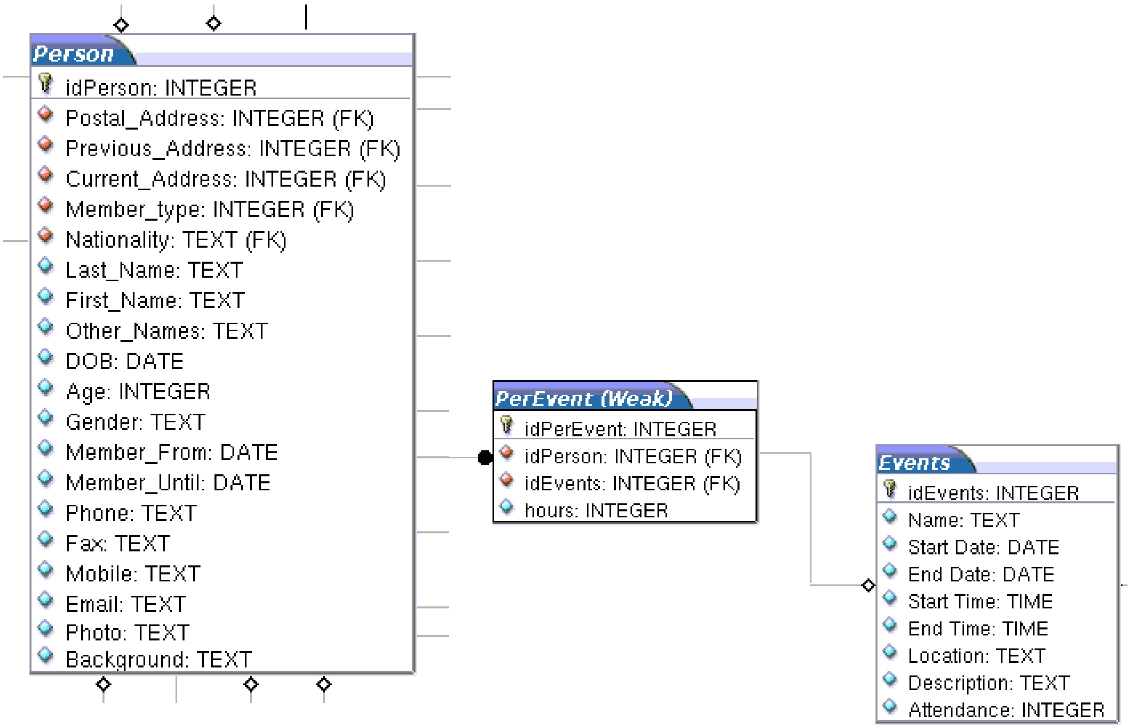
WEAK ENTITY SETS

Weak entities exist only because of a relationship between two or more entities. They only contain the keys of the “strong” entities and attributes directly relating to the relationship. They do not have a separate primary key.

> *Staff members who work at a branch is a M-N relationship. We cannot store the information about the relationship in either the Staff or Branch tables, so a WorksAt table (containing the primary keys from the Staff table, matched against the relevant primary keys of the Branch table) is created to store this data.*
>
> *Likewise if a staff member does not work at a branch, there is no need to store any information about it.*

We can form a primary key for a weak entity by taking a combination of the primary keys of the associated strong entities. In ER diagrams weak entities are denoted by double-boxes and discriminators are denoted by dotted underline.

Notes:



THE RELATIONAL DATA MODEL

The relational data model has existed for over 30 years and has created numerous database design methodologies. It has also helped to develop the standard database access language, SQL.

The relational model is a mathematical theory; it has no "standard" and is based heavily on set theory. There are thus two kinds of terminology in use:

- Mathematical: relation, tuple, attribute, etc
- Physical: table, record, field/column, etc

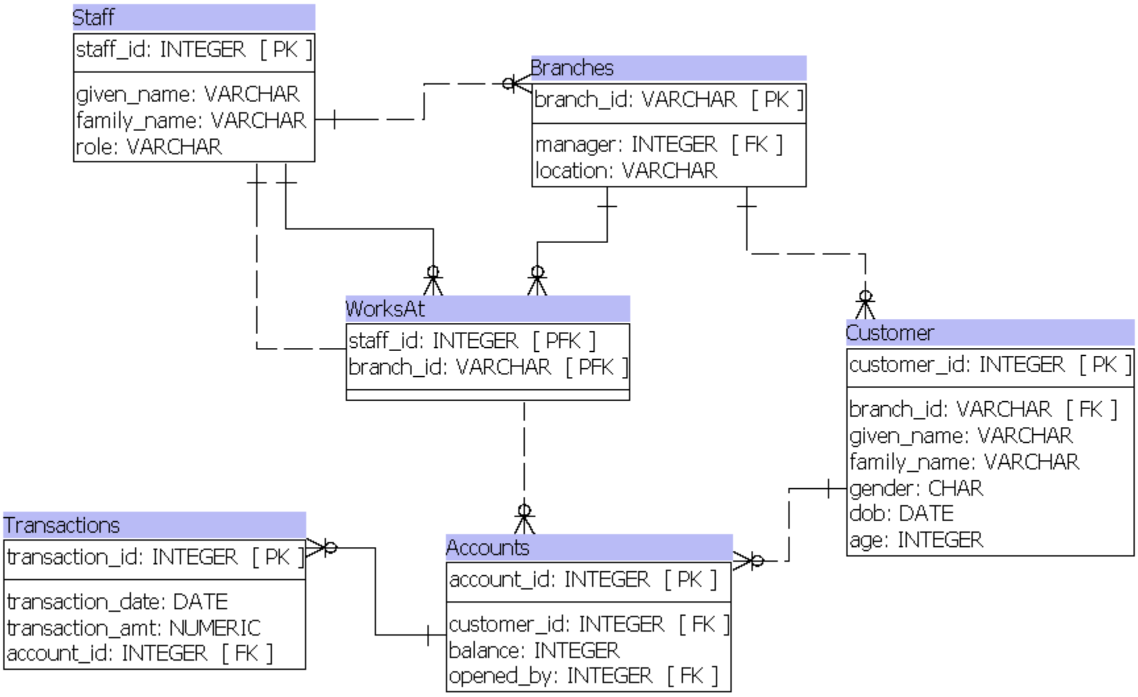
CORRECTNESS OF DESIGNS

In general, there is no single "best" design for any given application. If a model contains all available data, relations and constraints it can be considered 'correct'. However, we may describe a design as incorrect or inadequate if it:

Notes:

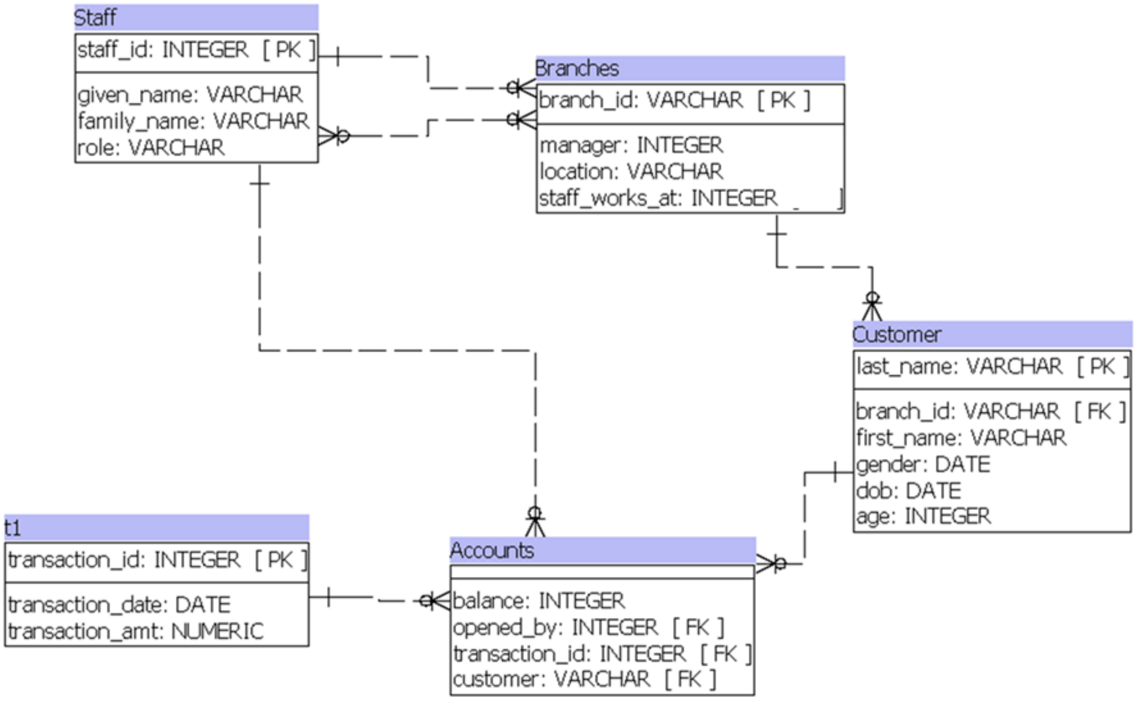
- Omits some information that is meant to be modelled
- Contains redundant information, though this may be unavoidable
- Leads to an inefficient implementation
- Violates the rules of the chosen modelling method

For example; an initial data model did not store the balance of a customer's accounts. It also stored the customer name in both the Customer and Account tables. And because it did not store the account balance, any balance check required it to be calculated from all previous transactions.



A "correct" ER Diagram

Notes:



An “incorrect” ER Diagram. Some of the issues within this ER diagram include;

- Meaningless tablename (t1)
- Incorrect type (Gender as date & balance as integer)
- Incorrect cardinality (1-M on the wrong side) between t1 & Accounts
- M-N relationship without weak entity between Staff & Branches
- Accounts entity without primary key
- Customer does not have a unique primary key
- Branches foreign key is not labelled
- Unclear label in Customer table (first name / surname rather than given / family)

Notes:

3: SQLSERVER (DATABASE MANAGEMENT SYSTEMS)

'It is always wise to look ahead, but difficult to look further than you can see.'

Winston Churchill, ~1960

Notes:

CONSTRAINT CHECKING

When a process creates, changes or deletes a record, SQLServer checks that the new values do not break the constraints against each attribute. As long as the constraints are configured correctly, it is not necessary to test any of these elements. If any constraint is not met, the operation is cancelled and any prior changes in the operation are rolled back.

TYPE CHECKING

Each attribute in a database is given a type (or domain). New and updated values must be of the same type. No checking is done for deleted rows.

> *A person's date of birth is not allowed in the age field. Likewise the date of birth must be a valid date. If this constraint is not met, the row will not be inserted or updated.*

PRIMARY KEY CHECKS

As each row in the database must have a unique primary key, new and updated values must not occur elsewhere in the entity. No checking is done for deleted rows.

> *No two branches are allowed to have the same name.*

REFERENTIAL INTEGRITY CHECKS

For attributes that are foreign keys, any new or updated values must exist in the parent table. If an attempt is made to update or delete a record in a parent table then SQLServer may:

- Abort the change and the user must find all referring entities and either remove each one manually or change their foreign keys to an acceptable value.
- Remove or update all referring entities automatically.
  ON DELETE/UPDATE CASCADE
- Set foreign key attributes to NULL in all referring entities.
  ON DELETE/UPDATE SET NULL

> *Customer records are considered very important data, much more so than branch data. So if the branch a customer belongs to is deleted from the database, the branch_id attribute will be set to NULL.*

```
branch_id INT REFERENCES Branch ON DELETE SET NULL
```

Notes:

> *However if that same branch is deleted, the information describing staff working at a branch is no longer important and can be deleted.*
> ```
> branch_id INT REFERENCES Branch ON DELETE CASCADE
> ```

ACCESSING DATA

Like all modern DBMSs, SQLServer provide access to the data via SQL. And while SQLServer uses its own dialect of SQL (Transact-SQL or T-SQL), SQL:2011 is the current standard.

In SQLServer, SQL queries can be directly entered via an interactive shell, the SQLServer Management Studio (SSMS). This interface is useful for administrative and debugging purposes. Although, users would never be expected to use this shell.

APIs included in most programming languages, (Java, C, PHP, Python, etc) allow developers to connect to and communicate with SQLServer from within their application. Users then use the front-end application and remain blissfully ignorant of SQL or the database backend.

SQLServer also provides various kinds of extensibility of the database such as views, stored procedures and triggers.

INFORMATION SCHEMA

SQLServer also provides access to database metadata, or the catalogue. Metadata is typically presented as collection of tables that can be queried as you would a standard entity. This is available through the through the Information_Schema tables.

There are a variety of uses for this metadata, ranging from the simple to the complex. Many test cases will use this metadata to do analysis of the data stored, and provide statistics and other relevant information.

Software has also been written that takes advantage of the database metadata to create simple CRUD (create, retrieve, update, delete) applications that generate a user interface on the fly from nothing more than the database metadata.

```
INFORMATION_SCHEMA.CHECK_CONSTRAINTS
INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE
INFORMATION_SCHEMA.COLUMN_PRIVILEGES
INFORMATION_SCHEMA.COLUMNS
```

Notes:

```
INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE
INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS
INFORMATION_SCHEMA.DOMAINS
INFORMATION_SCHEMA.KEY_COLUMN_USAGE
INFORMATION_SCHEMA.PARAMETERS
INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
INFORMATION_SCHEMA.ROUTINE_COLUMNS
INFORMATION_SCHEMA.ROUTINES
INFORMATION_SCHEMA.SCHEMATA
INFORMATION_SCHEMA.TABLE_CONSTRAINTS
INFORMATION_SCHEMA.TABLE_PRIVILEGES
INFORMATION_SCHEMA.TABLES
INFORMATION_SCHEMA.VIEW_COLUMN_USAGE
INFORMATION_SCHEMA.VIEW_TABLE_USAGE
INFORMATION_SCHEMA.VIEWS
```

TRANSACTIONS AND CONCURRENCY

Often in application programming a single application-level operation or transaction involves multiple DBMS-level operations. To faithfully represent the application-level operation, either all DBMS-level operations must complete or all DBMS-level operations must fail. If the transaction fails partway, any completed operations must be undone. While, SQLServer should enforce this automatically, test cases need to be written to validate these transactions.

> *When a customer withdraws money from their account, a financial transaction record is created recording this information. One recorded the balance of the customer's account is reduced by the withdrawal amount. If the either of these processes fail, the other process needs to be aborted to ensure all records are correct (Data Integrity).*

```
BEGIN TRANSACTION tran_name
      BEGIN TRY
            UPDATE Customer SET <FAILS>
            COMMIT TRANSACTION tran_name
      END TRY
      BEGIN CATCH
            ROLLBACK TRANSACTION tran_name
      END CATCH
```

Notes:

MULTI-VERSION CONCURRENCY CONTROL

SQLServer uses Multiversion Concurrency Control (MVCC), when using READ_COMMITTED_SNAPSHOT to ensure that users reading data do not need to wait for users writing data to finish, and that the reads will always be accurate and complete.

When a user sends a SELECT query, SQLServer displays a snapshot, or version, of all the data that was committed before the query began. Any data updates or inserts that are part of open transactions or were committed after the query began will not be displayed.

ACCESSING AND TESTING DATABASES

Each programming language will have different syntax to connect to and communicate with a database. However most follow a fairly similar process.

```
--  establish connection to SQLServer
db = dbConnect("dbname=X user=Y passwd=Z");
query = "SELECT a,b FROM R,S WHERE ... ";
--  invoke query and return result set
results = dbQuery(db, query);
--  for each tuple in result set
while (tuple = dbNext(results)) {
      --  do something
}
dbClose(results);
```

The application connects to the database with a username and password which returns a connection (handle) to the database. This connection is used to send and receive data.

The test cases (or queries) are then composed and sent through the connection to the database which returns a database result set. This result set can then be iterated through to get each row of data.

Notes:

4: SQL

'Make everything as simple as possible, but not simpler.'

Albert Einstein (paraphrased), 1933

Notes:

STRUCTURED QUERY LANGUAGE

SQL stands for “Structured Query Language“ which is sometimes called "sequel".

SQL is an ANSI/ISO standard language for querying and manipulating relational databases. It is designed to be a "human readable" language comprising data definition facilities, database modification operations, relational algebra operations and aggregation operations.

SQLServer implements a version of the SQL standard called Transact-SQL or T-SQL. This is very similar to the SQL standard, but has a number of differences including the use of FROM clauses in UPDATE & DELETE statements and the inclusion of TRY/CATCH blocks.

SQL SYNTAX

SQL identifiers and keywords are case insensitive though as a general rule write keywords (SELECT, WHERE, CREATE) in upper case, relation names (Customer, Account) with an initial upper-case letter and write attribute names (balance, dob) in all lower-case. Multi word attribute names should be separated with an underscore (customer_id, family_name).

SQL KEYWORDS

There are 185 reserved words in T-SQL. A categorised list of the more frequently used keywords:

Querying	Defining	Changing
SELECT	CREATE	INSERT
FROM	TABLE	INTO
WHERE	VIEW	VALUES
GROUP	INDEX	UPDATE
HAVING	COLUMN	SET
ORDER BY	DATABASE	DELETE
DESC	KEY	DROP
ASC	PRIMARY	ALTER
EXISTS	FOREIGN	BEGIN
IS	REFERENCES	END
NOT	CONSTRAINT	
NULL	CHECK	
IN	UNIQUE	
DISTINCT		

Notes:

AS
AND
OR
BETWEEN
ROLLBACK
CASCADE
COMMIT
JOIN
LIKE

SQL DATA TYPES (DOMAINS)

SQL supports a small set of useful built-in data types:

- Text string: “Hello World!”
- Numbers (Integers, Real): 123, 3.14
- Dates and Times: 13/05/1981, 10:00
- Boolean (Bit): 0, 1 (t, f)
- Bit-string or binary data

Basic type (domain) checking is performed automatically by SQLServer. The NULL value is treated as a member of all data types. These do not need to be further tested.

Constraints can be used to enforce more complex domain membership conditions depending on the requirements of the application.

SQL OPERATORS

When comparing data a series of default comparison operators are defined on all data types.

- Less Than (<): 1 < 2
- Greater Than (>): 2 > 1
- Less Than or Equal To (<=): 2 <= 2
- Greater Than or Equal To (>=): 2 >= 2
- Equals (=): “Hello” = “Hello”

Notes:

- Not Equals (!= or <>): 10:00 != 11:00
- Test for NULL value (IS)
- Test for false NULL value (IS NOT)

A common error that needs to be tested for is the use of:

```
= NULL
```

This is incorrect syntax and will lead to unexpected results. The correct syntax is:

```
IS NULL / IS NOT NULL
```

Boolean operators AND, OR, NOT are available within WHERE expressions to combine results of comparisons. Most data types also have type-specific operations available (e.g. arithmetic for numbers).

NUMERIC TYPES

There are different number types available in SQLServer. Each is appropriate for different applications.

INTEGER (or INT), BIGINT, SMALLINT, TINYINT: 4/8/2/1 byte whole numbers.

REAL, FLOAT: 4/4-8 byte floating point (decimal) numbers.

NUMERIC: Exact decimal numbers. Slow but accurate, best used for currency calculations.

INCREMENTING

The T-SQL keyword IDENTITY specifies that a numeric type field is to be automatically incremented when a new record is inserted. There are two parameters, the seed (or starting number), and the increment size. For example;

```
user_id INT IDENTITY(1, 1);
```

This defines a field starting at 1 and incremented by 1 each time. Whereas;

```
user_id INT IDENTITY(10000, 3);
```

Notes:

This defines a field starting at 10,000 and incremented by 3 each time. That is the second row will be 10,003, the third 10,006, and so on.

COMMON FUNCTIONS

+ - * /: The standard operations

```
1+1 = 2
1-1 = 0
2*2 = 4
4/2 = 2
```

sin(x), cos(x), tan(x) etc: Trigonometric

```
sin(90) = 0.893996663600558
```

abs(x): The absolute value of x

```
abs(-1) = 1
```

ceiling(x): The small integer above x

```
ceil(4.5) = 5
```

floor(x): The largest integer below x

```
floor(4.5) = 4
```

power(x, y): x to the power of y

```
power(2, 3) = 8
```

sqrt(x): The square root of x

```
sqrt(4) = 2
```

round(x, y): x to y decimal points

```
round(3.14159265359, 2) = 3.14
```

Notes:

random(): Between 0.0 and 1.0

```
random() = 0.341076350305229
```

STRING MANIPULATION

There are three different text or string types available in SQL Server. Each is appropriate for different applications.

CHAR(n): Fixed length text padding with spaces to n characters.

VARCHAR(n): Variable length text with a predefined limit n.

NCHAR(n)/NVARCHAR(n): Identical to CHAR and VARCHAR, except these allow the use of Unicode characters. These should be used in preference to CHAR or VARCHAR expect where there are justifiable reasons.

TEXT/NTEXT: Variable length text of (effectively) unlimited length.

COMMON FUNCTIONS

str1 + str2: Concatenate two strings

```
'Jenny' + ' ' + 'Smith' = 'Jenny Smith'
```

LEN(str): Return length of string (also valid for Unicode strings)

```
LENGTH('Tobias') = 6
```

SUBSTRING(str,start,count): Extract characters from within a string

```
SUBSTR('Canberra', 1, 3) = 'Can'
```

LOWER(str): Convert to lowercase

```
LOWER('NSW') = 'nsw'
```

UPPER(str): Convert to uppercase

```
UPPER('nsw') = 'NSW'
```

Notes:

POSITION(substring, string, start location): Location of the substring within the string, searching from the option start location.

```
POSITION(' ', 'Tobias Cruz') = 7
```

LEFT(string, length): Returns the left part of the given string with 'length' characters.

```
LEFT('Ashley Jones', 3) = 'Ash'
```

RIGHT(string, length): Returns the right part of the given string with 'length' characters.

```
RIGHT('Ashley Jones', 3) = 'nes'
```

str LIKE pattern: Matches string to pattern (not regular expressions). “%” matches anything and “_” matches any single character.

```
--Name begins with Je
first_name LIKE 'Je%'

--Name has 'i' as 2nd letter
first_name LIKE '_e%'

--Name contains an E or e anywhere
first_name ILIKE '%e%'
```

DATE AND TIME MANIPULATION

There are three primary date and time types available in SQL Server. Each is appropriate for different applications.

DATETIME: A date + time data type (Note: This is equivalent to the TIMESTAMP data type as defined in the SQL Standard, but NOT related to the TIMESTAMP data type as defined by T-SQL).

DATETIMEOFFSET: A date + time data type that takes into account time zones.

DATE: A date data type

TIME: A time data type

Notes:

COMMON FUNCTIONS

GETDATE(): Get the current date

DATEDIFF(datepart, datetime, datetime): Subtract arguments, and produce the “age” (as measured by the date part) of the result. You can use GetDate() in the seconds datetime to get the age as of now.

```
DATEDIFF('Year', '2001-04-10', '1957-06-13') = 43 years
DATEDIFF('Day', '2001-04-10', '1957-06-13') = 16007 days
```

DAY/MONTH/YEAR(datetime): Returns the day, month or year from the specified datetime.

```
DAY('2001-02-16 20:38:40')    = 16
MONTH('2001-02-16 20:38:40')  = 2
YEAR('2001-02-16 20:38:40')   = 2001
```

DATEPART(datepart, datetime): Returns the specified part of the specified datetime. DATEPART('day', '…') is equivalent to DAY('...')

AGGREGATIONS

Aggregations can apply to a column of numbers in a relation:

COUNT(col): Number of rows in the column. Unlike the other aggregation functions, count can apply to any data type.

Sum, average, maximum and minimum can only be used with numeric data types.

SUM(col): Sum of all values in the column. Can only be used with numeric types.

AVG(col): Average of the values in the column.

MIN/MAX(attr): Min/max of values for attr

NULL OPERATORS

A NULL value in any arithmetic operation always yields NULL.

```
3 + NULL = NULL
1 / NULL = NULL
```

Notes:

NULL in aggregations is ignored (treated as unknown)

```
sum(1,2,3,4,5,6) = 21
sum(1,2,NULL,4,NULL,6) = 13
avg(1,2,3,4,5) = 3
avg(NULL,2,NULL,4) = 3
```

OTHER DATA TYPES

SQLServer supports many other data types such as XML and Geospatial data types. Some of these extend the SQL standard and so other DBMS systems may not have these types.

USER-DEFINED DATA TYPES

SQLServer has several basic data types which are suitable for most applications, e.g. INT, CHAR, BOOL etc. However sometimes these are not enough and additional data types are required. These can be created through the CREATE TYPE command.

```
CREATE TYPE NewDataType FROM ExistingDataType
```

Example:

```
CREATE TYPE postcode FROM CHAR(5) NOT NULL;
```

SQL DATA DEFINITION LANGUAGE

SQL is normally considered to be a query language. However, it also has a data definition sub-language (DDL) for describing database schemas.

Each database description contains the names of all the tables and the names and types for each column. The description also describes all the various types of constraints, such as primary and foreign keys, unique and not null.

Notes:

DEFINING A DATABASE SCHEMA

CREATING TABLES

Relations (tables) are described using:

```
CREATE TABLE RelName (
    attribute1 type constraints,
    attribute2 type constraints,
    ...
    table-level constraints, ...
)
```

Community Bank Branch Table

```
CREATE TABLE Branch (
    name VARCHAR(15),
    location VARCHAR(99) UNIQUE NOT NULL
)
```

This will define the table schema and create an empty instance of the table. Constraints can include details about primary keys, foreign keys, default values, and constraints on attribute values.

DELETING TABLES

Tables are removed via

```
DROP TABLE RelName;
```

PRIMARY KEYS IN SQL

The primary key needs to be defined in the CONSTRAINT pragma of the CREATE TABLE query.

```
CREATE TABLE Branch (
    name VARCHAR(15) NOT NULL,
    location VARCHAR(99) UNIQUE NOT NULL,
    CONSTRAINT branchpk PRIMARY KEY CLUSTERED (name)
)
```

Notes:

If the primary key is built from multiple attributes, it is declared in the same way.

```
CREATE TABLE WorksAt (
      branchid VARCHAR(50) REFERENCES Branch,
      staffid INT REFERENCES Staff,
      CONSTRAINT workspk PRIMARY KEY (branchid, staffid)
)
```

FOREIGN KEYS IN SQL

Declaring foreign keys in the SQL assures referential integrity when creating, updating or deleting data.

Like the primary keys, if the foreign key is a single attribute it can be declared inline.

```
customer_id INT REFERENCES Customer(customerid)
customer_id INT REFERENCES Customer
```

If the foreign key is built from several attributes, it can be specified in table constraints.

```
FOREIGN KEY (customer, name)
      REFERENCES Customer(customer, name)
```

While you do not need to write tests to validate the foreign keys, you do need to ensure that they are present.

BEHAVIOUR

The default behaviour when a referential conflict is identified it to cancel the operation. Other behaviours include setting the referring attribute to NULL, or cascading the change.

Cascading changes will delete all referring rows when a parent row is deleted, and change all the referring attributes to the new value when a parent key is updated.

```
customer_id INT REFERENCES Customer
      ON DELETE NO ACTION
customer_id INT REFERENCES Customer
      ON DELETE CASCADE
customer_id INT REFERENCES Customer
      ON DELETE SET NULL
```

Notes:

OTHER ATTRIBUTE PROPERTIES

NULL

To specify that an attribute must cannot be null or have an empty value.

```
location VARCHAR(50) NOT NULL
```

UNIQUE

To specify that an attribute must have a unique value. This actually creates a UNIQUE index against this column.

```
location VARCHAR(50) UNIQUE
```

Primary keys are automatically UNIQUE and NOT NULL.

DEFAULTS

The default value of an attribute will be assigned if no value is supplied during an insert.

```
age INT DEFAULT 18
```

CONSTRAINTS

To specify an arbitrary constraint against any attribute using the check syntax. This condition can be arbitrarily complex, and may even involve other attributes, relations and SELECT queries.

```
gender CHAR CHECK (gender IN ('M','F'))
```

QUERIES

An SQL query is a declarative program that retrieves data from a database. The most common kind of SQL statement is the SELECT query:

```
SELECT attributes FROM relations WHERE condition
SELECT * FROM Customer WHERE gender= 'M';
```

The conditions applied to the query can be an arbitrarily complex boolean-valued expression using the operators mentioned previously.

Notes:

SEMANTICS OF SELECT

It is possible to select all the columns from one or more tables. The symbol * denotes a list of all attributes.

```
SELECT * FROM Branch;
```

To alias a column in the output results use the AS syntax.

```
SELECT family_name AS customer FROM Customer;
```

The values of the results can be modified through the use of SQL functions and expressions.

```
SELECT balance*100 AS cents FROM Account;
```

MULTI-RELATION SELECT QUERIES

So far we have only seen a query returning the results from a single table. Often we want the results from multiple tables, which have been joined together. The syntax for this is similar to simple SELECT queries:

```
SELECT attributes FROM relation1, relation2, ... WHERE condition
```

The main between the single relation and multi-relation query is the FROM clause contains a list of relations, and the conditions includes cross-relation (join) conditions.

```
SELECT *
      FROM Customer, Account
      WHERE customer.customer_id =  account.customer_id
```

Notice that each attribute is prefixed with the relation it belongs to. This defines where the attribute belongs and how to join the entities together. The relation.attribute convention doesn't help if we happen to use the same relation twice in a SELECT.

To handle this, we need to define new names for each "instance" of the relation in the FROM clause.

Notes:

```
SELECT *
      FROM Account a1, Account a2
      WHERE a1.customer_id = a2.customer_id
            AND a1.account_id != a2.account_id
```

This can also be used to shorten otherwise lengthy queries.

```
SELECT *
      FROM Customer c, Account a
      WHERE c.customer_id = a.customer_id
```

JOINS

Previously we looked at implicit joins, where two or more relations are joined through the conditions (WHERE clause). More complex joins require the query to contain an explicit join statement.

JOIN ON

This is the most general form of join. The ON clause is in the same boolean expression form as the WHERE conditions. All other joins assume identical names between the two joined relations (i.e. a.customer_id, c.customer_id). If the foreign key in the relation is named differently to the parent table you must use this join.

```
SELECT *
      FROM Customer c JOIN Account a ON c.customer_id=a.customer_id
      WHERE ...
```

OUTER JOIN

Joins only produces results where there are matching values in both of the relations involved in the join. Often, it is useful to produce results for all tuples in one or both relations, even if it has no matches in the other.

```
SELECT *
      FROM Account a LEFT OUTER JOIN Transactions t ON
(a.account_id=t.account_id)
      WHERE ...
```

Notes:

This will return a list of all accounts with transactions, as well as all new accounts which do not have any transactions. Those accounts without transactions will have NULL values in the transaction fields in the results.

Outer joins may use the ON, USING or NATURAL syntax.

[LEFT | RIGHT | FULL] OUTER JOIN

The table which returns NULL values is by default the first (or leftmost) table in the SQL query. This can be overridden by specifying RIGHT or FULL.

```
SELECT *
      FROM Customer c RIGHT OUTER JOIN Branch b
      WHERE ...
```

This will return a list of all customers with branches, as well as all branches that do not have any customers. Likewise;

```
SELECT *
      FROM Customer c FULL OUTER JOIN Branch b
      WHERE ...
```

This will return a list of all customers with branches, all branches who do not have any customers, and all customers who do not have any branches.

SUBQUERIES

The result of a SELECT query can be used in the WHERE clause of another query. In the simplest case the subquery returns a single result. The query can then treat the result as a single constant value and use in expressions.

```
SELECT *
      FROM Account
      WHERE balance = (SELECT max(balance) FROM Account);
```

It is also possible to return multiple results from your subquery. This can be treated as a list of values in the main query using the IN function.

Notes:

```
SELECT *
      FROM Account
      WHERE customer_id IN (SELECT customer_id
                            FROM Customer
                            WHERE gender='F');
```

UNION, INTERSECTION, DIFFERENCE

SQL implements the standard set operations, union, intersection and difference on multiple queries.

```
SELECT ... UNION SELECT ...
```

A union will return the set of results from each query as one result. An intersection will return the set of results that exist only in both queries. And except will return the set of results in the first query that do not exist in the second.

Each SELECT statement must return the same number of attributes, and each corresponding attribute must be of the same type.

```
SELECT min(date), transaction_amt
      FROM Transactions GROUP BY transaction_amt
UNION
SELECT max(date), transaction_amt
      FROM Transactions GROUP BY transaction_amt

SELECT date, transaction_amt
      FROM Transactions
EXCEPT
SELECT max(date), transaction_amt
      FROM Transactions GROUP BY transaction_amt
```

ORDER BY

SQL does not guarantee that the results of a given query will be in any particular order. The only way to guarantee this is to re-order the results in the order by statement. The order by statement is a comma separated list of attributes from the SELECT statement.

Notes:

```
SELECT *
      FROM Customer
      ORDER BY family_name, given_name ASC;
```

By default it will sort in ascending order, but this can be overridden with the keywords DESC (descending) or ASC (ascending).

GROUP BY

The GROUP BY statement partitions the results into groups according to a given list of attributes. This is most commonly used to treat each group separately in computing aggregations such as COUNT() or MAX().

```
SELECT count(*), gender
      FROM Customer
      GROUP BY gender;
```

If the results include aggregate, and non-aggregate values, every non-aggregate value must appear in the GROUP-BY clause.

DATA MODIFICATION IN SQL

SQL provides mechanisms for modifying data within tables. Constraint checking is applied automatically on any change. Unlike selection, you cannot perform these operations across multiple tables.

- INSERT: Add a new row into a table.
- DELETE: Remove rows from a table based on a given condition.
- UPDATE: Modify values in exiting tuples based on a given condition.

SQL also provides mechanisms for modifying table meta-data:

- CREATE TABLE: Create a new empty table
- ALTER TABLE: Change properties of existing table
- DROP TABLE: Remove table from database

Similar operations are available on other kinds of database objects.

Notes:

- CREATE VIEW, FUNCTION, RULE, ...
- DROP VIEW, FUNCTION, RULE, ...

Other objects do not have any UPDATE capability, for these use DROP and then CREATE. However, this may lose custom permissions on the object.

DATA INSERTION

Accomplished via the INSERT operation.

```
INSERT INTO Customer VALUES ('1', 'Cruz', 'Tobias' ,...);
```

The values to insert must be supplied for all attributes of the table in same order as appear in the CREATE TABLE statement. To supply the details in a different order, or to skip attributes you can specify which fields to insert into.

```
INSERT INTO Customer (family_name, given_name, gender) VALUES
('Cruz', 'Tobias', 'M');
```

Unspecified attributes are assigned NULL or, if applicable, a default value.

INSERTION FROM QUERIES

It is possible to use the result of a query to insert data. This can also allow you to insert multiple rows in a single operation.

```
INSERT INTO AccountTmp SELECT * FROM Account;
```

DATA DELETION

Accomplished via the DELETE operation. This will remove all rows from the table that satisfy a given condition.

```
DELETE FROM Customer WHERE family_name='Snoad';
```

If you do not specify a where clause it will delete all rows from the table, use with care.

```
DELETE FROM Customer;
```

Notes:

UPDATES

An update allows you to modify the values in a row or rows in a table that satisfy a given condition.

```
UPDATE Account SET balance='1234.00' WHERE customer_id='12345';
```

Each row in the table that satisfies the condition has the assignments applied to it. So it is possible to update every row in a table by neglecting the conditions. Assignments may assign constant values to attributes.

```
UPDATE Account SET balance='1234.00' WHERE customer_id='12345';
```

Or use existing values in the tuple to compute new values,

```
UPDATE Account SET balance=balance+'10.00' WHERE customer_id='12345';
```

CHANGING TABLES

Once a table has been created through the CREATE TABLE operation, modifications to the table structure can be made using the ALTER TABLE operation.

```
ALTER TABLE Customer ...
```

Some possible modifications are:

Add a new column or attribute. This will set all values to NULL unless a default is given.

```
ALTER TABLE Customer ADD salutation VARCHAR(10);
```

Change the type of a column.

```
ALTER TABLE Customer ALTER COLUMN salutation varchar(5);
```

Change properties of an existing attribute such as add new constraints.

```
ALTER TABLE Customer ALTER COLUMN salutation varchar(5) NOT NULL;
ALTER TABLE Customer ADD CONSTRAINT cons_name DEFAULT 'Mr' FOR
salutation ;
```

Notes:

Remove a column

```
ALTER TABLE Customer DROP COLUMN salutation;
```

INDEXES

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

```
CREATE INDEX index_name ON Customer (family_name);
CREATE INDEX index_name2 ON Customer (family_name, given_name);
```

Indexes can also be used to enforce uniqueness on a column. Note that Primary Keys will automatically be given a unique index.

```
CREATE UNIQUE INDEX index_name3 ON Branch (manager);
```

You will need to test the indexes improve performance and do not add unnecessary overhead.

VIEWS

A view is like a "virtual relation" or “virtual table” defined through a query. Each attribute selected (or calculated) in the query is an attribute in the view. The Query may be any SQL query.

```
CREATE VIEW FemaleCustomers AS SELECT * from Customer
      WHERE gender='F';
SELECT * FROM FemaleCustomers;
DROP VIEW FemaleCustomers;
```

A view is valid only as long as its underlying query is valid and does not have to use all the attributes of the base relations. A view can use computed attribute values defined during the query.

Notes:

Views can be used in queries as if they were stored relations. However, they differ from stored relations in two important respects:

1. Their "value" can change without being explicitly modified, i.e. a view may change whenever one of its base tables is updated.
2. They may not be able to be explicitly modified or updated, only a certain simple kinds of views can be explicitly updated.

Once again, you need to test the views for performance and to ensure the underlying query is valid.

Notes:

5: DOMAINS, STORED PROCEDURES AND TRIGGERS

'Fall seven times. Stand up eight.'

Old Japanese Proverb

Notes:

PROGRAMMING WITH SQL

SQL is a powerful language for manipulating relational data. But it is not a powerful programming language. At some point in developing and testing complete database applications

- We need to implement user interactions
- We need to control sequences of database operations
- We need to process query results in complex ways

and SQL cannot do any of these. Database programming requires a combination of

- manipulation of data in DB (via SQL)
- conventional programming (via procedural code)

This combination is realised in a number of ways:

- passing SQL commands via a "call-level" interface (programming language is decoupled from SQLServer; most flexible; e.g. Java/JDBC, PHP)
- embedding SQL into augmented programming languages (requires pre-processor for language; typically DBMS-specific; e.g. SQL/C)
- special-purpose programming language in SQLServer (closely integrated with DBMS) called T-SQL.

FUNCTIONS AND STORED PROCEDURES

Stored Procedures in SQLServer allow the developer to write small logical programs to interact with the user and the data. Stored Procedures are often written to check input and validate data, aggregate and extract data for reporting, and develop control functions external to the top level application. Stored Procedures can be called manually or can be called when an event occurs through triggers (itself a special kind of Stored Procedure).

Functions (or a User-Defined-Function) in SQLServer allow the developer to encapsulate complex query logic into a structure that can be called like any other systems function, such as count() or sin(). However functions can introduce significant performance issues if poorly designed. Finally, functions, unlike stored procedures, cannot change the database (e.g. you can't use DELETE, UPDATE, INSERT).

Notes:

T-SQL

T-SQL is a SQLServer specific language, extending the SQL standard, and integrating features of a procedural programming language. Functions and Stored Procedures are stored in the database with the data, which provides a means for extending SQLServer functionality. Common Stored Procedures include implementing constraint checking, triggered functions, complex query evaluation and detailed control of displayed results.

FUNCTION SYNTAX

```
CREATE PROCEDURE name @variable INT OUTPUT
AS
      ...
GO
```

VARIABLE ASSIGNMENT

```
DECLARE @var type;
SET @var = expression;
```

The expression may be an SQL query or a simple value.

VARIABLE TYPES

T-SQL constants and variables can be defined using:

- Standard SQL data types (CHAR, DATE, NUMBER, ...)
- User-defined data types (e.g. Point)
- Table types (e.g. Branches)

RETURN TYPES

An SQLServer function can return a value which is:

- An atomic data type such as an integer, float, text etc (e.g. create function factorial(int) returns int ...)
- A result set (e.g. create function EmployeeOfMonth(date) returns Employee ...)
- A cursor to a result set that can be accessed outside the stored procedure.

Notes:

SYNTAX

IF STATEMENT

```
IF cond_1
      statements_1 (or statement block)
ELSE IF cond_2
      statements_2 (or statement block)
ELSE
      statements_n (or statement block)
```

LOOPS

```
WHILE condition
      statement (or statement block)
```

Example: T-SQL Function

```
CREATE PROCEDURE withdraw @acctNum BIGINT, @amount NUMERIC, @r
VARCHAR(20) OUT
AS
      DECLARE @current NUMERIC;
      DECLARE @newbalance NUMERIC;

      SELECT @current=balance FROM Account
            WHERE account_id = @acctNum;

      IF @amount > @current
            SET @r = 'Insufficient Funds';
      ELSE
            BEGIN
                  SET @newbalance = @current - @amount;
                  UPDATE Account SET balance = @newbalance
                        WHERE account_id = @acctNum;
                  SET @r = 'New Balance: ' + cast(@newbalance as
varchar);
            END
GO
DECLARE @r varchar(20);
EXEC withdraw 1, 10, @r output;
SELECT @r
```

Notes:

INVOKING A FUNCTION OR STORED PROCEDURE

T-SQL stored procedures can be invoked in several ways:

- As part of the execution of another T-SQL function

```
EXEC myVoidFunction arg1,arg2;
```

- Automatically, via an insert/delete/update trigger
- Functions (but not Stored Procedures) may also be called as part of a SELECT statement

```
select myFunction(arg1,arg2);
select * from myTableFunction(arg1,arg2);
```

EXCEPTION HANDLING

Later versions of SQLServer allow the use of TRY/CATCH blocks to handle errors. When an exception occurs:

1. Control is transferred to the relevant exception handling code
2. All database changes so far in this transaction are undone
3. Handler executes and then transaction aborts (and function exits)

```
BEGIN TRY
      Statements...
END TRY
BEGIN CATCH
      Statements for failure...
END CATCH
```

You can output messages via the RAISERROR operator. These messages generate server log entries of different severities. For example:

```
RAISERROR 'error message', severity (0-25 higher is worse), state (to
differentiate between different versions of the same error) WITH log
```

Notes:

DYNAMICALLY GENERATED QUERIES

Strings can be built within a function and executed as a query. This is often useful when creating audit records in triggers. EXEC takes a string and executes it as an SQL query. This mechanism allows us to construct queries "on the fly". Note the multiple quote marks.

```
DECLARE @sql VARCHAR(100)
SET @sql = 'SELECT * FROM Account
      WHERE account_id=''' + cast(@id as varchar) + ''''
EXEC (@sql);
```

EXEC string can be used in any context where the query string could have been used.

TRIGGERS

Triggers are procedures that are stored in the database and are activated in response to database events. An database event, insert, update or delete activates the trigger which executes a stored procedure or statements.

Examples of uses for triggers:

- Checking schema-level constraints on update
- Maintaining summary data
- Building audit logs
- Performing multi-table updates (to maintain assertions)

Actions can be executed instead of or after the triggering event. Actions executed instead of the event, can modify or even skip the event. Actions executed after cannot, although they can rollback the event as the action exists within the same transaction.

Syntax for SQLServer trigger definition:

```
CREATE TRIGGER TriggerName ON TableName
      {AFTER|BEFORE} Event1 [OR Event2 ...]
AS statements;
```

Notes:

WORKED EXAMPLES

Notes:

CREATE_BANK.SQL

```
----------
-- DROP TABLES
--
-- Drop any existing tables. This is used to ensure the core tables
-- in the demo database are reset.
----------
DROP TABLE Transactions;
DROP TABLE Accounts;
DROP TABLE Customers;
DROP TABLE WorksAt;
DROP TABLE Branches;
DROP TABLE Staff;

----------
-- CREATE TABLES
--
-- Create the demo tables. Note that these tables have deliberate
-- logical errors
----------
CREATE TABLE Staff (
      staff_id INT IDENTITY(1,1) PRIMARY KEY,
      given_name NVARCHAR(50),
      family_name NVARCHAR(50),
      role NVARCHAR(50)
)

CREATE TABLE Branches (
      name VARCHAR(50),
      location VARCHAR(50),
      manager INT REFERENCES Staff
            ON DELETE SET NULL
            ON UPDATE CASCADE,
      CONSTRAINT branchpk PRIMARY KEY (name)
)

CREATE TABLE WorksAt (
      branch_id VARCHAR(50) REFERENCES Branches
```

Notes:

```
                ON UPDATE CASCADE
                ON DELETE CASCADE,
          staff_id INT REFERENCES Staff
                ON UPDATE NO ACTION
                ON DELETE NO ACTION,
          CONSTRAINT worksatpk PRIMARY KEY (branch_id, staff_id)
    )

    CREATE TABLE Customers (
          customer_id INT IDENTITY(1,1),
          given_name VARCHAR(50),
          family_name VARCHAR(50),
          gender CHAR(1),
          dob DATE,
          age INT,
          branch_id VARCHAR(50) REFERENCES Branches
                ON UPDATE CASCADE
                ON DELETE CASCADE,
          CONSTRAINT customerpk PRIMARY KEY (customer_id)
    )

    CREATE TABLE Accounts (
          account_id BIGINT IDENTITY(1,1) PRIMARY KEY,
          customer_id INT REFERENCES Customers
                ON UPDATE NO ACTION
                ON DELETE NO ACTION,
          balance NUMERIC CHECK (balance > 0),
          opened_by INT REFERENCES Staff
                ON UPDATE NO ACTION
                ON DELETE NO ACTION
    )

    CREATE TABLE Transactions (
          account_id BIGINT REFERENCES Accounts
                ON UPDATE NO ACTION
                ON DELETE NO ACTION,
          transaction_amt NUMERIC,
          date DATETIME
    )
```

Notes:

```
----------
-- LOAD DATA
--
-- Populate the database with demo data.
----------
INSERT INTO Staff VALUES ('Isaac', 'Asmiov', 'Manager');
INSERT INTO Staff VALUES ('Ray', 'Bradbury', 'Teller');
INSERT INTO Staff VALUES ('Arthur', 'Clarke', 'Teller');
INSERT INTO Staff VALUES ('Samuel', 'Delany', 'Customer Service');
INSERT INTO Staff VALUES ('Greg', 'Egan', 'Teller');

INSERT INTO Branches VALUES ('Canberra City', 'Canberra', '1');
INSERT INTO Branches VALUES ('St Lucia', 'Brisbane', '2');
INSERT INTO Branches VALUES ('Haymarket', 'Sydney', null);
INSERT INTO Branches VALUES ('Coburg', 'Melbourne', null);

INSERT INTO WorksAt VALUES ('Canberra City', '1');
INSERT INTO WorksAt VALUES ('St Lucia', '2');
INSERT INTO WorksAt VALUES ('Haymarket', '3');
INSERT INTO WorksAt VALUES ('Coburg', '2');

INSERT INTO Customers VALUES ('Esther', 'Friesner', 'F', '01-01-
1980', NULL, 'Canberra City');
INSERT INTO Customers VALUES ('William', 'Gibson', 'M', '01-01-1970',
NULL, 'St Lucia');
INSERT INTO Customers VALUES ('Peter', 'Hamilton', 'M', '01-01-1969',
NULL, 'St Lucia' );
INSERT INTO Customers VALUES ('Simon', 'Ings', 'M', '01-01-1971',
NULL, NULL);
INSERT INTO Customers VALUES ('Gweneth', 'Jones', 'F', '01-01-1980',
NULL, NULL);
INSERT INTO Customers VALUES ('Forest', 'Temmer', 'M', '01-01-1990',
NULL, NULL);

INSERT INTO Accounts VALUES ('1', 0, '1');
INSERT INTO Accounts VALUES ('1', 0, '1');
INSERT INTO Accounts VALUES ('2', 0, '1');
INSERT INTO Accounts VALUES ('3', 0, '2');
```

Notes:

```
INSERT INTO Accounts VALUES ('4', 0, '2');
INSERT INTO Accounts VALUES ('5', 0, '3');

INSERT INTO Transactions VALUES ('1', '100.00', getdate());
INSERT INTO Transactions VALUES ('1', '200.00', getdate());
INSERT INTO Transactions VALUES ('1', '-50.00', getdate());
INSERT INTO Transactions VALUES ('2', '100.00', getdate());
INSERT INTO Transactions VALUES ('2', '200.00', getdate());
INSERT INTO Transactions VALUES ('2', '200.00', getdate());
INSERT INTO Transactions VALUES ('2', '-500.00', getdate());
INSERT INTO Transactions VALUES ('2', '100.00', getdate());
INSERT INTO Transactions VALUES ('3', '100.00', getdate());
INSERT INTO Transactions VALUES ('3', '200.00', getdate());
INSERT INTO Transactions VALUES ('3', '200.00', getdate());
INSERT INTO Transactions VALUES ('3', '150.00', getdate());
INSERT INTO Transactions VALUES ('3', '-50.00', getdate());
INSERT INTO Transactions VALUES ('3', '-50.00', getdate());
INSERT INTO Transactions VALUES ('4', '100.00', getdate());
INSERT INTO Transactions VALUES ('4', '200.00', getdate());
INSERT INTO Transactions VALUES ('4', '200.00', getdate());
INSERT INTO Transactions VALUES ('4', '-250.00', getdate());
INSERT INTO Transactions VALUES ('4', '-50.00', getdate());
INSERT INTO Transactions VALUES ('5', '100.00', getdate());
INSERT INTO Transactions VALUES ('5', '200.00', getdate());
INSERT INTO Transactions VALUES ('5', '200.00', getdate());
INSERT INTO Transactions VALUES ('5', '250.00', getdate());
INSERT INTO Transactions VALUES ('5', '250.00', getdate());
INSERT INTO Transactions VALUES ('5', '200.00', getdate());
INSERT INTO Transactions VALUES ('1', '-2000.00', getdate());
```

Notes:

QUERIES – CREATING A DATABASE

```
----------
-- INITIAL STEPS
--
-- 1: Login to SSMS
-- 2: Create corebank database
-- 3: Open "New Query" window
-- 4: Run createbank.sql
-- 5: <optional> Attach AdventureWorks2012 database (ensure
--    persmissions are correct and log file can be created)
----------

----------
-- NAVIGATE SSMS
--
-- 1: Create and view ER Diagram
-- 2: Navigate tables heirarchy
-- 3: Investigate table and column options
-- 4: Investigate table and column constraints (primary keys, foreign
--    keys, check constraints)
-- 5: Navigate views heirarchy
----------

----------
-- INFORMATION SCHEMA
--
-- Explore the common information_schama views
----------
SELECT * FROM INFORMATION_SCHEMA.TABLES
SELECT * FROM INFORMATION_SCHEMA.COLUMNS
SELECT * FROM INFORMATION_SCHEMA.VIEWS
SELECT * FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
SELECT * FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS

-- Explore the uncommon information_schama views
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE
SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
```

Notes:

```
SELECT * FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
SELECT * FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES
SELECT * FROM INFORMATION_SCHEMA.VIEW_COLUMN_USAGE
SELECT * FROM INFORMATION_SCHEMA.VIEW_TABLE_USAGE

-- Explore the remaining information_schama views
SELECT * FROM INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE
SELECT * FROM INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS
SELECT * FROM INFORMATION_SCHEMA.DOMAINS
SELECT * FROM INFORMATION_SCHEMA.COLUMN_PRIVILEGES
SELECT * FROM INFORMATION_SCHEMA.PARAMETERS
SELECT * FROM INFORMATION_SCHEMA.ROUTINE_COLUMNS
SELECT * FROM INFORMATION_SCHEMA.ROUTINES
SELECT * FROM INFORMATION_SCHEMA.SCHEMATA

--Q1: Where would you find a list of all check constraints
SELECT * FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS
--Q2: Where would you find a list of all columns
SELECT * FROM INFORMATION_SCHEMA.COLUMNS
--Q3: Where would you find details of foreign keys
SELECT * FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS



----------
-- TRANSACTIONS
--
-- Show the syntax to wrap queries in transactions
----------
-- This will cause a deviation between the balance and the sum of all
-- transactions per account (check constraint on Acounts).
INSERT INTO Transactions VALUES (1, -1000000, getdate())
UPDATE Accounts SET balance = balance - 100000 WHERE account_id=1;
-- Proof
SELECT * FROM transactions WHERE account_id=1
DELETE FROM transactions WHERE transaction_amt=-1000000
-- Solution: wrap the queries in a transaction
BEGIN TRANSACTION
```

Notes:

```
      BEGIN TRY
            INSERT INTO Transactions VALUES (1, -1000000, getdate())
            UPDATE Accounts SET balance = balance - 100000 WHERE
account_id=1;
            COMMIT TRANSACTION
      END TRY
      BEGIN CATCH
            ROLLBACK TRANSACTION
      END CATCH
--Proof
SELECT * FROM transactions WHERE account_id=1
DELETE FROM transactions WHERE transaction_amt=1000000
-- And let's try it for a successful transaction
BEGIN TRANSACTION name
      BEGIN TRY
            INSERT INTO Transactions VALUES (1, 1000000, getdate())
            UPDATE Accounts SET balance = balance + 100000 WHERE
account_id=1;
            COMMIT TRANSACTION name
      END TRY
      BEGIN CATCH
            ROLLBACK TRANSACTION name
      END CATCH
--Proof
SELECT * FROM transactions WHERE account_id=1
```

Notes:

QUERIES – SQL

```
----------
-- SQL KEYWORDS
--
-- Show that SQL keywords are restricted
----------
CREATE TABLE select (
      id INT
)
CREATE TABLE foo (
      select INT
)
-- But if you really want to.
CREATE TABLE foo (
      "select" INT
)


----------
-- SQL DATATYPES
--
-- What would be a suitable type for?
-- Family name – nvarchar(50)
-- DOB - date
-- Age - int
-- Salary - numeric(8,2)
-- Timestamp - datetime
-- Gender field - char(1), boolean, varchar(20): depends on how you
--                define gender
-- Role Description – nvarchar(max) or ntext
-- Student ID eg s2006756 – char(8)
-- Transaction ID – bigint
-- Active Account – boolean
-- Last Login – datetime
-- File attachement – varbinary(max) or varchar(100) and store the
--                    file on the filesystem
----------
```

Notes:

```
----------
-- SQL OPERATORS: MATHS
--
-- Show the common mathematical operators
----------
SELECT 1+1
SELECT sin(180), cos(180), tan(180)
SELECT abs(-100)
SELECT CEILING(10.55), FLOOR(10.55), ROUND(10.55, 0), ROUND(10.55, 1)
SELECT power(2, 3), SELECT SQRT(9)
SELECT random()
-- Q1: How would you return a random whole number between 1 and 100
SELECT round(random()*100, 0)


----------
-- SQL OPERATORS: STRINGS
--
-- Show the common string operators
----------
SELECT 'Hello' + ' World'
SELECT len('Hello')
SELECT SUBSTRING('Hello', 4, 2)
SELECT lower('NSW'), upper('nsw')
SELECT CHARINDEX(' ', 'Hello World')
SELECT LEFT('Hello World', 5), RIGHT('Hello World', 5)
-- Q1: How would you strip the first word off a string
SELECT LEFT('Hello World', CHARINDEX(' ', 'Hello World'))


----------
-- SQL OPERATORS: DATES
--
-- Show the common date operators
----------
SELECT getdate()
SELECT datediff(yyyy, '01-01-1970', '01-01-1999')
```

Notes:

```
SELECT day(getdate()), month(getdate()), year(getdate()),
datepart(yyyy, getdate()), datepart(dy, getdate())
--Q1: How would you get someones age from their dob
SELECT datediff(yyyy, '01-01-1970', getdate())


----------
-- AGGREGATION
--
-- Show the aggregation (and group by) syntax
----------
SELECT count(*), sum(transaction_amt), avg(transaction_amt),
min(transaction_amt), max(transaction_amt) from transactions
SELECT sum(*), account_id
      FROM Accounts
      GROUP BY account_id


----------
-- NULL
--
-- Null will always return NULL (expect in aggregation)
----------
SELECT 1+NULL, 3/NULL
-- Proof aggregation should return the same value
SELECT sum(transaction_amt) FROM transactions
INSERT INTO transactions VALUES (1, null, getdate())
SELECT * FROM transactions
SELECT sum(transaction_amt) FROM transactions


----------
-- UNICODE
--
-- How to insert unicode strings into the database. This should be
-- the default unless there is a valid design reason not to.
----------
SELECT * FROM Staff
-- Unicode string
```

Notes:

```
INSERT INTO Staff VALUES (N'???', N'???', 'Teller')
-- Proof
SELECT * FROM Staff
-- Unicode string, but not encapsulated correctly
INSERT INTO Staff VALUES ('???', '???', 'Teller')
SELECT * FROM Staff


----------
-- SQL DEMONSTRATION
--
-- Demonstrate all the SQL functions that have been discussed to
-- date. Focus on JOINS.
----------
SELECT * FROM Customers
SELECT * FROM Customers WHERE gender IN ('M', 'F');
SELECT * FROM Customers c, Branches b
SELECT * FROM Customers , Accounts a
      WHERE c.customer_id = a.customer_id
SELECT * FROM Accounts a1, Accounts a2
      WHERE a1.customer_id = a2.customer_id


----------
-- SQL EXERCISES
--
-- Test all the SQL functions that have been discussed to date. Focus
-- on JOINS.
----------
--Q1. Return all branches
SELECT * FROM Branches
--Q2. List of all Female Customers
SELECT * FROM Customers WHERE gender='F'
--Q3. List of all Male Customers over 40
SELECT *
      FROM Customers
      WHERE gender='M' AND datediff(yy, dob, getdate()) > 40
--Q4. List of all Customers sorted by given name
SELECT * FROM Customers ORDER BY given_name
```

Notes:

```
--Q5. List of all Customers with a home branch
SELECT * FROM Customers
      WHERE branch_id is not null ---- or
SELECT * FROM Customers c, Branches b
      WHERE c.branch_id = b.name
--Q6. List of all Customers with their home branch (or not)
SELECT *
      FROM Customers c LEFT OUTER JOIN Branches b ON c.branch_id =
b.name
--Q7. Average Transaction size
SELECT avg(transaction_amt) FROM Transactions
--Q8. List of all Customers without a Branch
SELECT * FROM Customers WHERE branch_id is null



----------
-- MORE SQL EXERCISES
--
-- It never ends
----------
--Q1. A List of unique roles in the system
SELECT DISTINCT role from Staff
--Q2. List of all people (Customers & Staff) in the system
SELECT customer_id, family_name, given_name FROM Customers
      UNION
      SELECT staff_id, family_name, given_name FROM Staff
--Q2(ADVANCED): Include the type of person
SELECT customer_id, family_name, given_name, 'Customer' FROM
Customers
      UNION
      SELECT staff_id, family_name, given_name, 'Staff' FROM Staff
--Q3. The list of all Customers without Accounts
SELECT *
      FROM Customers
      WHERE customer_id NOT IN (SELECT customer_id FROM Accounts)
--Q4. The correct balance per account
SELECT sum(transaction_amt), account_id
      FROM Transactions
      GROUP BY account_id
```

Notes:

```
--Q5. The Customer with the highest balance (by transaction)
SELECT TOP 1 sum(t.transaction_amt) as sum, c.customer_id
      FROM Transactions t, Accounts a, Customers c
      WHERE t.account_id=a.account_id AND c.customer_id=a.customer_id
      GROUP BY c.customer_id
      ORDER BY sum desc
--Q6. Number of Staff per location
SELECT count(*), branch_id FROM worksat GROUP BY branch_id
--Q7. List of all Customers whose given name contains ‘e’
SELECT * FROM Customers WHERE given_name LIKE '%e%'
--Q8. List of all Customers whose name 2nd character is ‘e’
SELECT * FROM Customers WHERE given_name LIKE '_e%'
--Q9. List of all Customers whose name does start with ‘e’
SELECT * FROM Customers WHERE given_name LIKE '[^e]%'
--Q10. List of Customers with their highest balance Account and 
lowest balance Account
SELECT t.customer_id, min(t.sumt) as min, max(t.sumt) as max FROM
      (
      SELECT c.customer_id, a.account_id, sum(t.transaction_amt) as 
sumt
            FROM Customers c, Accounts a, Transactions t
            WHERE c.customer_id=a.customer_id AND 
a.account_id=t.account_id
            GROUP BY c.customer_id, a.account_id
      ) as t
            GROUP BY customer_id


----------
-- QUERY EXECUTION PLAN
--
-- Show QEP
-- Show how to read plan (backwards)
-- Explain how Cost % works
-- Show the difference between table scan and index scan
----------


----------
```

Notes:

```
-- BROKEN SQL
--
-- Identify the errors in the following statements
----------
--Q1: Missing GROUP BY
SELECT account_id, avg(transaction_amt) FROM Transactions
--Q2: IS NOT NULL not != NULL
SELECT * FROM Customers WHERE branch_id != null
--Q3: DISTINCT not UNIQUE
SELECT UNIQUE role from Staff
--Q4: ' not "
SELECT customer_id, family_name, given_name, "Customer" FROM
Customers
      UNION
      SELECT staff_id, family_name, given_name, "Staff" FROM Staff
--Q5: Incorrect comparison
SELECT * FROM Customers WHERE customer_id NOT IN (SELECT account_id
FROM Accounts)
--Q6: Case insensitivity
SELECT *
      FROM Customers
      WHERE gender='f'
--Q7: Missing JOIN fields
SELECT *
      FROM Customers c, Accounts a
      WHERE c.customer_id = 1
--Q8: Incorrect join fields
SELECT *
      FROM Customers c JOIN Transactions t ON
(t.account_id=c.customer_id)
--Q9: Different number of fields between the UNION
SELECT * FROM Customers
UNION
SELECT * FROM Staff
--Q10: Undifferentiated tables
SELECT *
      FROM Accounts, Accounts
      WHERE accounts.customer_id=accounts.customer_id
--Q11: yy not 'yy'
```

Notes:

```
SELECT * FROM Customers WHERE gender='M' AND datediff('yy', dob,
getdate()) > 40
--Q12: missing as t
SELECT t.customer_id, min(t.sumt) as min, max(t.sumt) as max FROM
      (
      SELECT c.customer_id, a.account_id, sum(t.transaction_amt) as
sumt
            FROM Customers c, Accounts a, Transactions t
            WHERE c.customer_id=a.customer_id AND
a.account_id=t.account_id
            GROUP BY c.customer_id, a.account_id
      )
            GROUP BY customer_id
--Q13: Incorrect join
SELECT t.customer_id, min(t.sumt) as min, max(t.sumt) as max FROM
      (
      SELECT c.customer_id, a.account_id, sum(t.transaction_amt) as
sumt
            FROM Customers c, Accounts a, Transactions t
            WHERE c.customer_id=a.customer_id AND
t.account_id=t.account_id
            GROUP BY c.customer_id, a.account_id
      ) as t
            GROUP BY customer_id
--Q14: sum() not min()
SELECT t.customer_id, min(t.sumt) as min, max(t.sumt) as max FROM
      (
      SELECT c.customer_id, a.account_id, min(t.transaction_amt) as
sumt
            FROM Customers c, Accounts a, Transactions t
            WHERE c.customer_id=a.customer_id AND
a.account_id=t.account_id
            GROUP BY c.customer_id, a.account_id
      ) as t
            GROUP BY customer_id
--Q15: Wrong database - Note: change the database to 'master'
SELECT * FROM Customers

----------
```

Notes:

```
-- TEST CASE STRUCTURE
--
-- Some options on how to wrap SQL in test cases.
----------
-- Type A: Returns result set
SELECT
      CASE
            WHEN <INCLUDE TEST CHECK <BASED ON TEST QUERY RESULTS>
HERE>  THEN 'Test Pass'
            WHEN <INCLUDE TEST CHECK <BASED ON TEST QUERY RESULTS>
HERE> THEN 'Test Fail'
      END
      FROM (
            <INCLUDE TEST QUERY HERE>
      ) as t

-- Type B: Returns Exception (RAISERROR)
IF <INCLUDE TEST CHECK HERE>(<INCLUDE TEST QUERY HERE>)
      RAISERROR ('Test Fail', 11, 1)
ELSE
      PRINT 'Test Pass'

-- Type C: Returns Exception with customer message
DECLARE @msg varchar(255)
DECLARE @result int
SET @result = 0
SET @msg = 'Test Fail: '

SET @result = (SELECT count(*) FROM (
            <INCLUDE TEST QUERY HERE>
      ) as t)

IF <INCLUDE TEST CHECK HERE>
      BEGIN
            SET @msg = @msg + cast(@result as varchar) + ' records
found. Expected 0'
            RAISERROR (@msg , 11, 100)
      END
ELSE
```

Notes:

```
      PRINT 'Test Pass'


----------
-- TEST CASE DEMO
--
-- Show how you can wrap simple SQL in a test case wrapper
----------
-- Setup (compare current DB schema to a previous snapshot)
DROP TABLE baseline_schema
SELECT * INTO baseline_schema FROM INFORMATION_SCHEMA.COLUMNS
-- Type A: Test Pass
SELECT
      CASE
            WHEN count(*) = 0 THEN 'Test Pass'
            WHEN count(*) > 0 THEN 'Test Fail'
      END
      FROM (
            SELECT * FROM INFORMATION_SCHEMA.COLUMNS
            EXCEPT
            SELECT * FROM baseline_schema
) as t

-- Type A: Test Fail
DELETE FROM baseline_schema WHERE COLUMN_NAME='name'
SELECT
      CASE
            WHEN count(*) = 0 THEN 'Test Pass'
            WHEN count(*) > 0 THEN 'Test Fail: ' + cast(count(*) as
varchar) + ' records found. Expected 0'
      END
      FROM (
            SELECT * FROM INFORMATION_SCHEMA.COLUMNS
            EXCEPT
            SELECT * FROM baseline_schema
) as t

-- Type B: Test Fail
IF EXISTS(
```

Notes:

```
		SELECT * FROM INFORMATION_SCHEMA.COLUMNS
		EXCEPT
		SELECT * FROM baseline_schema
		)
		RAISERROR ('Test Fail', 11, 1)
	ELSE
		PRINT 'Test Pass'


	----------
	-- TEST CASE EXAMPLES
	--
	-- Practices wrapping SQL in test cases
	----------
	--BR1: All branches must have a manager
	SELECT
		CASE
			WHEN count(*) = 0 THEN 'Test Pass'
			WHEN count(*) > 0 THEN 'Test Fail: ' + cast(count(*) as 
	varchar) + ' records found. Expected 0'
		END
		FROM (
			SELECT * FROM Branches WHERE manager is null
		) as t
	--or (show multiple rows - not prefered)
	SELECT
		CASE
			WHEN count(*) = 0 THEN 'Test Pass'
			WHEN count(*) > 0 THEN 'Test Fail: ' + cast(count(*) as 
	varchar) + ' records found. Expected 0'
		END, location
		FROM (
			SELECT * FROM Branches WHERE manager is null
		) as t
		GROUP BY location
	--or (bad practice)
	SELECT 'Test Fail', location FROM Branches WHERE manager is null
	--or (test the data rather than the count - not prefered)
	SELECT
```

Notes:

```
      CASE
            WHEN location != 'Sydney' THEN 'Test Pass'
            WHEN location = 'Sydney' THEN 'Test Fail: ' +
cast(count(*) as varchar) + ' records found. Expected 0'
      END, location
      FROM (
            SELECT * FROM Branches WHERE manager is null
      ) as t
      GROUP BY location
--BR2: The account balance must = the sum of all transactions
IF EXISTS(
      SELECT sum(t.transaction_amt)-a.balance
            FROM Transactions t, Accounts a
            WHERE t.account_id=a.account_id
            GROUP BY a.balance
            HAVING sum(t.transaction_amt)-a.balance > 0
      )
      RAISERROR ('Test Fail', 11, 1)
ELSE
      PRINT 'Test Pass'
--BR3: The manager must work at a branch
DECLARE @msg varchar(255)
DECLARE @result int
SET @result = 0
SET @msg = 'Test Fail: '
SET @result = (SELECT count(*) FROM (
      SELECT manager
            FROM Branches b
            WHERE b.manager NOT IN (SELECT staff_id FROM WorksAt
WHERE WorksAt.branch_id=b.name)
      ) as t)
IF @result > 0
      BEGIN
            SET @msg = @msg + cast(@result as varchar) + ' records
found. Expected 0'
            RAISERROR (@msg , 11, 100)
      END
ELSE
      PRINT 'Test Pass'
```

Notes:

```
	--BR4: The ages of each customer must be correct
	SELECT
		CASE
			WHEN sumage = 0 THEN 'Test Pass'
			WHEN sumage > 0 THEN 'Test Fail'
		END
		FROM (
			SELECT SUM(datediff(yy, dob, getdate()) - coalesce(age, 
	0)) as sumage
				FROM Customers
		) as t
	--BR5: All customers must have +’ve balances
	IF EXISTS(
		SELECT balance FROM Accounts WHERE balance < 0
		)
		RAISERROR ('Test Fail', 11, 1)
	ELSE
		PRINT 'Test Pass'
	--or (via insert)
	DECLARE @myrow INT
	SET @myrow = 0
	BEGIN TRANSACTION t1
		BEGIN TRY
			INSERT INTO Accounts VALUES (1, -100, 1)
			COMMIT TRANSACTION t1
		END TRY
		BEGIN CATCH
			ROLLBACK TRANSACTION t1
		END CATCH
	SELECT
		CASE
			WHEN rownum = 0 THEN 'Test Pass'
			WHEN rownum > 0 THEN 'Test Fail: Negative balance was 
	inserted'
		END
		FROM (
			SELECT @myrow as rownum
		) as t
	--BR6: Managers must have the correct staff role
```

Notes:

```
DECLARE @msg varchar(255)
DECLARE @result int
SET @result = 0
SET @msg = 'Test Fail: '
SET @result = (SELECT count(*) FROM (
      SELECT *
            FROM staff s, branches b
            WHERE s.staff_id=b.manager AND s.role != 'Manager'
      ) as t)
IF @result > 0
      BEGIN
            SET @msg = @msg + cast(@result as varchar) + ' records
found. Expected 0'
            RAISERROR (@msg , 11, 100)
      END
ELSE
      PRINT 'Test Pass'
--BR7: All Customers must have an account
IF EXISTS(
      SELECT customer_id FROM Customers c
      EXCEPT
      SELECT customer_id FROM Accounts a
      )
      RAISERROR ('Test Fail', 11, 1)
ELSE
      PRINT 'Test Pass'
--BR8: All Staff must work at a Branch
DECLARE @msg varchar(255)
DECLARE @result int
SET @result = 0
SET @msg = 'Test Fail: '
SET @result = (SELECT count(*) FROM (
            SELECT staff_id FROM Staff
            EXCEPT
            SELECT staff_id FROM WorksAt
      ) as t)
IF @result > 0
      BEGIN
```

Notes:

```
		SET @msg = @msg + cast(@result as varchar) + ' records
found. Expected 0'
		RAISERROR (@msg , 11, 100)
	END
ELSE
	PRINT 'Test Pass'
--BR9: All Accounts must be opened by Staff
DECLARE @myrow INT
SET @myrow = 0
BEGIN TRANSACTION t1
	BEGIN TRY
		INSERT INTO Accounts VALUES (1, 100, NULL)
		COMMIT TRANSACTION t1
	END TRY
	BEGIN CATCH
		ROLLBACK TRANSACTION t1
	END CATCH
SELECT
	CASE
		WHEN rownum > 0 THEN 'Test Pass'
		WHEN rownum = 0 THEN 'Test Fail: Account was inserted
without a staff member'
	END
	FROM (
		SELECT @myrow as rownum
	) as t


----------
-- ROW_NUMBER/RANK
--
-- Show how to return row numbers in the result set
----------
SELECT ROW_NUMBER() OVER (ORDER BY account_id), * FROM Transactions
SELECT ROW_NUMBER() OVER (ORDER BY transaction_amt), * FROM
Transactions
SELECT ROW_NUMBER() OVER (PARTITION BY account_id ORDER BY
transaction_amt), * FROM Transactions
SELECT RANK() OVER (ORDER BY account_id), * FROM Transactions
```

Notes:

```
SELECT RANK() OVER (ORDER BY transaction_amt), * FROM Transactions
SELECT RANK() OVER (PARTITION BY account_id ORDER BY
transaction_amt), * FROM Transactions

----------
-- SQL UPDATES, INSERTS & DELETE
--
-- Practice changing data in the database
----------
--Q1. Update Balance (Account)
INSERT INTO Transactions VALUES (1, 2000, getdate())
UPDATE Accounts
      SET balance=(SELECT sum(t.transaction_amt) FROM transactions t
WHERE Accounts.account_id=t.account_id)
SELECT * FROM accounts;
--Q2. Add 5 new Staff manually and 5 using SSMS
INSERT INTO Staff VALUES ('Tobias', 'Snoad', 'Teller')
INSERT INTO Staff VALUES ('Jennifer', 'Edmonson', 'Teller')
INSERT INTO Staff VALUES ('Ashley', 'Flynn', 'Teller')
INSERT INTO Staff VALUES ('Dev', 'Pathy', 'Teller')
INSERT INTO Staff VALUES ('Xianzheng', 'Zhou', 'Teller')
--Q3. Add a Manager for every Branch
UPDATE Branches SET manager=3 WHERE name=2;
UPDATE Branches SET manager=4 WHERE name=3;
--Q4. Update Age (Customer)
UPDATE Customers SET age=datediff(year, dob, getdate())
--Q5. Delete all Customers without an Account
DELETE FROM Customers
      WHERE customer_id NOT IN (SELECT customer_id FROM Accounts)
--Q6. Deposit $100 into Esthers Account
SELECT * FROM Customers c, Accounts a
      WHERE a.customer_id=c.customer_id AND c.customer_id=1
INSERT INTO Transactions VALUES (1, 100, getdate());
UPDATE Accounts SET balance=balance+100 WHERE account_id=1
--or (via transaction)
BEGIN TRANSACTION t1
      BEGIN TRY
            INSERT INTO Transactions
                  VALUES (
```

Notes:

```
                        (SELECT TOP 1 a.account_id
                              FROM Customers c, Accounts a
                              WHERE a.customer_id=c.customer_id AND
c.given_name='Esther'),
                        100,
                        getdate());

            UPDATE Accounts SET balance=balance+100
                  WHERE account_id=(
                        SELECT TOP 1 a.account_id
                              FROM Customers c, Accounts a
                              WHERE a.customer_id=c.customer_id AND
c.given_name='Esther')
            COMMIT TRANSACTION t1
      END TRY
      BEGIN CATCH
            ROLLBACK TRANSACTION t1
      END CATCH
--Q7. Convert all Customer given & family names to lower case
UPDATE Customers SET family_name=lower(family_name)
UPDATE Customers SET given_name=lower(given_name)
--Q8. Delete Customer #2
DELETE
      FROM Transactions
      WHERE account_id=(SELECT account_id FROM accounts where
customer_id=2)
DELETE FROM Accounts WHERE customer_id=2
DELETE FROM Customers WHERE customer_id=2



----------
-- CREATING CUSTOMER DATA TYPES
--
-- For common constraints such as postcode or email address
----------
CREATE TYPE postcode FROM CHAR(4) NOT NULL;
CREATE TABLE foo (
      user_pc POSTCODE,
      username VARCHAR(20)
```

Notes:

```
)
INSERT INTO foo VALUES ('1234', 'evan')
SELECT * FROM foo
INSERT INTO foo VALUES ('12345', 'evan')
SELECT * FROM foo
INSERT INTO foo VALUES ('x123', 'evan')
SELECT * FROM foo
DROP TABLE foo
DROP TYPE POSTCODE
-- or
CREATE TYPE postcode FROM INT
      CHECK (postcode > 10000 and postcode < 99999);
CREATE TABLE foo (
      user_pc POSTCODE,
      username VARCHAR(20)
)
INSERT INTO foo VALUES (11111, 'evan')
SELECT * FROM foo
INSERT INTO foo VALUES (123456, 'evan')
SELECT * FROM foo
INSERT INTO foo VALUES (100, 'evan')
SELECT * FROM foo


----------
-- DDL
--
-- Changing the database
----------
-- ALTER TABLE
ALTER TABLE Customers ADD salutation VARCHAR(10);
ALTER TABLE Customers ALTER COLUMN salutation varchar(5);
ALTER TABLE Customers ALTER COLUMN salutation varchar(5) NOT NULL;
ALTER TABLE Customers
      ADD CONSTRAINT cons_name DEFAULT 'Mr' FOR salutation ;
ALTER TABLE Customers DROP COLUMN salutation;
-- ALTER TABLE: Computed Columns
ALTER TABLE Customers
      ADD fullname2 AS given_name + ' ' + family_name
```

Notes:

```
ALTER TABLE Customers
      ADD fullname AS given_name + ' ' + family_name PERSISTED
SELECT * FROM Customers;
-- CREATE TABLE
CREATE TABLE Address (
      addressid INT IDENTITY (1,1),
      street VARCHAR(50),
      suburb VARCHAR(50),
      city VARCHAR(50),
      country VARCHAR(50) DEFAULT 'Malaysia',
      postcode INT CHECK (postcode > 1000 AND postcode < 9999)
)
INSERT INTO Address VALUES (NULL, NULL, NULL, DEFAULT, 1001)
SELECT * FROM Address
-- DROP TABLE
DROP TABLE Customer
-- CREATE INDEX
CREATE INDEX index_name ON Customer (family_name);
CREATE INDEX index_name2 ON Customer (family_name, given_name);
-- CREATE VIEW
CREATE VIEW FemaleCustomers AS SELECT * from Customer
      WHERE gender='F';
SELECT * FROM FemaleCustomers;
DROP VIEW FemaleCustomers;
```

Notes:

STORED PROCEDURES

```
----------
-- T-SQL (AND STORED PROCEDURE) SYNTAX
--
-- Show that basic T-SQL syntax. While there is a lot more, this is
-- sufficient to write good test cases
----------
-- Variables
DECLARE @var type;
SET @var = expression;
-- IF/THEN/ELSE
IF cond_1
      statements_1 (or statement block)
ELSE IF cond_2
      statements_2 (or statement block)
ELSE
      statements_n (or statement block)
-- Loops
WHILE condition
      statement (or statement block)



----------
-- STORED PROCEDURE EXAMPLES
--
-- Withdraw is a simple (and inaccurate) stored procedure updates the
-- balance in the Accounts table based on the given parameters
----------
--A very simple stored procedure
CREATE PROCEDURE getCustomer(@customerId INT) AS
BEGIN
      SELECT * FROM Customers WHERE customer_id=@customerId
END
-- Call a stored procedure
EXEC getCustomer 1
-- Note: CREATE PROCEDURE Must be the first statement in the block
-- when executing
DROP PROCEDURE withdraw
```

Notes:

```
CREATE PROCEDURE withdraw @acctNum BIGINT, @amount NUMERIC AS
BEGIN
      DECLARE @current NUMERIC;
      DECLARE @newbalance NUMERIC;

      SELECT @current=balance FROM Accounts
            WHERE account_id = @acctNum;

      IF @amount > @current
            SELECT 'Insufficient Funds';
      ELSE
            BEGIN
                  SET @newbalance = @current - @amount;
                  UPDATE Accounts SET balance = @newbalance
                        WHERE account_id = @acctNum;
                  SELECT 'New Balance: ' + cast(@newbalance as
varchar);
            END
END
-- Call a stored procedure
EXEC withdraw 1, 10;


----------
-- STORED PROCEDURE TEST CASES
--
-- Exercises on how to wrap a stored procedure in a test case
----------
-- Prepare
CREATE TABLE #result (
      result VARCHAR(100)
)
--Q1. Withdraw SP returns “New Balance XXX” when withdrawing $1
--    from User 1
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, 1
SELECT
      CASE
            WHEN count(*) = 1
```

Notes:

```
                        THEN 'Test Pass'
                ELSE 'Test Fail'
        END
        FROM (
                SELECT * FROM #result WHERE result LIKE 'New Balance%'
        ) as t
--Q2. SP returns “Insufficient Balance” when withdrawing $10000
--    from User 1
SET NOCOUNT ON
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, 10000
IF (SELECT count(*) FROM #result WHERE result LIKE 'Insufficient
Funds')=0
        RAISERROR ('Test Fail', 11, 1)
ELSE
        PRINT 'Test Pass'
--Q3. SP returns an error when withdrawing -$10
SET NOCOUNT ON
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, -10
IF (SELECT count(*) FROM #result WHERE result LIKE 'Error%')=0
        RAISERROR ('Test Fail', 11, 1)
ELSE
        PRINT 'Test Pass'
--Q4. SP returns an error when withdrawing from User 100
DELETE FROM #result
INSERT INTO #result EXEC withdraw 100, 1
SELECT
        CASE
                WHEN count(*) > 0 THEN 'Test Pass'
                ELSE 'Test Fail: Expected 1 record, found ' +
cast(count(*) as varchar)
        END
        FROM (
                SELECT * FROM #result WHERE result LIKE 'Error%'
        ) as t
--Q5. SP returns “New Balance” when withdrawing FLOAT
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, 1.5
```

Notes:

```
DECLARE @msg varchar(255)
DECLARE @result varchar(255)
SET @result = 0
SET @msg = ''
SET @result = (SELECT * FROM #result WHERE result LIKE 'New
Balance%')
IF count(@result) = 0
      BEGIN
            SET @msg = 'Test Fail: ' + cast(@result as varchar) + ''
            RAISERROR (@msg , 11, 100)
      END
ELSE
      BEGIN
            SET @msg = 'Test Pass: ' + cast(@result as varchar) + ''
            PRINT @msg
      END
--Q6. SP returns Bal-1.5 when withdrawing 1.499999999
DECLARE @msg varchar(255)
DECLARE @result numeric(18, 2)
DECLARE @expected numeric(18,2)
SET @result = 0
SET @msg = ''
SET @expected = (SELECT cast(balance as numeric(18, 2))-1.5
                              FROM Accounts
                              WHERE account_id=1)
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, 1.4999999999
SET @result = (SELECT cast(SUBSTRING(result, 14, 100) as numeric(18,
2))
                              FROM #result )
IF @result != @expected
      BEGIN
            SET @msg = 'Test Fail: Expected ' + cast(@expected as
varchar) + ', Got ' + cast(@result as varchar) + ''
            RAISERROR (@msg , 11, 100)
      END
ELSE
      BEGIN
            SET @msg = 'Test Pass: ' + cast(@result as varchar) + ''
```

Notes:

```
            PRINT @msg
        END
    --Q7. SP returns error when withdrawing “ABC”
    SET NOCOUNT ON
    DELETE FROM #result
    BEGIN TRANSACTION
        BEGIN TRY
            INSERT INTO #result EXEC withdraw 1, 'ABC'
            RAISERROR ('Test Fail', 11, 1)
            COMMIT TRANSACTION
        END TRY
        BEGIN CATCH
            PRINT 'Test Pass'
            ROLLBACK TRANSACTION
        END CATCH
    --Q8. Account table balance=balance-1 when withdrawing $1
    DECLARE @old NUMERIC
    DECLARE @new NUMERIC
    SET @old = (SELECT balance FROM Accounts where account_id=1)
    EXEC withdraw 1, 1
    SET @new = (SELECT balance FROM Accounts where account_id=1)
    IF @new = @old-1
        PRINT 'Test Pass'
    ELSE
        RAISERROR ('Test Fail', 11, 1)
    --Q9. Account table balance=sum (transaction table) when ...
    --Q10. Account balance=balance when withdrawing $10000
    SELECT * INTO #accountstmp FROM accounts
    EXEC withdraw 1, 100000
    SELECT
        CASE
            WHEN count(*) = 0
                THEN 'Test Pass'
            ELSE 'Test Fail'
        END
        FROM (
            SELECT * FROM Accounts
            EXCEPT
            SELECT * FROM #accountstmp
```

Notes:

```
      ) as t
DROP TABLE #accountstmp
--or
DECLARE @old NUMERIC
DECLARE @new NUMERIC
SET @old = (SELECT balance FROM Accounts where account_id=1)
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, 100000
SET @new = (SELECT balance FROM Accounts where account_id=1)
IF @new = @old
      PRINT 'Test Pass'
ELSE
      RAISERROR ('Test Fail', 11, 1)
--Q11. Account table should not change when withdrawing $-10
SELECT * INTO #accountstmp FROM accounts
DELETE FROM #result
INSERT INTO #result EXEC withdraw 1, -10
SELECT
      CASE
            WHEN count(*) = 0
                  THEN 'Test Pass'
            ELSE 'Test Fail'
      END
      FROM (
            SELECT * FROM Accounts
            EXCEPT
            SELECT * FROM #accountstmp
      ) as t
DROP TABLE #accountstmp


----------
-- TRIGGERS
--
-- How to automatically execute a stored procedure when an insert,
-- update or delete occurs. Note the "inserted" table.
----------
CREATE TRIGGER withdraw ON Transactions AFTER INSERT AS
BEGIN
```

Notes:

```
      DECLARE @current NUMERIC;
      DECLARE @newbalance NUMERIC;
      DECLARE @amount NUMERIC;
      DECLARE @acctNum BIGINT;
      SELECT @current=balance, @amount=i.transaction_amt,
@acctNum=i.account_id
            FROM Accounts a, inserted i
            WHERE a.account_id = i.account_id;
      IF @amount + @current < 0
            BEGIN
                  RAISERROR('Insufficient funds', 1, 1)
                  ROLLBACK TRANSACTION
            END
      ELSE
            BEGIN
                  SET @newbalance = @current + @amount;
                  UPDATE Accounts SET balance = @newbalance
                        WHERE account_id = @acctNum;
            END
END
-- Proof
INSERT INTO Transactions VALUES (1, 100, getdate())
SELECT * FROM Accounts WHERE account_id=1
SELECT * FROM Transactions WHERE account_id=1
-- Proof
INSERT INTO Transactions VALUES (1, -500, getdate())
SELECT * FROM Accounts WHERE account_id=1
SELECT * FROM Transactions WHERE account_id=1
-- Simple ON DELETE TRIGGER to stop a deletion from occuring
CREATE TRIGGER stopDelete ON Transactions AFTER DELETE AS
BEGIN
      RAISERROR('Cannot Delete transactions.', 17, 1)
      ROLLBACK TRANSACTION
END
-- Proof
DELETE FROM Transactions


----------
```

Notes:

```
-- DYNAMIC SQL
--
-- How to automatically run an SQL query that is stored in a string
-- or table
----------
-- Simple execution
EXEC ('SELECT * FROM Customers WHERE customer_id=1' )
-- Execute through a look - select customers 1-5
DECLARE @num int
SET @num = 5
DECLARE @sql varchar(max)
SET @sql = ''
WHILE @num > 0
      BEGIN
            SET @sql = @sql + 'SELECT * FROM Customers
                  WHERE customer_id=' + cast(@num as varchar)+ ' '
            IF @num <> 1
                  SET @sql = @sql + 'UNION '
            SET @num = @num - 1
      END
EXEC (@sql)
-- Build SQL in SQL
SELECT
    'select ''' + Table_Schema + ''' as SchemaName,
    ''' + Table_Name + ''' as ObjectName ,
    ''' + case Table_Type when 'base table' then 'table' else
lower(Table_Type) end + ''' as ObjectType ,
    count(*) as Rows from ' + Table_Schema +'.'+ Table_Name + ' '
FROM information_schema.tables
ORDER BY Table_Type;
-- Execute a series of SQL from a SQL query result set (via a cursor)
-- This is very advanced.
DECLARE c CURSOR FOR select
    'select ''' + Table_Schema + ''' as SchemaName,
    ''' + Table_Name + ''' as ObjectName ,
    ''' + case Table_Type when 'base table' then 'table' else
lower(Table_Type) end + ''' as ObjectType ,
    count(*) as Rows from ' + Table_Schema +'.'+ Table_Name + ' '
      from information_schema.tables
```

Notes:

```
DECLARE @bigsql varchar(max)
SET @bigsql = ''
OPEN c
DECLARE @sql varchar(max)
FETCH NEXT FROM c INTO @sql
WHILE @@FETCH_STATUS = 0
      BEGIN
            SET @bigsql = @bigsql + @sql + ' union all '
            FETCH NEXT FROM c INTO @sql
      END
CLOSE c
DEALLOCATE c
EXEC(@bigsql + @sql)
-- Execute sql that is stored in a table.
DECLARE @var varchar(50)
SET @var = (SELECT * FROM sql)
DECLARE c CURSOR FOR SELECT sql FROM sql
OPEN c
DECLARE @sql varchar(max)
FETCH NEXT FROM c INTO @sql
WHILE @@FETCH_STATUS = 0
      BEGIN
            EXEC (@sql)
            FETCH NEXT FROM c INTO @sql
      END
CLOSE c
DEALLOCATE c
-- Dynamic SQL to look for nulls
declare @fullsql varchar(max),
        @sql varchar(255)
SET @fullsql = ''
declare curse insensitive cursor for
      SELECT 'SELECT '''+TABLE_NAME+''', '''+COLUMN_NAME+''',
count('+COLUMN_NAME+') as nulls, count(*)-count('+COLUMN_NAME+') as
notnulls FROM '+TABLE_NAME+'
      '
            FROM INFORMATION_SCHEMA.COLUMNS
open curse
fetch next from curse into @sql
```

Notes:

```
while @@fetch_status = 0
begin
      IF (@fullsql = '')
            SET @fullsql = @sql
      ELSE
            SET @fullsql = @fullsql + ' UNION ALL '+ @sql
    fetch next from curse into @sql
end -- while fetch
close curse
deallocate curse
exec(@fullsql)


----------
-- CSV EXPORT
--
-- How to export data to CSV format. Note: this is not supported by
-- SQLServer and requires some jumping through hoops.
----------
-- Start by setting the correct text format in the options screen
-- Set SQLCMD Mode from the Query menu
:OUT C:\directories\demo.csv
SET NOCOUNT ON; SELECT * FROM Customers


----------
-- CSV IMPORT
--
-- How to import data from CSV format. Note: this is not supported
-- by SQLServer and requires some jumping through hoops.
----------
BULK INSERT Staff FROM 'C:\directories\demo.csv'
      WITH (
            FIRSTROW = 2,
            FIELDTERMINATOR = ',',
            ROWTERMINATOR = '\n',
            TABLOCK
      )
-- If you want to overrule the IDENTITY column
```

Notes:

```
BULK INSERT Staff FROM 'C:\directories\demo.csv'
      WITH (
            FIRSTROW = 2,
            FIELDTERMINATOR = ',',
            ROWTERMINATOR = '\n',
            TABLOCK,
            KEEPIDENTITY
      )


----------
-- STATISTICS (Optional)
--
-- Look at the statistics engine and rules to update
----------
UPDATE STATISTICS Customers
-- Database Console Command
DBCC SHOW_STATISTICS (Customers, customerpk)


----------
-- PROFILER (Optional)
--
-- Initialisation from SSMS
-- Save to file/table
-- Show events; select errors>user errors and errors>error log
-- Show output > Run all queries > Show output
-- Show Performance Monitor
-- Show Data Collector and Reports
-- Run all queries
----------
```

Notes: