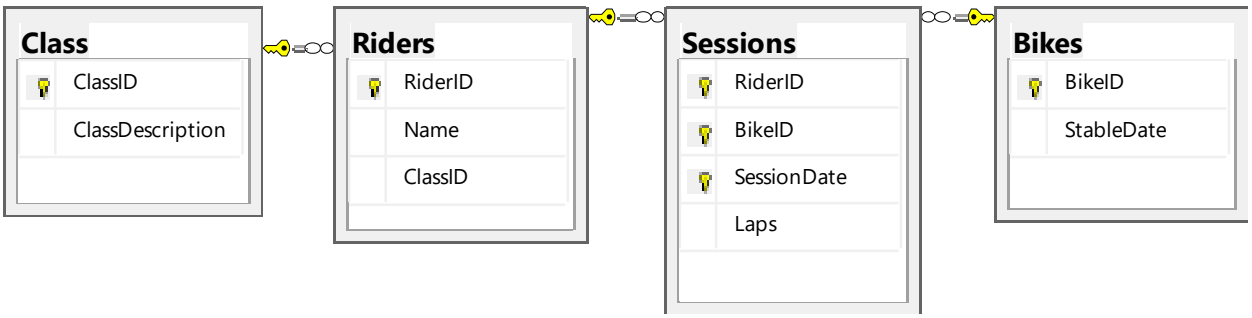
Lab01- Database Creation

Part I – Creation Scripts

In this lab/assignment, you will write the scripts to create a database, the tables within it, Preparing for the creation of the stored procedures required to operate it.

You are going to create a simple database to manage a racetrack and riding season.

Write the statements necessary to drop and recreate the following database with the tables shown, with the relationships shown:



Use the following sample data to determine the datatype for your fields, when a value is specified, you may assume it represents the biggest value that might be required :

Class :
ClassID : 'moto_2'
ClassDescription : 'Common 600cc engine and electronics, Custom Chassis'

Riders :
RiderID : 999
Name : 'Justhopi Gofastenough' (allow up to 64 characters)

Bikes:
BikeID : will be '**###X-A**', exactly, where the **###** is the bike number (ie. 005), **X** is Alphabetic representing manufacturers (H – Honda, Y – Yamaha, S – Suzuki), and **A** is Alphabetic representing an AM (A) or PM (P)
StableDate : a date field denoting bike arrival into the paddock

Sessions :
SessionDate : a datetime field

Table constraints

- field constraint : set RiderID to an auto increment field starting at 10
- field constraint : ensure SessionDate field is after 1 Sep 2017
- field constraint : ensure Name is longer than 4 characters
- field constraint : ensure BikeID has form characterized by '###X-A' format
- index : Session table : add composite index on RiderID and SessionID
- foreign key : add foreign key for Riders to Class
- field constraint : Class primary key
- field constraint : Rider primary key
- table constraint : Sessions Composite primary key

Add after table creation via Alter table :

- field constraint : ensure Laps is not negative
- foreign key for Sessions to Riders and Sessions to Bikes
- primary key for Bikes

Your script must include a header block with your name, and assignment.

Create this database as YourUserName_lab01. Your appropriate drop/create statements must exist BUT be commented out at the top, for use if a wholesale removal is required, but your script shall run assuming the database exists, and must appropriately and individually drop/create all entity elements as required. It may include a 'using' command to allow use of shortened entity names.

Part II – Stored Procedures :

You must add stored procedures for the following operations. For all stored procedures, the return value will be >=0 on success, or -1 on any error. All stored procedures will contain an output parameter that will contain a text message indicating the outcome of the operation. It will contain 'OK' if the operation was a success, or a descriptive error message if it was not.

Population :

PopulateClass – stored procedure, will populate Class table with the following data:

```
values
	('moto_3', 'Default Chassis, custom 125cc engine'),
	('moto_2', 'Common 600cc engine and electronics, Custom Chassis'),
	('motogp', '1000cc Full Factory Spec, common electronics')
```

PopulateBikes - Bikes exist without a Rider or Sessions entered. You must write a stored procedure called PopulateBikes to populate the Bikes table, representing the current stable of bikes. The format for a BikeID will be:

###X-A

where :

is the bike number (ie. 005),

X is Alphabetic representing manufacturers (H – Honda, Y – Yamaha, S – Suzuki)

A is Alphabetic representing an AM (A) or PM (P) availability.

You must populate bikes 000 – 019 for each Manufacturer, one for AM and one for PM. Your stored procedure must programmatically generate the (20 * 3 * 2 = 120) BikeIDs, and should use a single multi-element INSERT to add the Bikes.

For the remaining stored procedures, you will preemptively avoid, programmatically, operations that would result in the database engine generating errors. This means that all issues related to referential integrity, deletion, addition, and modification will be checked for validity before they are passed to the database engine for execution.

- **AddRider** – Supplied Name and Class ID, if successful, return generated RiderID as the return value.
- **RemoveRider** – Supplied Rider ID, Optional bool parameter Force, default false - means remove all relevant Sessions, Error if Force is false and related Sessions exist.
- **AddSession** – Supplied Rider ID, Bike ID and SessionDate. No parameter for Laps, Laps initializes to 0, Must be a future date
- **UpdateSession** – Supplied Rider ID, Bike ID, SessionDate, Laps update must be greater than existing
- **RemoveClass** – Supplied Class ID. Manually cascade deletes through to Sessions
- **ClassInfo** - Supplied a ClassID, and optional RiderID, retrieve **all** relevant information

- **ClassSummary** - Optional ClassID and Optional RiderID, rollup summary for #sessions, average/min/max laps for **all**/each relevant Rider/Class. Riders must appear even if no relevant Session data exists - zeros should be evident.

** Optional arguments implies get all for that argument..

Here is a summary of the testing matrix for your procedures:

AddRider	Name !Null	CID.!Exists					
RemoveRider	RID Null	RID !Exists	!Force & Sessions	Force & Sessions			
AddSession	RID !Null	BID !Null	D Invalid	R.Exists	B Exists	B!Already Assigned	
UpdateSession	RID !Null	BID !Null	D !Null	!Exists	Laps Invalid		
RemoveClass	CID !Null	CID !Exists	R.Removed	S.Removed			
ClassInfo	CID !Null	RID!Null	!Sessons	!Riders			
ClassSummary	CID !Null	R !Null	!Sessions	!Riders	Zeros		

In order to be compliant for check-off, you must write procedure calls that validate and test all indicated (highlighted) control paths through all of your stored procedures. Use the table above, and ensure that you test all required conditions.

For example, AddRider should have 2 test executions, one successful, and one test execution verifying that using a non-existing ClassID fails appropriately. ** You still need ALL the checking code – but your test executions only verify the highlighted tests.

Sample procedure (local names used):

```
create procedure RemoveBike
@BikeID as char(6) = null,
@ErrorMessage as nvarchar(max) output
as
        if @BikeID is null
                begin
                        set @ErrorMessage = 'BikeID can''t be NULL'
                        return -1
                end
        if not exists ( select * from Bikes where BikeID = @BikeID )
                begin
                        set @ErrorMessage = 'Can''t remove : ' + @BikeID + ' doesn''t exist'
                        return -1
                end
        if exists ( select * from Sessions where BikeID = @BikeID )
                begin
                        set @ErrorMessage = 'Can''t remove - Currently in Session'
                        return -1
                end

        delete from Bikes where BikeID = @BikeID

        set @ErrorMessage = 'OK'
        return 0
go
-- Then you will execute your stored procedure
--   Explicitly exercise execution paths as highlighted, though you should verify all paths**
```