**School of Media Arts and Technology**

BSc (Hons) **Software Engineering**

**Academic Year 2019-2020**

**STEFAN SHKENDEROV**

**Star Trek Database**

Report

**Enterprise Database Development**

Contents

[Elicit requirements & Analysis 2](#_Toc31106721)

[Logical Database Design 3](#_Toc31106722)

[ERD: 3](#_Toc31106723)

[Functions 6](#_Toc31106724)

[Member Location: 7](#_Toc31106725)

[Distance between two star systems: 8](#_Toc31106726)

[Procedures 10](#_Toc31106727)

[Attack On a Spaceship: 10](#_Toc31106728)

[Beaming: 11](#_Toc31106729)

[Triggers 12](#_Toc31106730)

[Attack\_trigger 12](#_Toc31106731)

[Mission Trigger 12](#_Toc31106732)

[Performance enhancement 13](#_Toc31106733)

# Elicit requirements & Analysis

This database should adequately represent and store data about Star Trek and the Enterprise star ship. It will save information about its maintenance, warp speed status and crew members. The database will save information about the ship’s weapons, which will include stats like cost and power. The design is so that the weapons are treated individually, instead of by type. This means that a weapon can be only on one ship, and a ship can have many weapons (one-to-many relationship). The crew members are a separate table as well as the crew itself. The members have location, which can be saved in the captain’s log. The log can be accessed and edited only by the captain. The crew consists of the members with additional attributes like experience, battles won and battles lost. For simplicity, the location is saved as a varchar value. The navigation between the star systems is going to be handled via saving each one’s location in a separate table, in X-Y-Z coordinates and then the distance will be measured by the coordinates distance formula.

# Logical Database Design

## ERD:

The ERD shows that the database has 8 tables. The ship’s qualities are entered in the Ship table, except for the location line, which is a reference to the location table. Locations are in a separate table because they can be used to identify either where a star ship is or where a star system is located. The Earth will be used in our case as the centre of the Coordinate system, i.e. the point (0,0,0). The location table is connected with the database only using one-to-many relationships and it does not have any foreign keys in itself, meaning that no two combination of star systems or ships can be at the same spot. The star system table consists of further information about a star system that has been entered. In the mission table there are 3 foreign keys – to the tables ship, crew and star\_system. The database is designed so that there are no multi – ship missions, as well as that a mission can be only on one star system and only one crew can part take. All of this means that the crew is specific to the mission and not the star ship, which is what the case is in the movies from my research. For example the Enterprise ship changes crew multiple times.

The crew consists of people from the person table. One person can be only in one crew at a time.

The weapons are specific to the ships as well, allowing one ship to have many weapons, but the weapons themselves can only be on that ship.

Ship table :

1. Id – Number(6), Primary key
2. Health – Like in videogames, 100 is full health and 0 is broken. Number(3) DEFAULT 100
3. Name – varchar2 (30) NOT NULL
4. Warp\_drive\_status – number(3) DEFAULT 0
5. Maintenance – A number representing the cost of the annual maintenance of the ship – float(15) DEFAULT 0
6. Year\_made number(3) NOT NULL
7. Location – foreign key, referencing the Location table. Represents the current location of the ship via x-y-z coordinates.

Location Table:

1. Id – Number(6), Primary key
2. coord\_x – float(15) NOT NULL
3. coord\_y – float(15) NOT NULL
4. coord\_z – float(15) NOT NULL

*As described in the previous section, all of the coordinates have to exist upon an entry in order for the location to be represented in three dimensional space.*

Star System Table:

1. Id – Number(6), Primary key
2. Name – varrchar2(30) NOT NULL
3. Description – Provides information about the star system if needed varchar2(100) DEFAULT NULL
4. Location – A foreign key which references the Location table number(6) NOT NULL

Weapon Table:

1. Id – number(6) Primary Key
2. Name – varchar2(30) NOT NULL,
3. Power – A number showing how powerful the weapon is – number(10) NOT NULL
4. Accuracy – A number showing how accurate the weapon is – number(10) NOT NULL
5. Cost – A number showing the cost of the weapon – number(10) NOT NULL

Mission Table:

1. Id – Primary key – number(6)
2. Name – varchar2(30) NOT NULL,
3. start\_date – DATE NOT NULL,
4. end\_date – DATE NOT NULL,
5. description – varchar2(200) NOT NULL,
6. ship – Foreign key, referencing the ship that will do the mission – number(6) NOT NULL,
7. crew – Foreign key, referencing the crew, that will do the mission -number(6) NOT NULL,
8. star\_system – Foreign key, referencing in which star system the mission is carried out - number(6) NOT NULL

Crew Table:

1. Id – Primary key - NUMBER(6)
2. name varchar2(30) NOT NULL,
3. people – The number of people in the crew - number(3) DEFAULT 0,
4. battles\_won NUMBER DEFAULT 0
5. battles\_lost NUMBER DEFAULT 0
6. experience\_rating – The experience rating of the crew number(3) DEFAULT 0

Person Table:

1. id – Primary Key - number(6)
2. first\_name - varchar2(30) NOT NULL,
3. econd\_name - varchar2(30) NOT NUL
4. age number(2) - NOT NULL
5. Location – Foreign key, referencing the location table varchar2(30) NOT NULL,
6. kill\_level number(3) - DEFAULT 0
7. Role varchar(15) NOT NULL
8. Crew – Foreign key, referencing the crew table, showing which person belongs to which crew number(6) NOT NULL

*The is\_captain entry has constraints, limiting it to either 0 or 1, effectively making it a Boolean variable. If 1, then the person is captain and can access the Captain’s log.*

Captain’s Log Table:

1. Id – Primary key - NUMBER(6)
2. Date - DATE NOT NULL,
3. Heading - varchar2(30) NOT NULL,
4. Text – An optional entry, if the captain wants to log only a member’s location - varchar2(300) DEFAULT NULL,
5. member\_name - varchar2(30) DEFAULT NULL,
6. member\_location - varchar2(30) DEFAULT NULL

To test the database, it is populated with data via the INSERT INTO command. The data consists of entering the star systems of the earth and the Romulan Planet with its locations, the Enterprise Star Ship and its location and two of its weapons and the crew of Cpt Kirk on an exploration mission on the Romulus Planet from the year 2264 to 2269.

Because of the size of the crew, operating the ship(>400) during that mission, only two members’ data is entered in the “person” table for the testing purposes – Cpt. Kirk and Reginald Barclay.

# Functions

*DISCLAIMER: A person’s location is differently represented to that of a location of a star system or a ship. As explained in the first part of the report, the latter is represented by 3 values in a coordinate system and they are recorded in the “location” table. The crew’s location is represented by a varchar2 value, which is a name of the ship deck/room where they are currently, or if they are outside of the ship.*

## Member Location:

This function returns the location of a member of the crew:

CREATE OR REPLACE FUNCTION member\_location (member\_f\_name person.first\_name%TYPE,member\_s\_name person.second\_name%TYPE)

RETURN varchar2 IS

location\_m varchar2(30);

BEGIN

SELECT p.location

INTO location\_m

FROM person p

WHERE p.first\_name = member\_f\_name AND p.second\_name=member\_s\_name;

RETURN location\_m;

EXCEPTION

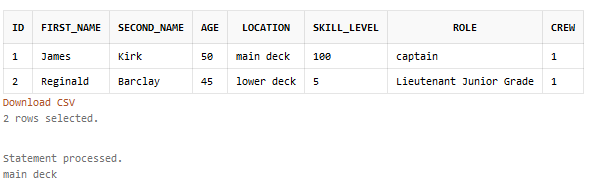
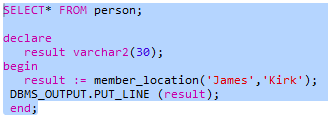
WHEN NO\_DATA\_FOUND

THEN DBMS\_OUTPUT.PUT\_LINE ('No data found.');

END;

/

To compare the function result to a query and test the function, test code is written(top picture) and the result is shown below:



## Distance between two star systems:

The three-dimensional space has three coordinates for the position of an object. Having a separate “location” table enables us to extend the functionality of the coord\_distance function ,so it can be used to calculate the distance not only between two star systems, but between two ships or a star system and a ship as well. In the formula, shown below, A and B are the end points of navigation => AB is the distance between them. The result is always a positive number.



The function is split into 2 parts, the first one takes the coordinates as parameters and calculates the distance via the formula, and the second part has 3 versions, one for finding the distance between two star systems, the second is for finding the distance between two ships, and the third one is for finding the distance between a ship and a star system. That function grabs the coordinates needed via a select statement and parses them onto the coord\_distance function. The 3 versions are different only in that SELECT statement. In this report only one versions is shown, the others are in the script file and they are called ship\_distance and ship\_star\_system\_distance.

The code for the function calculating the distance:

CREATE OR REPLACE FUNCTION coord\_distance(coord\_x1 FLOAT,coord\_y1 FLOAT,coord\_z1 FLOAT, coord\_x2 FLOAT, coord\_y2 FLOAT, coord\_z2 FLOAT)

RETURN FLOAT IS

return\_val FLOAT;

BEGIN

return\_val := SQRT(POWER((coord\_x1-coord\_x2),2)+POWER((coord\_y1-coord\_y2),2)+POWER((coord\_z1-coord\_z2),2));

RETURN return\_val;

END;

/

The code for the function, selecting the values and calling the coord\_distance function, then outputting the result for a distance between two Star Systems .

CREATE OR REPLACE FUNCTION star\_system\_distance(system\_1\_name varchar2, system\_2\_name varchar2)

RETURN NUMBER IS

distance FLOAT;

coord\_x1 FLOAT;

coord\_y1 FLOAT;

coord\_z1 FLOAT;

coord\_x2 FLOAT;

coord\_y2 FLOAT;

coord\_z2 FLOAT;

BEGIN

SELECT l.coord\_x , l.coord\_y, l.coord\_z

INTO coord\_x1, coord\_y1, coord\_z1

FROM location l, star\_system s

WHERE system\_1\_name = s.name AND s.location=l.id;

SELECT loc.coord\_x , loc.coord\_y, loc.coord\_z

INTO coord\_x2 ,coord\_y2 ,coord\_z2

FROM location loc , star\_system ss

WHERE system\_2\_name = ss.name AND ss.location=loc.id;

distance := coord\_distance(coord\_x1, coord\_y1, coord\_z1, coord\_x2 ,coord\_y2 ,coord\_z2 );

RETURN distance ;

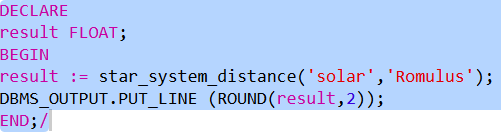
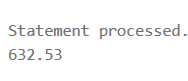
EXCEPTION

WHEN NO\_DATA\_FOUND

THEN DBMS\_OUTPUT.PUT\_LINE ('No data found.');

END;/

The test code on the left and the result is on the right for calculating the distance between Romulus (coordinates (200,10,600) and the Solar System (coordinates (0,0,0), rounded up to two numbers after the decimal point.

# Procedures

## Attack On a Spaceship:

The following procedure takes 2 variables as arguments – the name of the ship and the damage taken and alters the entry of a ship with the same name and deducts the damage taken from the health of the ship. If a ship with no such name as the entered one is found, then a message prints out. The code is below:

CREATE OR REPLACE PROCEDURE ship\_attacked(ship\_name IN varchar2,hit\_pts IN NUMBER) IS

old\_health NUMBER;

new\_health NUMBER;

BEGIN

SELECT s.health

INTO old\_health

FROM ship s

WHERE ship\_name = s.name;

new\_health:=old\_health-hit\_pts;

UPDATE ship

SET ship.health=new\_health

WHERE ship.name=ship\_name;

EXCEPTION

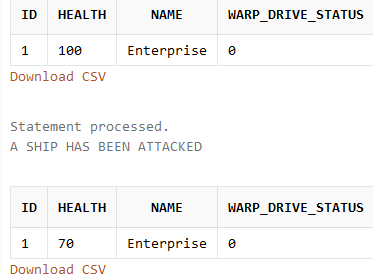
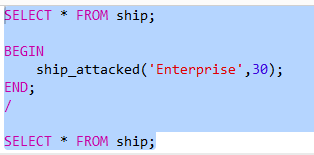
WHEN NO\_DATA\_FOUND

THEN DBMS\_OUTPUT.PUT\_LINE ('No data found. wrong ship name.');

END;

/

A screenshot of the procedure working is shown below:



## Beaming:

The beaming procedure changes a person’s location by using a query for his names and parsing the new location as an argument.

CREATE OR REPLACE PROCEDURE beam(person\_f\_name IN varchar2,person\_s\_name IN varchar2,new\_location IN varchar2) IS

BEGIN

UPDATE person p

SET p.location=new\_location

WHERE p.first\_name=person\_f\_name AND person\_s\_name =p.second\_name ;

EXCEPTION

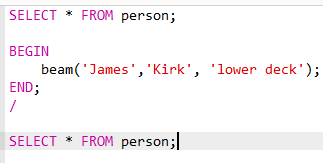
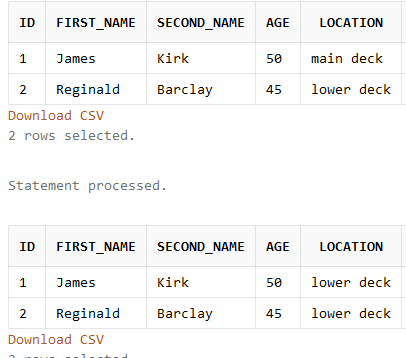
WHEN NO\_DATA\_FOUND

THEN DBMS\_OUTPUT.PUT\_LINE ('No data found. wrong name.');

END;

/

Results:

# Triggers

## Attack\_trigger

This trigger outputs a message when the ship is attacked, i.e. when the ship’s health is decreased. It can be seen working in the screenshot of the ship\_attacked procedure. The code is:

CREATE TRIGGER attack\_trigger

AFTER UPDATE

OF health ON ship

BEGIN

DBMS\_OUTPUT.PUT\_LINE('A SHIP HAS BEEN ATTACKED');

END;

/

Mission Trigger

This trigger returns a message if a new mission is added to the database. The code is:

CREATE TRIGGER mission\_trigger

AFTER INSERT ON mission

FOR EACH ROW

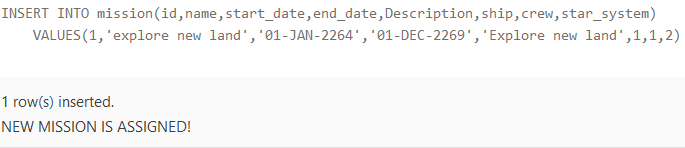
BEGIN

DBMS\_OUTPUT.PUT\_LINE('NEW MISSION IS ASSIGNED!');

END;

/

The trigger can be seen working upon running the script in the livesql website. This is one of the statements of that script:



# Performance enhancement

For the last requirement, indexes need to be created. In this database, 3 are created.  
The first one is people\_name\_i and it indexes the first and last names of the person table. The second one is on the captain’s log table and indexes the date. The third one is in the mission table and indexes the name, start\_date and end\_date columns. The code for all three is seen below:

CREATE INDEX people\_name\_i

ON person(second\_name,first\_name);

CREATE INDEX log\_i

ON captain\_log(date\_recorded);

CREATE INDEX mission\_i

ON mission(name,start\_date,end\_date);

In the case of this database, all of these indexes are appropriate for when the database itself grows in size. The indexes will make it faster, because they are implemented in the tables that will have the most entries.