MEDER KUTBIDIN UULU
DIAS ALYMBEKOV
CÜNEYT EREM

TUTORIAL:

Introduction

SQL(Street Querying Language) is a language which helps to its users to define streets and points, and make queries over them. The network of the streets and points can be so huge data, but SQL makes a query so simple. Users can define a starting point, an ending point and passing through points of their route. In this tutorial, the syntax of the SQL will be explained and examples will be provided in order to make it easy to understand.

Comments

Comments are an essential part of SQL language, they are supported in various ways:
-Single line comments are used in this way: // *after this sign nothing is being executed till the next new line.*
-Syntax of multiple line comments are used in this fashion: /* *No matter how many lines comes after this sign, they will not be executed unless it is closed with this sign* */

The street network definition language;

This language shows the street connections between points of the network.

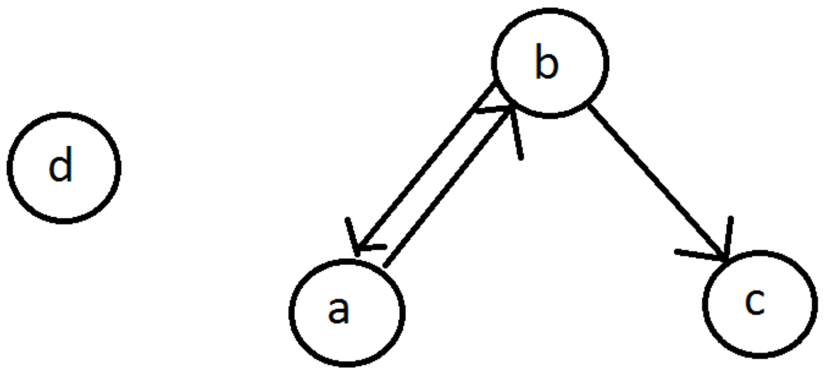
A: One-way street network:

It is an one directed route from specific point to another.
Route can be from point1 to point2, can be from point2 to point1 or both from point1 to point2 and from point2 to point1 separately.

To do one way street, "->", "<->" and "<-" should be apply between to points;

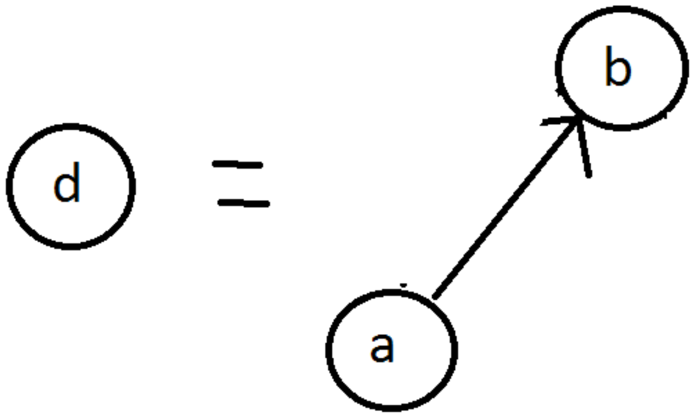
```
oneWayStreet network1 {
// a, b, c and d are points
        point a; point b; point c; point d;       //assume, points are initialized in other examples

        a->b;           //assign one-way from a to b
        b->c;           //assign one-way from b to c
        b->a;           //assign one-way from b to a
        point d ;
}
```

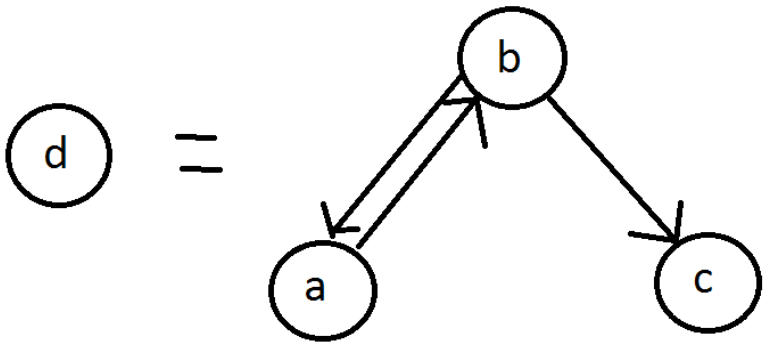


To apply these changes to another point, assign new point variable.

```
oneWayStreet network2 {
// a, b, c and d are points
        d  =  a->b;                 //assign one-way from a to b to point d (ex; d=ankara)
        point d ;
}
```



```
network2 {
// a, b, c and d are points
         b->c;            //update one-way from  b to c
         b->a;            //update one-way from  b to a
}
```



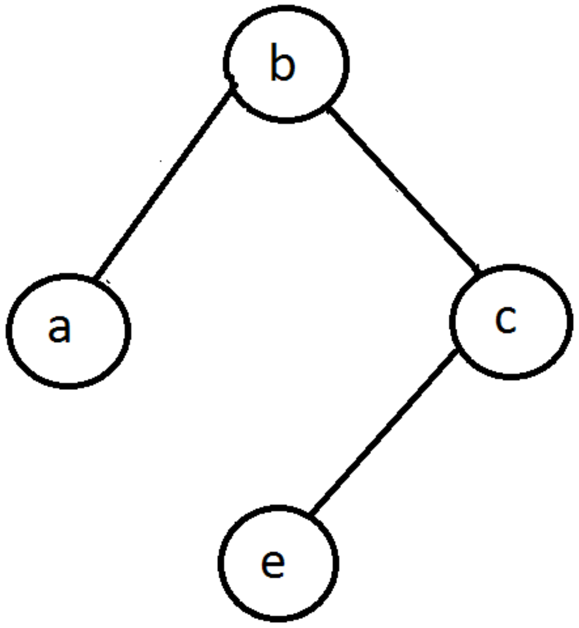
B: Only two-way street network

It is an only two-directed route from specific point to another. It can be from point1 to point2 and from point2 to point1.

To do one way street, “ --- ” should be apply between to points;

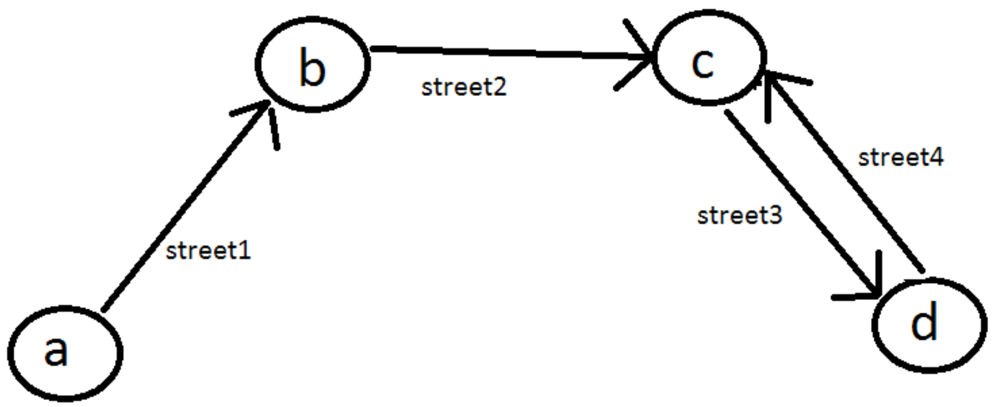
```
twoWayStreet network3 {
// a, b, c and e are points
        street street1;              //assume, streets are initialized in other examples

        a --- b --- c;               //assign two-way network between a and b, b and c.
        street1 = e --- c;           //define street1 instead of using point in prev examples
}
```



C/D: Defining point and street properties

Multiple properties can be attached to both point and street. For the points, they have multiple ‘name’ and ‘value’ features. For the streets, they have avarage time passing property and some temporary properties (such as delay). Points and streets can be attached to each other with their properties.



```
oneWayStreet villageNetwork {
// a, b, c and d are points, street1, 2, 3 and 4 are streets.

        point a;
        point b;
        point c;
        point d;
        street street1;
        street street2;
        street street3;
        street street4;


        street1 = a->b;
        street2 = b->c;
        street3 = c->d;
        street4 = c<-d;

        //point properties

        point a {
        //gas station
                a.”stationName” = “Bilkent Petrol”
                a.”type” = “gasStation”
```

```
	}
	point b {
	//house
		b.“name” = “Wammy’s house”
	}
	point c {
		c.“name” = “headquarters”
		c.“type” = “militaryStation”
	}
	point d {
		“name” = “unknown”
		”paymentOption” = “cashOnly”
	}


	//street properties

	street street1 {
		“avgtime” = “60”
	}
	street street2 {
		“avgtime” = “40”
	}
	street street3 {
		“avgtime” = “20”
		“delay” = “10”
	}
	street street4 {
		“avgtime” = “20.5”
		“delay” = “15.511”
	}

}
```

Points and streets can have base properties and other optional properties.

E: Dynamic type system for properties:

Primitive types & Variables
In SQL there are 4 primitive type variables:

- Integers

- Boolean
- String
- Float

It is dynamically typed language, so there is no need to specify the type of a variable.
All types of variables can be assigned to the same variable, and it is set at a run-time, the last assigned value will remain as a value of that particular variable. For instance,

```
$example = 5;
$example = 5.4;
$example = “This is an example string”
$example = true;
```

As you can see, all types of variables are assigned to $example, and the value of this variable is true, because it changes at a run-time.

There is no character limitation in naming a variable.

Use of Integers, Floating Point Numbers, Strings and Booleans;

Basic arithmetic functions are being supported by Integers and Floating Point Numbers. Subtraction, Addition, Multiplication and Division are similar to any other programming languages:

```
$first = 4;        $second = 2;
$result = $first + $second; // 6
$result = $first - $second; // 2
$result = $first * $second; // 8
$result = $first / $second; // 2
```

However when these operations come consecutively, the priority from high to low is as follows: Bracket, Exponent, Division or Multiplication, Subtraction or Addition.

If the multiple operations come together, then the statement is being executed from left to right, however an exponent operation is being executed from right to left. For instance,

```
$result = 4 + 3 - 2; // 5
$result = 12/4/2; //1.5
$result = 2^3^3; // 2^(3^3)
```

Strings are concatenated with ‘+’ sign. Integers and Floats are converted to string when they concatenated with a string.

```
        $firstName = “Meder”;
        $lastName = “Kutbidin”;
        $birthday = 12;
        $fullInfo = $firstName +” ” +  $lastName + “ “ + $birthday; // “Meder Kutbidin 12
```

Boolean can get only two values, either true or false and it supports following logical operations:

- AND (&&)
- OR (||)
- INVERSE (!)

LIST: This street network language supports lists which uses strings and integers for the points and streets. To add list variables, point and its name will be written, and list can be inserted after “=” operation. It is same for the streets aldo but it uses integer and float variables. To add new variable, there will be some operations such as add().

```
twoWayStreet network3 {

        point a;
        street street1;
        street street2;

        a.”sells” = list1[“LPG”, “Diesel”]      //point list as string

        street1.”avgtime” = list2[15, 30]       //street list as integer

        street2.”avgtime” = list3[15.0, 30.5]  //street list as float

        street1.”avgtime”.add(20);              //list2[15, 30, 20]
        //there may some ops like add() operation
}
```

SET: This street network language supports sets which uses strings and integers for the points and streets. It does some operations such as ‘+’, ‘-’, ‘|’, ‘&’ (adding, difference ect.) to update set.

```
twoWayStreet network3 {
        point a;
        street street1;
        street street2;

        a.”paymentOpition” = set1[“Cash”, “Visa”]  //point set as string

        a.”paymentOpition” = a.”paymentOpition” + “Mastercard”;

        //a.”paymentOpition” becomes =  set1[“Cash”, “Visa”, “Mastercard”]

        street1.”delay” = set2[10, 20]          //street set as integer

        street2.”delay” = set3[5.0, 10.5]       //street set as float
}
```

MAP: This street network language supports maps which uses strings and integers for the points and streets. It keeps data in pairs and easy to write. It can change variables from string to list, or string to int/float etc.

```
twoWayStreet network3 {

        point a;
        point b;
        street street1;

        a.”sells” = map1[“gas-types” : [“LPG”, “Diesel”]]                       //string to list
        b.”sells” = map2[“payment-options” : [“Cash”, “Visa”, “Mastercard”]]    //string to list
        street1.”avgtime” = map3[“time” : [35]]                                 string to int
}
```

FUNCTIONS:

There are several functions which are supported by SQL. Those built-in functions are aimed to make users’ life easier. Here is the list of the functions:

length():
when this function is called, it returns the length of a collection(arrays, sets, maps).

sqrt(x):
it returns the square root of the given variable x.

power(x, y):

it returns x to the power of y.

absValue(x):
it returns the absolute value of given variable x.

contain(x):
If the given x is contained in the variable which is calling this function, then it returns true, false otherwise

QUERY:

a)Creating route queries:

Querying in the language can be supported by a specific grammar.
Query is an expression specifying a route while visiting several points and/or streets. To find this route user needs can specify the filtering constraints as Boolean expressions, which are composed of predicates defined over street properties as well as incident point properties. << operator is used for this purpose.

route = network >> [“name” = “Bilkent Building 81”] < type=’gas station’; sells = “Diesel” > [“type” ==”Shopping Mall”];

The type of expression above finds all routes from a specific building at Bilkent to a shopping mall, the route passes through a point with properties ‘gas station. [] inside the first brackets user specifies the starting point or street, then between <> brackets user can specify the point or street properties that needs to be included in the route, several properties can be included if separated by ‘;’ character, finally between last [] brackets in the end user specifies the final point or the street. User can leave one of the brackets empty in order to perform wider search.

User can specify alternation and repetition in the route query. This operations are performed using || , * operations respectively. Concatenation performed automatically if two parts are not separated by space.

The language also supports existence predicates and arithmetic expressions and functions in predicate expressions. Supported functions are listed in the functions part.

route = network >> [“name”[0] = “A”] < “avg_time” < sqrt(16) > [“type” = “Shopping mall”];

The query above performs search for all routes from all points that start with A, The route must contain street with avg_time less than 4, the final destination is Shopping mall.

If user wants to to find the shortest route in time, shortest route in distance or the simplest route, then following expressions have to be used inside the middle brackets.

<*min_time* = *true*> for the query with the shortest route in terms of time.
<*min_dist* = *true*> for the query with the shortest route in terms of distance
<*simplest* = *true*> for the query with the simplest route.

In order to limit the number of routes in the resulting query user can specify the following expressions in the middle brackets: <limit_results = 3> the following expression will show the best three three routes.

route = network >> [“name” = “Bilkent Building 81”] <min_time = true> [“type” ==”Shopping Mall”];

route = network >> [“name” = “Bilkent Building 81”]<min_dist = true> [“type” ==”Shopping Mall”];

route = network >> [“name” = “Bilkent Building 81”]<simplest = true> [“type” ==”Shopping Mall”];

route = network >> [“name” = “Bilkent Building 81”]<limit_results = 3>[“type” ==”Shopping Mall”];

Expressions above demonstrate the use of the additional constraints in the query.

The language supports variables in the query. If the values of properties are dynamic then variables can be used in the request.

route = network >> [“name” = $point]<limit_results = 3>[“type” == $point];

The expression above finds the best three results for the points that are going to be assigned in the future.

The language also supports modularity, which means that user can divide regular route queries into multiple pieces, they can be specified separately. This feature is realised with the use of variables. One query variable can be assigned to several others, in this way final query will consist of requests from all subqueries. In order to organise queries user can make use of alternation, concatenation and repetition.

*query1 = query2 (query3 || query4) **

The query1 unites the query2 with query3 or query4, all possible requests are included in the query 1.