Graph and Cypher

BangDB 2.0, User Guide

1. Introduction

Data in a graph table for BangDB is defined as triples. A triple contains subject, object and relationship (predicate) between them. All data is stored as triple within the DB. BangDB does clever arrangements and housekeeping to store the data such that various queries can be written and run efficiently.
The structure of the query is very similar to “Cypher”. BangDB uses Cypher-like queries to process the data. The basic structures look like following.

```
CREATE () -[]-> ()          - for creating node or triple

S=>() -[]-> ()              - for querying data

<op USING attr1 SORT_DESC attr2 LIMIT n>    - operation on disjoint sets of queries
query1
++
query2
```

The ‘()’ denotes subject or object and ‘[]’ denotes relation (predicate) with ‘->’ defining the direction. The arrangement is always “subject Predicate Object”.

The node has a label associated with it. Every node is written as “label:name”.

There are basically following keywords associated with all the queries.

Node, entity creation

CREATE	- to create a single node, or triple

Running query and selecting data

S=>	- namespace for the unit of query
RETURN	- selecting attributes for any query
WHERE	- conditions for the query
AS	- selecting columns/attributes with alias
DATAQUERY	- for filtering within node and relations for properties
SORT_DESC	- for sorting in descending order
SORT_ASC	- for sorting in ascending order
LIMIT	- for limiting number of selections

Statistics

COUNT	- counting all using COUNT(*) or COUNT(A.col)
UCOUNT	- unique counting
AVG	- average of any attribute
MIN	- min value
MAX	- max value
STD	- standard deviation
SUM	- sum
EXKURT	- ex-kurtosis
SKEW	- skewness

Functional properties

SYMM	- symmetric relations
ASYMM	- asymmetric relations

Graph algos

ALL_PATH	- all paths between any two given nodes
SHORT_PATH	- shortest path between any two given nodes

Set operations (op)

ADD	- adding two or more sets (UNION)
CROSS	- cross product of two sets (INTERSECT)
SUBTRACT	- difference of two sets (DIFFERENCE)

Data Science

SIMILARITY	- compute similarities among set of nodes based on various data
CLUSTER	- to find and create natural clusters
CENTRALITY	- finding the node centrality

COMMUNITY_DETECTION – for detecting several communities within graph

GROUPS	- finding several groups given properties
ML_ALGO	- this brings entire ML algorithms to the Graph, model name is supplied as well
Deep Learning	- DNN, RNN, ResNet. Embeddable within graph

Information Extraction – Ontologies or triple generation through IE

Data is processed from left to right. There could be several triples chained to form a query, like.

S1=>() -[]-> () -[]-> () …

Here in the above example, the first triple will intermediate-output a set of results, these intermediate-output will become input from subsequent processing etc. Therefore, it will keep evaluating from left to right using the intermediate results. The subject for subsequent chained query will be the intermediate result of the previous triple and so on.

In some cases, we would like to keep object of the triple as subject for the subsequent triple, then we can use the structure like following.

S2=>[S1=>() -[]-> ()] -[]-> () …

We will see the examples for these in subsequent sections
We will use BangDB CLI to perform these exercises. But before we go there, let's see how BangDB Cypher is different from the original Cypher

2. Difference between BangDB and Neo4j – very high level

Following are basic differences from DB point of view

- BangDB is multi-model database which can deal with most of the data types that we come across in real world scenario. This means we can store different data as it should be rather than forcing it to form one form or the other
- BangDB has natively integrated Graph with streaming component in the DB, this makes it very easy for users to create Graph as data streams into the system. Further, it allows users to apply CEP on the data to find patterns/anomaly in continuous manner

- AI is native part of BangDB, which means we don't need a silo or component external to the database. This is very important for simplification of model building/training exercise and also for automation of the processes. Further ontologies could be created which allows users to build graph in probabilistic manner using a text file/data
- BangDB performance (IOPS) is much better than Neo4j, this helps in dealing with fast moving data for real-time analysis, and also for better efficiency. Creating nodes and relations 10X faster in BangDB and for triple creation, BangDB goes beyond 100X for data set which is large enough (10M + nodes and entities)
- BangDB can be used as full-fledged NoSQL DB which supports rich support for data organization and query with many traditional and advanced database concepts going way beyond Graph concepts

3. Difference between Cypher used by BangDB, and Cypher used by Neo4j

Following are basic differences from Cypher point of view.

- For all queries (retrieval), the query starts with "<token>=>", instead of "Match"
- The queries could be nested using the "[]" bracket which becomes subject for subsequent nodes. For example, look at following two queries.

    - S2=>[S1=>(Person:*)-[ACTED_IN]->(Movie:*)]-[LIVES_IN]->(City:Bangalore)
    - S1=>(Person:*)-[ACTED_IN]->(Movie:*)-[DIRECTED_BY]->(Person:*)-[LIVES_IN]->(City:SFO)

  First one will first fetch all the persons who have acted in any movies, and who lives in city Bangalore
  Second one will fetch all the persons who live in "SFO" who have directed movies in which some persons have acted in etc…
- Specific methods like; #ALL_PATH, #SHORT_PATH, #ALL_NODE is supported to find all paths between any two nodes, shortest path between two nodes and all nodes in between two nodes respectively
- Here we use "label:name" instead of "name"label" as in Cypher. This is mainly because it's more intuitive to use labels first and then name. If we wish to take any name, we can put '*' in place of it. In-fact we can put '*' for label or relation as well
- The variable or alias begins with'@', like (@p Person:Sachin) etc.
- All data science or ML related activities can be done using the Keywords like "SIMILARITY", "CLUSTER", "CENTRALITY", "COMMUNITY_DETECTION", "GROUPS" to leverage inbuilt dedicated ML+Graph processing. However, we can bring the entire ML algos to Graph by simply using "ML_ALGO" keyword along with the model's name that is trained within BangDB

4. Creating graph table

We must create and use a graph table before storing and querying data. There could be different tables that we can create within a db to keep different kinds of network data as we like

USE GRAPH <table_name>

For example, to create or use table my_graph_table

```
bangdb> CREATE GRAPH my_graph_table
```

5. Creating data

'CREATE' keyword should be used to create data.

Let's create movie Matrix (single node)

```
bangdb> CREATE (Movie:Matrix { "title":"The Matrix","released":1997 })
{
  "msg" : [
    "success"
  ],
  "errcode" : 0
}
```

Let's create triple here, Tom Hanks and Forrest Gump nodes with relation ACTED_IN as predicate

```
bangdb> CREATE (Person:"Tom Hanks" { "born":1956 })-[ACTED_IN { "roles": ["Forrest"]}]->(Movie:"Forrest Gump" { "released":1994 })
{
  "msg" : [
    "success"
  ],
  "errcode" : 0
}
```

```
bangdb> CREATE ("Person:Robert Zemeckis" { "born":1951 })-[DIRECTED]->("Movie:Forrest Gump")
{
  "msg" : [
    "success"
  ],
  "errcode" : 0
}
```

6. Matching patterns (querying data)

In BangDB we follow a similar to Cypher format for matching data or preferably querying data. However, BangDB doesn't use 'MATCH' keyword/clause for the same instead it uses something like <var_name>=> before the query

For ex; S=>

To find the data we've created so far, we can start looking for all nodes labelled with the Movie label.

```
bangdb> S=>(Movie:*)
{
  "num_items" : 2,
  "nodes" : [
    "{\"released\":1994,\"label\":\"Movie\",\"name\":\"Forrest Gump\",\"_pk\":\"889565392:607499282203478465\",\"_v\":1}",
    "{\"title\":\"The Matrix\",\"released\":1997,\"label\":\"Movie\",\"name\":\"Matrix\",\"_pk\":\"889565392:7731255011213178614\",\"_v\":1}"
  ]
}
```

Note: the _pk and _v is added by the db for its housekeeping reasons

We can also look for a specific Movie, like "Forrest Gump"

```
bangdb> S=>(@p "Person:Tom Hanks")-[@r ACTED_IN]->(@m Movie:*); RETURN m.name AS Title, r.roles AS Role
+---------------+------------------+
| Role          | Title            |
+---------------+------------------+
| ["Forrest"]  | Forrest Gump |
+---------------+------------------+

We could also see in json format by setting output as JSON
{
```

```
  "rows" : [
    {
      "Title" : "Forrest Gump",
      "Role" : "[\"Forrest\"]"
    }
  ]
}
```

Let's add another film where Tom Hanks acted in. Since Tom Hanks already exists hence, we may just use the name

```
bangdb> CREATE (Person:"Tom Hanks")-[ACTED_IN {"roles":["Zachry"]}]->(Movie:"Cloud Atlas" {"released":2012})
{
  "errcode" : 0,
  "msg" : [
    "success"
  ]
}
```

Now it's a good time to create another graph and do following.

```
CREATE GRAPH ex_graph
USE GRAPH ex_graph


CREATE (Movie:matrix {"title":"The Matrix", "released":1997})
CREATE (Movie:cloudAtlas {"title":"Cloud Atlas", "released":2012})
CREATE (Movie:forrestGump {"title":"Forrest Gump", "released":1994})
CREATE (Person:keanu {"fullname":"Keanu Reeves", "born":1964})
CREATE (Person:robert {"fullname":"Robert Zemeckis", "born":1951})
CREATE (Person:tom {"fullname":"Tom Hanks", "born":1956})
CREATE (Person:tom)-[ACTED_IN {"roles":["Forrest"]}]->(Movie:forrestGump)
CREATE (Person:tom)-[ACTED_IN {"roles":["Zachry"]}]->(Movie:cloudAtlas)
CREATE (Person:robert)-[DIRECTED]->(Movie:forrestGump)
```

7. Filtering results

Select all the people/ persons

```
bangdb> S=>(@p Person:*);RETURN p.name AS name, p.fullname AS FullName
+---------- +-----------------------+
|name     | FullName              |
+----------+-----------------------+
|tom        | Tom Hanks           |
+----------+-----------------------+
|keanu    | Keanu Reeves      |
+----------+-----------------------+
|robert    | Robert Zemeckis  |
+----------+-----------------------+
```

```
bangdb> S=>(@p Person:*); RETURN COUNT(*) AS People
{
  "count" : 3
}
```

Select movie title "The Matrix"

```
bangdb> S=>(Movie:* {title="The Matrix"})
{
  "num_items" : 1,
  "nodes" : [
    "{\"title\":\"The Matrix\",\"released\":1997,\"label\":\"Movie\",\"name\":\"matrix\",\"_pk\":\"889565392:7007891359330926487\",\"_v\":1}"
  ]
}
```

BangDB allows conditions to be also defined for filtering data for the properties within the node or relations, for example movie released > 2000

```
bangdb> S=>(@p Person:*)-[@r ACTED_IN]->(@m Movie:* {released> 2000}); RETURN p.fullname AS FullName, r.roles AS Roles, m.title AS Title
+-------------+---------------+----------------+
| FullName | Roles          |      Title       |
+-------------+---------------+----------------+
| Tom Hanks | ["Zachry"] | Cloud Atlas |
+-------------+---------------+----------------+

Or in json
{
  "rows" : [
    {
      "Title" : "Cloud Atlas",
      "Roles" : "[\"Zachry\"]",
      "FullName" : "Tom Hanks"
    }
  ]
}
```

8. Aggregation

Aggregation in Cypher (for BangDB) just works. Users select aggregate columns along with non-aggregated columns. The non-aggregated columns become groupby keys.

To find out how often an actor and director worked together, run the following

```
bangdb> S=>(@p Person:*)-[@d DIRECTED]->(@m Movie:*)<-[@a ACTED_IN]-(@p2 Person:*); RETURN p2.name AS actor, p.name AS director, COUNT(*) AS collabs

+-------+-----------+---------+
| actor | director | collabs |
+-------+-----------+---------+
| tom   | robert   |     1    |
+-------+-----------+---------+
```

To find how many times actors have had appearances

```
bangdb> S=>(@p Person:*)-[@a ACTED_IN]->(@m Movie:*); RETURN p.fullname AS actor, COUNT(*) AS appearances

+---------------+------------------+
| actor          | appearances |
+---------------+------------------+
| Tom Hanks | 2                   |
+---------------+------------------+
```

9. Shortest path

One of the most important tasks with graph is to find if there is a connection between two nodes and if there are more than one then finds the shortest path

BangDB supports this with minor modifications in the Cypher construct (but to simplify only) and use this for finding the shortest path, here is how it goes

Let’s add a triple to have some multi-level connection. Let’s add a triple for movie “forrestGump” winning “oscar”

```
bangdb> CREATE (Movie:forrestGump)-[WON_AWARD]->(Award:oscar)
{
  "msg" : [
    "success"
  ],
  "errcode" : 0
}
```

Now, let's find the shortest path between Tom Hanks and Oscar

```
bangdb> S=>(Person:tom)-[#SHORT_PATH *]->(Award:oscar)
{
  "nodes" : [
    {
      "label" : "Person",
      "name" : "tom",
      "_pk" : "2471870506:978343320853564008"
    },
    {
      "_pk" : "889565392:3325115057284420805",
      "name" : "forrestGump",
      "label" : "Movie"
    },
    {
      "label" : "Award",
      "name" : "oscar",
      "_pk" : "3098302716:362283378203006142"
    }
  ],
  "path_count" : 1,
  "edges" : [
    {
      "rel" : "ACTED_IN",
      "_to_node" : "889565392:3325115057284420805",
      "_from_node" : "2471870506:978343320853564008"
    },
    {
      "_to_node" : "3098302716:362283378203006142",
      "_from_node" : "889565392:3325115057284420805",
      "rel" : "WON_AWARD"
    }
  ]
}
```

As you see, the returned result describes the path. If there was more than one path (more than one movie of Tom Hanks winning an Oscar, then it would have returned the shortest path one. TO list all such paths between two nodes, we can use #ALL_PATH as shown in next section

10. All paths

```
bangdb> S=>(Person:tom)-[#ALL_PATH *]->(Award:oscar)
{
  "paths" : [
    {
      "nodes" : [
        {
          "label" : "Person",
          "_pk" : "2471870506:978343320853564008",
          "name" : "tom"
        },
        {
          "_pk" : "889565392:3325115057284420805",
          "label" : "Movie",
          "name" : "forrestGump"
        },
        {
          "label" : "Award",
```

```
            "_pk" : "3098302716:362283378203006142",
            "name" : "oscar"
          }
        ],
        "edges" : [
          {
            "_from_node" : "2471870506:978343320853564008",
            "_to_node" : "889565392:3325115057284420805",
            "rel" : "ACTED_IN"
          },
          {
            "_from_node" : "889565392:3325115057284420805",
            "rel" : "WON_AWARD",
            "_to_node" : "3098302716:362283378203006142"
          }
        ]
      }
    ],
    "path_count" : 1
}
```

11. Query for no direct relation

First create a graph table

```
bangdb> CREATE GRAPH g
bangdb> USE GRAPH g
```

Let's create following nodes and triples

```
CREATE (Person:john {"firstname":"John"})
CREATE (Person:joe {"firstname":"Joe"})
CREATE (Person:steve {"firstname":"Steve"})
CREATE (Person:sara {"firstname":"Sara"})
CREATE (Person:maria {"firstname":"Maria"})
CREATE (Person:john)-[FRIEND]->(Person:joe)-[FRIEND]->(Person:steve)
CREATE (Person:john)-[FRIEND]->(Person:sara)-[FRIEND]->(Person:maria)
```

query which finds a user called **'John'** and **'John's'** friends (though not his direct friends) before returning both **'John'** and any friends-of-friends that are found.

```
bangdb> S=>(@p Person:john)-[@f FRIEND]->(@p2 Person:*)-[@f2 FRIEND]->(@p3 Person:*); RETURN p.name AS self, p3.name AS fof
+------+--------+
| self  | fof     |
+------+--------+
| john | steve |
+------+--------+
| john | maria |
+------+--------+
```

Query with pattern in the value/field

Next up we will add filtering to set more parts in motion:
We take a list of user names and find all nodes with names from this list, match their friends and return only those followed users who have a 'name' property starting with 'S'.

```
bangdb> S=>(@p Person:*)-[@f1 FRIEND]->(@p2 Person:* {name = "s$%"});RETURN p.name, p2.name

+--------+--------+
| name | name |
+--------+--------+
| steve | steve  |
+--------+--------+
| sara   | sara   |
+--------+--------+
```

Note: Adding '$%' at the end of pattern will query for all such data where the pattern is satisfied

12. Graph in Stream

BangDB streams can also use Graph for the events which are time-series data. To use Graph in the stream, we may simply edit the schema and add set of relations as needed.
Example;

Let's say we have a schema where we receive purchase events for e-commerce and we would like to link user and the product he/she purchases, then we would simply add following;

"rels":[{"sub":"user","obj":"product","rel":"BUYS"}]

Here is the short schema definition for example;

```
{
        "schema":"grpahschema",
        "streams":[
                        {
                        "name":"order","type":1, "swsz":86400,
                        "inpt":[],
                        "attr":[
                {"name":"product","type":"STRING","sidx":1,"kysz":32,"stat":"UCOUNT"},
                        {"name":"user","type":"STRING","sidx":1,"stat":"UCOUNT"}
                         ],
                        "rels":[{"sub":"user","obj":"product","rel":"BUYS"}]
                        }
        ]
}
```

that's it, where the "user" and "product" are attributes in the Order stream.
Once defined, db would simply keep creating the relations (and creating nodes as required).

To query, we can use Cypher as described above using usual methods.

Now, let's look at some of the topics in detail and in advance sense, also we will introduce SYMM, ASYMM, AGGREGATION in detail, Filtering in detail, SHORT_PATH and ALL_PATH in more complex situations and SIMILARITY, CLUSTER for merging ML, Cypher in a unique way to provide the result in simple send

13. SYMM and ASYMM keywords

Create the graph table first

```
CREATE GRAPH g1
USE GRAPH g1
```

Add few triples, person A is calling person B

```
CREATE (Person:john)-[CALLS]->(Person:mike)
CREATE (Person:mike)-[CALLS]->(Person:john)
CREATE (Person:abbie)-[CALLS]->(Person:polly)
CREATE (Person:polly)-[CALLS]->(Person:john)
```

Here as we see, john calls mike and then mike calls john. It will be a complicated query to figure out all those who called each other. If we simply run following, then it will return results which will not be desirable

```
bangdb> S=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*)
+------------+-----+------------------------+
|sub         |pred |         obj            |
+------------+-----+------------------------+
|Person:john |CALLS| Person:mike |
+------------+-----+------------------------+
|Person:abbie|CALLS|Person:polly |
+------------+-----+------------------------+
|Person:polly|CALLS| Person:john  |
+------------+-----+------------------------+
|Person:mike |CALLS| Person:john |
+------------+-----+------------------------+
```

As we see, we get all persons calling other.
Let's modify it to ensure that we have people calling each other. But how do we do that?
We somehow need to know the Person1 calling Person2, then connect Person2 to Person1 with the same "CALLS" relation. This is not easy as we wish to know the list for all the persons (note Person:*). The query can become really complicated and it may not even be feasible with single query.

But if we look closely, we are talking about a relationship which are symmetrical to each other. Hence it will be good if we can somehow tell that in the query.

"SYMM" keyword does exactly that. Look at the query and how it simplifies things for us.

```
bangdb> S{SYMM}=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*)
+-----------+-----+-----------+
|sub        |pred |        obj|
+-----------+-----+-----------+
|Person:john|CALLS|Person:mike|
+-----------+-----+-----------+
|Person:mike|CALLS|Person:john|
+-----------+-----+-----------+
```

This is quite powerful, and, in many cases, we need this, having "SYMM" solves the problem really elegantly

Let's modify the query a bit, now we wish to know all persons who called someone but didn't receive call from the people who the called.

This is quite opposite to "SYMM", which is "ASYMM", so let's use "ASYMM" here

```
bangdb> S{ASYMM}=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*)
+------------+-----+------------+
|sub         |pred |         obj|
+------------+-----+------------+
|Person:abbie|CALLS|Person:polly|
+------------+-----+------------+
|Person:polly|CALLS| Person:john|
+------------+-----+------------+
```

Now Add following triple.

```
CREATE (Person:polly)-[CALLS]->(Person:john)
```

Note that this is a repeated triple, it already exists but still a valid triple.

Now if we run following query, (polly, john) combination appears twice in the list of all the callers, since she has made two calls to john.

```
bangdb> s=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*); RETURN p1.name AS Caller p2.name AS Callee
+------+------+
|Caller|Callee|
+------+------+
|john  |mike  |
+------+------+
|abbie |polly |
+------+------+
|polly |john  |
+------+------+
|polly |john  |
+------+------+
|mike  |john  |
+------+------+
```

But that's not what we wish to see, we want to know all the caller and callee set uniquely.

How do we enforce uniqueness for (sub, obj) pair?
We can use "UNIQUE" keyword here for this, it does exactly same

```
bangdb> s{UNIQUE}=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*); RETURN p1.name AS Caller p2.name AS Callee
+------+------+
|Caller|Callee|
+------+------+
|john  |mike  |
+------+------+
|abbie |polly |
+------+------+
|polly |john  |
+------+------+
|mike  |john  |
+------+------+
```

This is good. Now what if we wish to know all unique callers or callee, not at the pair level (i.e., uniqueness at (sub,obj)), but simply uniqueness based on what we select in the query (or RETURN from the query). This uniqueness is very contextual, let's see what this means.

Let's add another triple where polly again calls someone but this time a different person, let's say travis.

```
CREATE (Person:polly)-[CALLS]->(Person:travis)
```

Now if we run the previous query again and let's observe the result

```
bangdb> s{UNIQUE}=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*); RETURN p1.name AS Caller p2.name AS Callee
+------+------+
|Caller|Callee|
+------+------+
|john  |mike  |
+------+------+
|abbie |polly |
+------+------+
|polly |john  |
+------+------+
|polly |travis|
+------+------+
|mike  |john  |
+------+------+
```

Here, the sets (polly, john) and (polly, travis) shows up as they are unique as a pair. But what if we just want to

see caller or callee. Look at the following query

```
bangdb> s{UNIQUE}=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*); RETURN p1.name AS Caller
+------+
|Caller|
+------+
|john  |
+------+
|abbie |
+------+
|polly |
+------+
|polly |
+------+
|mike  |
+------+
```

Here we see Polly appearing twice. Although this is correct from the point of view that Polly calls two different persons, so these calls are unique, but here in this query we just want to see unique callers (not the unique calls).

Therefore, there is a context here within the query which is the uniqueness is needed for the given attribute which is being returned from the query. To do this, we will use uniqueness in context, which is UNIQUE_IN_CONTEXT

```
bangdb> s{UNIQUE_IN_CONTEXT}=>(@p1 Person:*)-[@c CALLS]->(@p2 Person:*); RETURN p1.name AS Caller
+------+
|Caller|
+------+
|john  |
+------+
|abbie |
+------+
|polly |
+------+
|mike  |
+------+
```

Now we get the right result (in the context) what we were looking for.

14. Aggregation

The Graph in BangDB supports following aggregations.

COUNT	counting, this can be used for numerical or categorical/string attributes
UCOUNT	unique counting, this can be used for numerical or categorical/string attributes
AVG	average for numerical type
MIN	minimum for numerical type
MAX	maximum for numerical type
STD	standard deviation for numerical type
SUM	sum/total for numerical type
EXKURT	kurtosis for numerical type
SKEW	skewness for numerical type

But we often want to aggregate something group by something else. For example, average sales per region, max number of products bought per category, number of unique visitors per page of a website etc. There could also be more than one group by variables, for example, total number of phones purchased in a period of time group by model and color, etc.

The Cypher in BangDB supports these queries in elegant way. We just need to tell what aggregation we wish to do on a particular attribute and then select another attribute in the same query which will be used as group by automatically. Users don't need to specifically define the group by attributes in the query.

Also, it's important to note that we can select as many different aggregates as we wish and different set of groupby keys (non-aggregate keys in the RETURN statement) as we need. Now it's time to look at few queries Let's create different graph and add few triples

CREATE GRAPH g2
USE GRAPH g2

CREATE (person:sachin)-[BUYS {"amount":211.45}]->(product:computer)
CREATE (person:sachin)-[BUYS {"amount":123.75}]->(product:computer)
CREATE (person:manu)-[BUYS {"amount":231.2}]->(product:hardware)
CREATE (person:ramesh)-[BUYS {"amount":345}]->(product:grocery)
CREATE (person:ramesh)-[BUYS {"amount":165.5}]->(product:electronics)

Let's count different persons and number of times they bought any product

```
bangdb> S=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer, COUNT(s.name) AS NUMT
+-------+--------+
|buyer |NUMT |
+-------+--------+
|manu  |1       |
+-------+--------+
|ramesh|2       |
+-------+--------+
|sachin|2       |
+-------+--------+
```

As you see, BangDB used buyer as group by attribute and counted the number of times they purchased any products

Let's count different persons and number of times they bought different unique products

```
bangdb> S=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer, COUNT(s.name) AS NUMT, UCOUNT(s.name)
AS unique_prods
+------+----+------------+
|buyer |NUMT|unique_prods|
+------+----+------------+
|manu  |1   |           1|
+------+----+------------+
|ramesh|2   |           2|
+------+----+------------+
|sachin|2   |           1|
+------+----+------------+
```

Here, we have two aggregates, with single group by.

Let's now also see average sales values (price) for each person along with other values that we are retrieving

```
bangdb> S=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer, COUNT(s.name) AS NUMT, UCOUNT(s.name)
AS unique_prods, AVG(b.amount) AS avg_price
+------+----+------------+----------+
|buyer |NUMT|unique_prods|avg_price |
+------+----+------------+----------+
|manu  |1   |           1|231.200000|
+------+----+------------+----------+
|ramesh|2   |           2|255.250000|
+------+----+------------+----------+
|sachin|2   |           1|167.600000|
+------+----+------------+----------+
```

Now, let's sort the result by avg_price and limit it to 2 rows only

```
bangdb> S=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer, COUNT(s.name) AS NUMT, UCOUNT(s.name)
AS unique_prods, AVG(b.amount) AS avg_price SORT_DESC avg_price LIMIT 2
+------+----+------------+----------+
|buyer |NUMT|unique_prods|avg_price |
+------+----+------------+----------+
|ramesh|2   |           2|255.250000|
```

```
+------+----+-----------+----------+
|manu  |1   |          1|231.200000|
+------+----+-----------+----------+
```

This is as simple as that. Let's now see standard deviation, min, max and sum as well

```
bangdb> S=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer, COUNT(s.name) AS NUMT, UCOUNT(s.name) AS unique_prods, AVG(b.amount) AS avg_sales, MIN(b.amount) AS MIN_Price MAX(b.amount) AS MAX_PRICE STD(b.amount) AS STD_DEV_AMOUNT SUM(b.amount) AS SUM_TOTAL
+------+----+-----------+----------+----------+-------------+----------+----------+
|buyer |NUMT|unique_prods|avg_sales |SUM_TOTAL |STD_DEV_AMOUNT|MAX_PRICE |MIN_Price |
+------+----+-----------+----------+----------+-------------+----------+----------+
|manu  |1   |          1|231.200000|231.200000|0            |231.200000|231.200000|
+------+----+-----------+----------+----------+-------------+----------+----------+
|ramesh|2   |          2|255.250000|510.500000|126.925667   |345       |165.500000|
+------+----+-----------+----------+----------+-------------+----------+----------+
|sachin|2   |          1|167.600000|335.200000|62.013265    |211.450000|123.750000|
+------+----+-----------+----------+----------+-------------+----------+----------+
```

Now let's say we wish to see unique results for buyer and product he/she bought

```
bangdb> S=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer s.name AS product
+----------+------+
|product   |buyer |
+----------+------+
|hardware  |manu  |
+----------+------+
|computer  |sachin|
+----------+------+
|computer  |sachin|
+----------+------+
|grocery   |ramesh|
+----------+------+
|electronics|ramesh|
+----------+------+
```

Here we see that the pair sachin and computer are repeating, which is true since sachin bought computer twice, but we wish to see unique pair and for this we will use UNIQUE keyword

```
bangdb> S{UNIQUE}=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer s.name AS product
+----------+------+
|product   |buyer |
+----------+------+
|hardware  |manu  |
+----------+------+
|computer  |sachin|
+----------+------+
|grocery   |ramesh|
+----------+------+
|electronics|ramesh|
+----------+------+
```

It gives us now the desired results.
Now let's list unique buyers.

```
bangdb> S{UNIQUE}=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer
+------+
|buyer |
```

```
+------+
|manu  |
+------+
|sachin|
+------+
|ramesh|
+------+
|ramesh|
+------+
```

Here we were expecting only unique names, but we see “ramesh” repeated twice. This is because the “UNIQUE” keyword enforces uniqueness of (sub, obj) pair and ramesh bought two different products hence they are unique together.

In order to see unique list based on what we wish to RETURN then we should use “UNIQUE_IN_CONTEXT” since we want query to simply return unique values based on what we return (or select)

Let’s see the result when the same query run with UNIQUE_IN_CONTEXT

```
bangdb> S{UNIQUE_IN_CONTEXT}=>(@p person:*)-[@b BUYS]->(@s product:*); RETURN p.name AS buyer
+------+
|buyer |
+------+
|manu  |
+------+
|sachin|
+------+
|ramesh|
+------+
```

15. SHORT_PATH (advance)

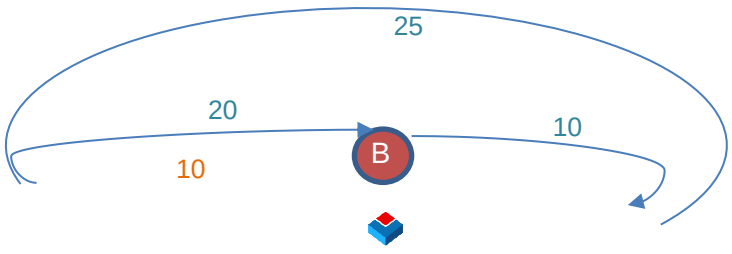
One of the central property or the very reason for the existence of any graph is to be able to efficiently find connection between any given pair of nodes and in many cases also finding the shortest path. BangDB provides very efficient way to achieve this in different scenarios.

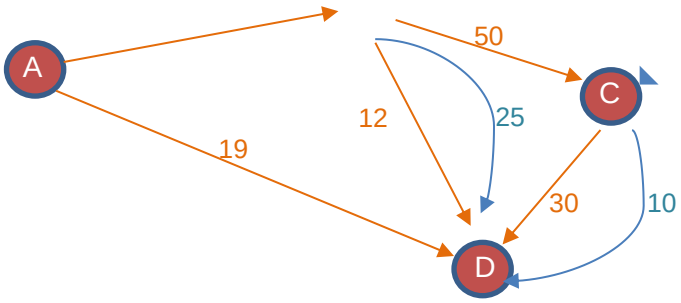
Find the shortest path between given two nodes, without worrying about relationships in between. This means we are just interested in finding the shortest path

CREATE GRAPH g4
USE GRAPH g4

```
CREATE (city:A {"lat":12.97,"lon":77.59})-[BYROAD {"distance":100}]->(city:B {"lat":12.29,"lon":76.64})
CREATE (city:B {"lat":12.29,"lon":76.64})-[BYROAD {"distance":50}]->(city:C {"lat":11.41,"lon":76.69})
CREATE (city:B {"lat":12.29,"lon":76.64})-[BYROAD {"distance":120}]->(city:D {"lat":13.34,"lon":76.11})
CREATE (city:A {"lat":12.97,"lon":77.59})-[BYROAD {"distance":190}]->(city:D  {"lat":13.34,"lon":76.11})
CREATE (city:C {"lat":11.41,"lon":76.69})-[BYROAD {"distance":30}]->(city:D  {"lat":13.34,"lon":76.11})
CREATE (city:A)-[BYAIR {"distance":20}]->(city:B)
CREATE (city:B)-[BYAIR {"distance":10}]->(city:C)
CREATE (city:C)-[BYAIR {"distance":10}]->(city:D)
CREATE (city:B)-[BYAIR {"distance":25}]->(city:D)
CREATE (city:A)-[BYAIR {"distance":25}]->(city:C)
```

Let’s assume that we have four cities, A,B,C and D. They are connected via road and air. Distances by road (in green) and by air (in blue) are shown in the graph below. Also, the latitude and longitude pairs are provided for all the cities.





We would like to know the shortest distance between city A and D, by road, air and finally by latitude and longitude based.

Let's write the query now.

First let's find out by road without the weight (without considering the distance)

```
bangdb> S=>(city:A)-[#SHORT_PATH *]->(city:D)
{
  "nodes" : [
    {
      "name" : "A",
      "label" : "city"
    },
    {
      "label" : "city",
      "name" : "D"
    }
  ],
  "edges" : [
    {
      "rel" : "BYROAD"
    }
  ],
  "path_count" : 1
}
```

As we see, without considering the distance, it picks the shortest path means of number of connections in between.

Now, let's find out the shortest path by road, so we will use distance here

```
bangdb> S=>(city:A)-[#SHORT_PATH BYROAD {weight="distance"}]->(city:D)
{
  "nodes" : [
    {
      "name" : "A",
      "label" : "city"
    },
    {
      "name" : "B",
      "label" : "city"
    },
    {
      "name" : "C",
      "label" : "city"
    },
    {
      "label" : "city",
      "name" : "D"
    }
  ],
  "path_count" : 1,
  "edges" : [
```

```
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYROAD"
      }
    ]
}
```

Now, let's run the query for shortest distance by AIR

```
bangdb> S=>(city:A)-[#SHORT_PATH BYAIR {weight="distance"}]->(city:D)
{
  "edges" : [
    {
      "rel" : "BYAIR"
    },
    {
      "rel" : "BYAIR"
    }
  ],
  "nodes" : [
    {
      "label" : "city",
      "name" : "A"
    },
    {
      "name" : "C",
      "label" : "city"
    },
    {
      "label" : "city",
      "name" : "D"
    }
  ],
  "path_count" : 1
}
```

If we run the query to find shortest distance by any means (Road or Air), we will get the right results in this case too
Notice the '*' after #SHORT_PATH to indicate any relation

```
bangdb> S=>(city:A)-[#SHORT_PATH * {weight="distance"}]->(city:D)
{
  "path_count" : 1,
  "nodes" : [
    {
      "label" : "city",
      "name" : "A"
    },
    {
      "label" : "city",
      "name" : "C"
    },
    {
      "name" : "D",
      "label" : "city"
    }
  ],
  "edges" : [
    {
      "rel" : "BYAIR"
    },
    {
      "rel" : "BYAIR"
    }
  ]
}
```

Finally, let's use latitude and longitude to compute the distance

```
bangdb> S=>(city:A)-[#SHORT_PATH * {geo_distance="lat.lon"}]->(city:D)
{
  "path_count" : 1,
  "nodes" : [
    {
      "label" : "city",
      "name" : "A"
    },
    {
      "label" : "city",
      "name" : "D"
    }
  ],
  "edges" : [
    {
      "rel" : "BYROAD"
    }
  ]
}
```

16. ALL_PATH (advance)

As SHORT_PATH, we can find out all paths between two nodes, using any or specific relationship

Let's find all paths between A and D. As you see, there are 15 possible paths between A and D. We should be careful while running ALL_PATH query, as it could result into costly computation

```
bangdb> S=>(city:A)-[#ALL_PATH *]->(city:D)
{
  "path_count" : 15,
  "paths" : [
    {
      "edges" : [
        {
          "rel" : "BYROAD"
        },
        {
          "rel" : "BYROAD"
        }
      ],
      "nodes" : [
        {
          "name" : "A",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "B"
        },
        {
          "name" : "D",
          "label" : "city"
        }
      ]
    },
    {
      "nodes" : [
        {
          "label" : "city",
          "name" : "A"
        },
        {
          "name" : "B",
          "label" : "city"
        },
        {
          "name" : "C",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "D"
        }
      ],
```

```
    "edges" : [
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYROAD"
      }
    ]
  },
  {
    "nodes" : [
      {
        "name" : "A",
        "label" : "city"
      },
      {
        "label" : "city",
        "name" : "B"
      },
      {
        "name" : "C",
        "label" : "city"
      },
      {
        "name" : "D",
        "label" : "city"
      }
    ],
    "edges" : [
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYAIR"
      }
    ]
  },
  {
    "edges" : [
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYAIR"
      }
    ],
    "nodes" : [
      {
        "name" : "A",
        "label" : "city"
      },
      {
        "name" : "B",
        "label" : "city"
      },
      {
        "label" : "city",
        "name" : "D"
      }
    ]
  },
  {
    "edges" : [
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYAIR"
      },
      {
        "rel" : "BYROAD"
      }
    ],
    "nodes" : [
      {
```

```
          "name" : "A",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "B"
        },
        {
          "label" : "city",
          "name" : "C"
        },
        {
          "name" : "D",
          "label" : "city"
        }
      ]
    },
    {
      "edges" : [
        {
          "rel" : "BYROAD"
        },
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYAIR"
        }
      ],
      "nodes" : [
        {
          "name" : "A",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "B"
        },
        {
          "label" : "city",
          "name" : "C"
        },
        {
          "label" : "city",
          "name" : "D"
        }
      ]
    },
    {
      "nodes" : [
        {
          "label" : "city",
          "name" : "A"
        },
        {
          "label" : "city",
          "name" : "D"
        }
      ],
      "edges" : [
        {
          "rel" : "BYROAD"
        }
      ]
    },
    {
      "nodes" : [
        {
          "label" : "city",
          "name" : "A"
        },
        {
          "label" : "city",
          "name" : "B"
        },
        {
          "name" : "D",
          "label" : "city"
        }
      ],
      "edges" : [
```

```
      {
        "rel" : "BYAIR"
      },
      {
        "rel" : "BYROAD"
      }
    ]
  },
  {
    "edges" : [
      {
        "rel" : "BYAIR"
      },
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYROAD"
      }
    ],
    "nodes" : [
      {
        "name" : "A",
        "label" : "city"
      },
      {
        "label" : "city",
        "name" : "B"
      },
      {
        "label" : "city",
        "name" : "C"
      },
      {
        "name" : "D",
        "label" : "city"
      }
    ]
  },
  {
    "nodes" : [
      {
        "name" : "A",
        "label" : "city"
      },
      {
        "label" : "city",
        "name" : "B"
      },
      {
        "label" : "city",
        "name" : "C"
      },
      {
        "label" : "city",
        "name" : "D"
      }
    ],
    "edges" : [
      {
        "rel" : "BYAIR"
      },
      {
        "rel" : "BYROAD"
      },
      {
        "rel" : "BYAIR"
      }
    ]
  },
  {
    "edges" : [
      {
        "rel" : "BYAIR"
      },
      {
        "rel" : "BYAIR"
      }
    ],
    "nodes" : [
      {
```

```
          "name" : "A",
          "label" : "city"
        },
        {
          "name" : "B",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "D"
        }
      ]
    },
    {
      "nodes" : [
        {
          "name" : "A",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "B"
        },
        {
          "name" : "C",
          "label" : "city"
        },
        {
          "name" : "D",
          "label" : "city"
        }
      ],
      "edges" : [
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYROAD"
        }
      ]
    },
    {
      "nodes" : [
        {
          "label" : "city",
          "name" : "A"
        },
        {
          "name" : "B",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "C"
        },
        {
          "name" : "D",
          "label" : "city"
        }
      ],
      "edges" : [
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYAIR"
        }
      ]
    },
    {
      "nodes" : [
        {
          "name" : "A",
          "label" : "city"
        },
```

```
        {
          "name" : "C",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "D"
        }
      ],
      "edges" : [
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYROAD"
        }
      ]
    },
    {
      "nodes" : [
        {
          "name" : "A",
          "label" : "city"
        },
        {
          "label" : "city",
          "name" : "C"
        },
        {
          "name" : "D",
          "label" : "city"
        }
      ],
      "edges" : [
        {
          "rel" : "BYAIR"
        },
        {
          "rel" : "BYAIR"
        }
      ]
    }
  ]
```


17. SIMILARITY

One of the major reasons for have graph structure in place is to be able to find the context and also entities & their relationships. If we have such arrangement in place, then based on the relationships we should be able to find the similarities between the entities. And if we know how close or far a given set of pair of entities are, certain actions could be taken for achieving some goals. For example, if we are able to find two persons' degree of similarities then based on this, we could offer products to one or the other person with higher likelihood of conversion. Recommendations, serving ads, offering deals and discounts, personalization etc. As we see there are many use cases that could be enabled with this concept if we are able to do this efficiently.

Let's compute the similarities of persons based on their buying pattern here in this example

```
CREATE GRAPH g3
USE GRAPH g3
```

```
CREATE (person:dan)-[BUYS {"amount": 1.2}]->(product:cookies)
CREATE (person:dan)-[BUYS {"amount": 3.2}]->( product:milk)
CREATE (person:dan)-[BUYS {"amount": 2.2}]->( product:chocolate)

CREATE (person:annie)-[BUYS {"amount": 1.2}]->( product:cucumber)
CREATE (person:annie)-[BUYS {"amount": 3.2}]->( product:milk)
CREATE (person:annie)-[BUYS {"amount": 3.2}]->( product:tomatoes)

CREATE (person:matt)-[BUYS {"amount": 3}]->( product:tomatoes)
CREATE (person:matt)-[BUYS {"amount": 2}]->( product:kale)
```

```
CREATE (person:matt)-[BUYS {"amount": 1}]->( product:cucumber)

CREATE (person:jeff)-[BUYS {"amount": 3}]->( product:cookies)
CREATE (person:jeff)-[BUYS {"amount": 2}]->( product:milk)

CREATE (person:brie)-[BUYS {"amount": 1}]->( product:tomatoes)
CREATE (person:brie)-[BUYS {"amount": 2}]->( product:milk)
CREATE (person:brie)-[BUYS {"amount": 2}]->( product:kale)
CREATE (person:brie)-[BUYS {"amount": 3}]->( product:cucumber)
CREATE (person:brie)-[BUYS {"amount": 0.3}]->( product:celery)

CREATE (person:elsa)-[BUYS {"amount": 3}]->( product:chocolate)
CREATE (person:elsa)-[BUYS {"amount": 3}]->( product:milk)

CREATE (person:john)-[BUYS {"amount": 5}]->( product:kale)
CREATE (person:john)-[BUYS {"amount": 2}]->( product:peanut)

CREATE (person:steve)-[BUYS {"amount": 7}]->( product:orange)
CREATE (person:steve)-[BUYS {"amount": 3}]->( product:mango)
```

Let's run the query for computations of similarity
Note: The constraint for running this query is that you must RETURN the label, for which you wish to compute the similarity in the query, having exactly the same name. In the example below, we return p.name AS person, note that label is also "person"

```
bangdb> S{SIMILARITY}=>(@p person:*)-[@b BUYS]->(@c product:*); RETURN p.name AS person b.amount AS amount c.name AS product

{
  "errcode" : 0,
  "msg" : [
    "Successfully computed the similarity and updated the relations (use '_SIMILAR_' relation to retrieve the number)"
  ]
}
```

The above query used the graph data and implicitly trained KMEANS model to figure out several centroids. We can define how many such centroids to train for by providing the number along with SIMILARITY key. For example, to train for 10 centroids we may use following query

```
S{SIMILARITY, 10}=>(@p person:*)-[@b BUYS]->(@c product:*); RETURN p.name AS person b.amount AS amount c.name AS product
```

The query also updates the graph by adding a relationship _SIMILAR_ (by default) or by the name as provided in the query. We can provide the name of the relationship as follows.

```
S{SIMILARITY(any_rel_name), 10}
```

Now, we can query and see the result using the relation's name that we provided or default (_SIMILAR_) if we didn't provide. In this case we didn't provide the similarity relation name hence we will use _SIMILAR_

```
bangdb> S=>(@p person:*)-[@b _SIMILAR_]->(@c person:*); RETURN p.name AS person1 c.name AS person2 b.similarity AS similarity
+-------+----------+-------+
|person2|similarity|person1|
+-------+----------+-------+
|john   |0.915942  |  annie|
+-------+----------+-------+
|brie   |0.921728  |  annie|
+-------+----------+-------+
|john   |0.942952  |   brie|
```

```
+-------+----------+-------+
|annie  |0.913604  |   jeff|
+-------+----------+-------+
|john   |0.990302  |   jeff|
+-------+----------+-------+
|brie   |0.950601  |   jeff|
+-------+----------+-------+
|annie  |0.903701  |    dan|
+-------+----------+-------+
|john   |0.927200  |    dan|
+-------+----------+-------+
|brie   |0.918632  |    dan|
+-------+----------+-------+
|jeff   |0.936024  |    dan|
+-------+----------+-------+
|matt   |0.921103  |    dan|
+-------+----------+-------+
|elsa   |0.934002  |    dan|
+-------+----------+-------+
|annie  |0.944215  |   matt|
+-------+----------+-------+
|john   |0.970581  |   matt|
+-------+----------+-------+
|brie   |0.984370  |   matt|
+-------+----------+-------+
|jeff   |0.975808  |   matt|
+-------+----------+-------+
|annie  |0.956274  |   elsa|
+-------+----------+-------+
|john   |0.941132  |   elsa|
+-------+----------+-------+
|brie   |0.886888  |   elsa|
+-------+----------+-------+
|jeff   |0.944975  |   elsa|
+-------+----------+-------+
|matt   |0.914367  |   elsa|
+-------+----------+-------+
|annie  |0.757797  |  steve|
+-------+----------+-------+
|john   |0.866963  |  steve|
+-------+----------+-------+
|brie   |0.765535  |  steve|
+-------+----------+-------+
|jeff   |0.832822  |  steve|
+-------+----------+-------+
|dan    |0.894982  |  steve|
+-------+----------+-------+
|matt   |0.776241  |  steve|
+-------+----------+-------+
|elsa   |0.852883  |  steve|
+-------+----------+-------+
```

Or, to view only the results where score is greater than 0.95

```
bangdb> S=>(@p person:*)-[@b _SIMILAR_]->(@c person:*); RETURN p.name AS person1 c.name AS person2 b.similarity AS
similarity WHERE similarity > 0.95
+-------+----------+-------+
|person2|similarity|person1|
+-------+----------+-------+
|john   |0.990302  |   jeff|
+-------+----------+-------+
|brie   |0.950601  |   jeff|
+-------+----------+-------+
|john   |0.970581  |   matt|
+-------+----------+-------+
|brie   |0.984370  |   matt|
+-------+----------+-------+
|jeff   |0.975808  |   matt|
```

```
+-------+----------+-------+
|annie  |0.956274  |   elsa|
+-------+----------+-------+
```

Note, if you don't wish to update the graph by adding _SIMILAR_ relationships between the nodes with similarity score then you can use the keyword "SIMILARITY_TEST". And in this case BangDB won't update the graphs instead it will compute the scores and return to the user in a structured manner

```
bangdb> S{SIMILARITY_TEST}=>(@p person:*)-[@b BUYS]->(@c product:*); RETURN p.name AS person b.amount AS amount
c.name AS product
{
  "pairs" : [
    {
      "similarity" : 0.93547131724717,
      "A" : "dan",
      "B" : "elsa"
    },
    {
      "A" : "dan",
      "B" : "matt",
      "similarity" : 0.923687462697141
    },
    {
      "B" : "jeff",
      "A" : "dan",
      "similarity" : 0.98732715961013
    },
    {
      "similarity" : 0.905641110794971,
      "A" : "dan",
      "B" : "annie"
    },
    {
      "similarity" : 0.905918283036807,
      "A" : "dan",
      "B" : "brie"
    },
    {
      "similarity" : 0.961483504509432,
      "A" : "elsa",
      "B" : "matt"
    },
    {
      "A" : "elsa",
      "B" : "jeff",
      "similarity" : 0.942420408659908
    },
    {
      "similarity" : 0.957425185618457,
      "B" : "annie",
      "A" : "elsa"
    },
    {
      "B" : "brie",
      "A" : "elsa",
      "similarity" : 0.872247612472379
    },
    {
      "similarity" : 0.928872132753044,
      "A" : "matt",
      "B" : "jeff"
    },
    {
      "similarity" : 0.991039526450287,
      "A" : "matt",
      "B" : "annie"
    },
    {
      "B" : "brie",
```

```
      "A" : "matt",
      "similarity" : 0.935316150022884
    },
    {
      "A" : "jeff",
      "B" : "annie",
      "similarity" : 0.913827151108844
    },
    {
      "similarity" : 0.89344184143706,
      "B" : "brie",
      "A" : "jeff"
    },
    {
      "similarity" : 0.921594885722905,
      "A" : "annie",
      "B" : "brie"
    }
  ],
  "label" : "person"
}
```


18. CLUSTER

One of the major goals for Graph is be able to find natural groups or subgroups that may exist within the database. There could be plenty of groups based on different set of properties of the nodes. For example, we would like to find out different groups of people based on their demography, backgrounds, activities, family structure, buying patterns, interests, hobbies and many more attributes that could be used. The Graph should give us ability to not only find such groups but also to take actions as required such as finding other people in a given group or given a person. Further we should be able to find the common patterns and based on recommend something to a person from the same group or personalize the content or experiences. These groups are dynamic hence we should be able to create many clusters, whenever we wish and using as many different set of features as we need

Let's take an example of product recommendation to the users based on their various properties, such as age, education, marital status, number of children etc… Note that we could use totally different set of features for recommending different products or services or for personalization.

Therefore, we will do two things here;

- Train clusters based on set of features
- Recommend products to a given user based on the clusters (by looking at the most common products bought by the people from same cluster, which are not bought by the user yet)

First, let's put some data, here we will be adding some users' data and their purchase data

```
CREATE GRAPH g4
USE GRAPH g4

CREATE (person:dan {"age":34,"married":1,"num_child":2,"edu":"grad","income":100000,"job":"self"})
CREATE (person:john {"age":21,"married":0,"num_child":0,"edu":"grad","income":30000,"job":"employed"})
CREATE (person:ramesh {"age":24,"married":1,"num_child":0,"edu":"grad","income":20000,"job":"employed"})
CREATE (person:suman {"age":44,"married":1,"num_child":3,"edu":"postgrad","income":200000,"job":"employed"})
CREATE (person:mahesh {"age":32,"married":1,"num_child":0,"edu":"grad","income":90000,"job":"employed"})
CREATE (person:mangesh {"age":38,"married":1,"num_child":1,"edu":"phd","income":190000,"job":"employed"})
CREATE (person:prasad {"age":22,"married":1,"num_child":0,"edu":"grad","income":32000,"job":"employed"})
```

```
CREATE (person:sanjay {"age":33,"married":1,"num_child":1,"edu":"grad","income":90000,"job":"self"})
CREATE (person:ankit {"age":54,"married":1,"num_child":2,"edu":"grad","income":320000,"job":"self"})
CREATE (person:ashoke {"age":67,"married":1,"num_child":3,"edu":"grad","income":10000,"job":"none"})
CREATE (person:goyal {"age":71,"married":1,"num_child":2,"edu":"grad","income":12000,"job":"none"})
CREATE (person:mike {"age":62,"married":1,"num_child":4,"edu":"ugrad","income":5000,"job":"none"})
CREATE (person:patrick {"age":37,"married":0,"num_child":0,"edu":"ugrad","income":80000,"job":"employed"})
CREATE (person:ballu {"age":34,"married":1,"num_child":2,"edu":"grad","income":100000,"job":"self"})
CREATE (person:eric {"age":28,"married":0,"num_child":0,"edu":"ugrad","income":30000,"job":"self"})
CREATE (person:susan {"age":28,"married":1,"num_child":2,"edu":"grad","income":24500,"job":"employed"})
CREATE (person:kavita {"age":43,"married":1,"num_child":4,"edu":"postgrad","income":160000,"job":"employed"})
CREATE (person:teresa {"age":31,"married":1,"num_child":0,"edu":"grad","income":70000,"job":"employed"})
CREATE (person:bob {"age":35,"married":1,"num_child":2,"edu":"phd","income":145000,"job":"employed"})
CREATE (person:bill {"age":34,"married":1,"num_child":20,"edu":"grad","income":110000,"job":"employed"})
CREATE (person:steve {"age":31,"married":1,"num_child":1,"edu":"grad","income":120000,"job":"self"})
CREATE (person:marie {"age":68,"married":1,"num_child":2,"edu":"grad","income":2000,"job":"none"})
CREATE (person:shiela {"age":78,"married":1,"num_child":4,"edu":"postgrad","income":500000,"job":"self"})
CREATE (person:mahender {"age":67,"married":1,"num_child":4,"edu":"grad","income":10000,"job":"none"})

CREATE (product:milk {"type":"beverage","price":180,"usage":1})
CREATE (product:eggs {"type":"food","price":135,"usage":1})
CREATE (product:wine {"type":"beverage","price":242,"usage":3})
CREATE (product:meat {"type":"food","price":100,"usage":2})
CREATE (product:chips {"type":"snack","price":100,"usage":3})
CREATE (product:cigar {"type":"tobacco","price":102,"usage":3})
CREATE (product:water {"type":"beverage","price":100,"usage":1})
CREATE (product:noodle {"type":"food","price":100,"usage":2})
CREATE (product:coffee {"type":"beverage","price":100,"usage":2})
CREATE (product:tea {"type":"beverage","price":135,"usage":2})
CREATE (product:pencil {"type":"stationery","price":96,"usage":1})
CREATE (product:pen {"type":"stationery","price":100,"usage":1})

CREATE (person:dan)-[BUYS {"price":178.22}]->(product:milk)
CREATE (person:dan)-[BUYS {"price":130.21}]->(product:eggs)
CREATE (person:dan)-[BUYS {"price":233}]->(product:wine)
CREATE (person:john)-[BUYS {"price":78.23}]->(product:milk)
CREATE (person:john)-[BUYS {"price":98.09}]->(product:meat)
CREATE (person:john)-[BUYS {"price":100.20}]->(product:chips)
CREATE (person:ramesh)-[BUYS {"price":98.09}]->(product:meat)
CREATE (person:ramesh)-[BUYS {"price":100.20}]->(product:cigar)
CREATE (person:suman)-[BUYS {"price":100.20}]->(product:milk)
CREATE (person:suman)-[BUYS {"price":98.09}]->(product:meat)
CREATE (person:suman)-[BUYS {"price":100.20}]->(product:water)
CREATE (person:mahesh)-[BUYS {"price":100.20}]->(product:noodle)
CREATE (person:mahesh)-[BUYS {"price":98.09}]->(product:eggs)
CREATE (person:mahesh)-[BUYS {"price":100.20}]->(product:cigar)
CREATE (person:mangesh)-[BUYS {"price":98.09}]->(product:meat)
CREATE (person:mangesh)-[BUYS {"price":100.20}]->(product:water)
CREATE (person:mahesh)-[BUYS {"price":100.20}]->(product:chips)
CREATE (person:prasad)-[BUYS {"price":98.09}]->(product:coffee)
CREATE (person:prasad)-[BUYS {"price":134.23}]->(product:tea)
CREATE (person:sanjay)-[BUYS {"price":104.21}]->(product:milk)
CREATE (person:sanjay)-[BUYS {"price":95.49}]->(product:pencil)
CREATE (person:sanjay)-[BUYS {"price":103.23}]->(product:tea)
CREATE (person:ankit)-[BUYS {"price":92.39}]->(product:meat)
CREATE (person:ankit)-[BUYS {"price":103.27}]->(product:tea)
CREATE (person:ashoke)-[BUYS {"price":132.23}]->(product:chips)
CREATE (person:ashoke)-[BUYS {"price":93.39}]->(product:pen)
CREATE (person:ashoke)-[BUYS {"price":130.23}]->(product:tea)
CREATE (person:goyal)-[BUYS {"price":109.54}]->(product:meat)
CREATE (person:goyal)-[BUYS {"price":105.24}]->(product:coffee)
CREATE (person:goyal)-[BUYS {"price":198.39}]->(product:milk)
CREATE (person:goyal)-[BUYS {"price":120.27}]->(product:pencil)
CREATE (person:mike)-[BUYS {"price":109.54}]->(product:chips)
CREATE (person:mike)-[BUYS {"price":105.24}]->(product:coffee)
CREATE (person:mike)-[BUYS {"price":198.39}]->(product:milk)
CREATE (person:mike)-[BUYS {"price":120.27}]->(product:tea)
CREATE (person:patrick)-[BUYS {"price":125.34}]->(product:coffee)
CREATE (person:patrick)-[BUYS {"price":194.32}]->(product:milk)
CREATE (person:patrick)-[BUYS {"price":121.22}]->(product:tea)
CREATE (person:ballu)-[BUYS {"price":121.32}]->(product:chips)
CREATE (person:ballu)-[BUYS {"price":191.21}]->(product:pencil)
CREATE (person:ballu)-[BUYS {"price":122.45}]->(product:tea)
CREATE (person:eric)-[BUYS {"price":133.89}]->(product:meat)
CREATE (person:eric)-[BUYS {"price":156.36}]->(product:chips)
CREATE (person:eric)-[BUYS {"price":134.56}]->(product:pen)
CREATE (person:eric)-[BUYS {"price":152.55}]->(product:tea)
CREATE (person:susan)-[BUYS {"price":142.56}]->(product:meat)
CREATE (person:susan)-[BUYS {"price":121.43}]->(product:pencil)
CREATE (person:susan)-[BUYS {"price":146.57}]->(product:tea)
```

```
CREATE (person:kavita)-[BUYS {"price":123.63}]->(product:coffee)
CREATE (person:kavita)-[BUYS {"price":124.32}]->(product:tea)
CREATE (person:kavita)-[BUYS {"price":148.77}]->(product:pen)
CREATE (person:teresa)-[BUYS {"price":184.86}]->(product:pencil)
CREATE (person:bob)-[BUYS {"price":164.39}]->(product:milk)
CREATE (person:bob)-[BUYS {"price":141.97}]->(product:pencil)
CREATE (person:bill)-[BUYS {"price":104.61}]->(product:pencil)
CREATE (person:bill)-[BUYS {"price":162.32}]->(product:milk)
CREATE (person:bill)-[BUYS {"price":101.57}]->(product:pen)
CREATE (person:steve)-[BUYS {"price":112.34}]->(product:meat)
CREATE (person:steve)-[BUYS {"price":131.72}]->(product:pencil)
CREATE (person:marie)-[BUYS {"price":112.74}]->(product:milk)
CREATE (person:marie)-[BUYS {"price":116.48}]->(product:meat)
CREATE (person:marie)-[BUYS {"price":135.72}]->(product:pen)
CREATE (person:shiela)-[BUYS {"price":126.42}]->(product:milk)
CREATE (person:shiela)-[BUYS {"price":135.76}]->(product:tea)
CREATE (person:marie)-[BUYS {"price":135.72}]->(product:pen)
CREATE (person:mahender)-[BUYS {"price":146.96}]->(product:meat)
CREATE (person:mahender)-[BUYS {"price":135.36}]->(product:coffee)
```

Let's train the cluster now

```
bangdb> S{CLUSTER(user_cluster),3}=>(@p person:*); RETURN p.name AS person p.age AS age p.married AS married p.num_child
AS num_child p.edu AS edu p.income AS income p.job AS job
{
  "msg" : [
    "successfully computed the clusters and updated the relations (use 'user_cluster' relation to retrieve the number)"
  ],
  "errcode" : 0
}
```

We can now single query to get the recommended products based on this cluster. But before that let's also run some interesting queries which most often might be needed

See all those who are in the 'user_cluster'

```
bangdb> S=>(@p person:*)-[@c user_cluster {_cluster_ = *}]->(@p2 person:*); RETURN p.name AS Name p.age AS age p.married AS
married p.num_child AS num_child p.income AS income p.job AS job
+--------+--------+------+---------+-------+---+
|Name    |job     |income|num_child|married|age|
+--------+--------+------+---------+-------+---+
|patrick |employed| 80000|0        |0      |37 |
+--------+--------+------+---------+-------+---+
|john    |employed| 30000|0        |0      |21 |
+--------+--------+------+---------+-------+---+
|ballu   |self    |100000|2        |1      |34 |
+--------+--------+------+---------+-------+---+
|mahesh  |employed| 90000|0        |1      |32 |
+--------+--------+------+---------+-------+---+
|ashoke  |none    | 10000|3        |1      |67 |
+--------+--------+------+---------+-------+---+
|eric    |self    | 30000|0        |0      |28 |
+--------+--------+------+---------+-------+---+
|goyal   |none    | 12000|2        |1      |71 |
+--------+--------+------+---------+-------+---+
|teresa  |employed| 70000|0        |1      |31 |
+--------+--------+------+---------+-------+---+
|dan     |self    |100000|2        |1      |34 |
+--------+--------+------+---------+-------+---+
|prasad  |employed| 32000|0        |1      |22 |
```

```
+--------+--------+------+---------+-------+---+
|susan   |employed| 24500|2        |1      |28 |
+--------+--------+------+---------+-------+---+
|ramesh  |employed| 20000|0        |1      |24 |
+--------+--------+------+---------+-------+---+
|steve   |self    |120000|1        |1      |31 |
+--------+--------+------+---------+-------+---+
|sanjay  |self    | 90000|1        |1      |33 |
+--------+--------+------+---------+-------+---+
|mahender|none    | 10000|4        |1      |67 |
+--------+--------+------+---------+-------+---+
|marie   |none    |  2000|2        |1      |68 |
+--------+--------+------+---------+-------+---+
|mike    |none    |  5000|4        |1      |62 |
+--------+--------+------+---------+-------+---+
```

Let's create product cluster as well

```
bangdb> S{CLUSTER(product_cluster),3}=>(@p product:*); RETURN p.name AS product p.type AS type p.price AS price p.usage AS
usage
{
  "errcode" : 0,
  "msg" : [
    "successfully computed the clusters and updated the relations (use 'product_cluster' relation to retrieve the number)"
  ]
}
```

Let's view all the products in the 'product_cluster'

```
bangdb> S=>(@p product:*)-[@c product_cluster {_cluster_= *}]->(@p2 product:*); RETURN p.name AS product p.type AS type p.price
AS price p.usage AS usage
+-------+-----+-----+--------+
|product|usage|price|type    |
+-------+-----+-----+--------+
|eggs   |1    |  135|food    |
+-------+-----+-----+--------+
|meat   |2    |  100|food    |
+-------+-----+-----+--------+
|noodle |2    |  100|food    |
+-------+-----+-----+--------+
|milk   |1    |  180|beverage|
+-------+-----+-----+--------+
|tea    |2    |  135|beverage|
+-------+-----+-----+--------+
|wine   |3    |  242|beverage|
+-------+-----+-----+--------+
|water  |1    |  100|beverage|
+-------+-----+-----+--------+
|coffee |2    |  100|beverage|
+-------+-----+-----+--------+
```

Now, let's work towards the recommendation of products to different users

Find out the cluster id for user 'shiela'

```
bangdb> S=>(@p person:shiela)-[@c user_cluster]->(@p2 person:shiela); RETURN c._cluster_ AS cid
+---+
|cid|
+---+
|2  |
+---+
```

Now, let's find out all the users belonging to this cluster

```
bangdb> S=>(@p person:*)-[@c user_cluster {_cluster_ = 2}]->(@p2 person:*); RETURN p.name AS Name
+-------+
|Name   |
```

```
+-------+
|suman  |
+-------+
|mangesh|
+-------+
|bob    |
+-------+
|kavita |
+-------+
|shiela |
+-------+
|ankit  |
+-------+
```

We can also do this without running two queries as above (first to figure out the cluster id and then getting all users from the same cluster). We can run following. Note the '$$' attached to person:shiela, this tells the BangDB to first find the cluster id of Shiela and then use this to find all people belonging to the same cluster

```
bangdb> S=>(@p person:*)-[@c user_cluster {_cluster_ = $$person:shiela}]->(@p2 person:*); RETURN p.name AS Name
+-------+
|Name   |
+-------+
|suman  |
+-------+
|mangesh|
+-------+
|bob    |
+-------+
|kavita |
+-------+
|shiela |
+-------+
|ankit  |
+-------+
```

Now, let's find top 5 most bought products by the people belonging to the cluster of Shiela, in the descending order of number of times they are bought

```
bangdb> S=>(@p person:*)-[@c user_cluster {_cluster_ = $$person:suman}]->(@p2 person:*)-[@b BUYS]->(@pr product:*); RETURN
pr.name AS product COUNT(p.name) AS num_times SORT_DESC num_times LIMIT 5
+-------+---------+
|product|num_times|
+-------+---------+
|milk   |3        |
+-------+---------+
|meat   |3        |
+-------+---------+
|tea    |3        |
+-------+---------+
|water  |2        |
+-------+---------+
|pen    |1        |
+-------+---------+
```

To recommend products to a given user, we wish to find the cluster he/she belongs to, then all the people in the cluster, then most common set of products that they bought. Finally we need to remove the products that the user (for whom the recommendation is being done) has bought. This will give us the list of items that should be recommended to the user.

All of the above computations can be written in single query and it will return the recommendation. Here is the query

```
bangdb> <SUBTRACT USING product SORT_DESC num_times LIMIT 3> S1=>(@p person:*)-[@c user_cluster {_cluster_ =
$$person:shiela}]->(@p2 person:*)-[@b BUYS]->(@pr product:*); RETURN pr.name AS product COUNT(p.name) AS num_times
SORT_DESC num_times ++ S2=>(@p person:suman)-[@b BUYS]->(@pr product:*); RETURN pr.name AS product COUNT(p.name)
AS num_times SORT_DESC num_times
+-------+---------+
|product|num_times|
+-------+---------+
|tea    |3        |
+-------+---------+
|pencil |1        |
+-------+---------+
|pen    |1        |
+-------+---------+
```

This single query is enough for such findings which are otherwise quite complex and sometimes involves lots of computations and workflows.

Here is what's happening in the query;
There are two sub queries here. First one is finding the most common set of products bought by the people belonging to the same group as the user. Second query is finding the products bought by the user. And then following command is asking BangDB to subtract the second resultset from the first one and return only top 3 recommended products in sorted order.

```
<SUBTRACT USING product SORT_DESC num_times LIMIT 3>
```

This concludes short introduction for the Graph in BangDB, however, please go to https://bangdb.com/developer for detail discussion and examples for the same

19. Geospatial Hashing

BangDB Graph supports Geospatial hashing-based indexing. Which means we can add latitude, longitude pairs for set of entities and then query for various entities within given Radius or Quadrants etc. For example, some of the queries related to "Near me …", "within radius of 2KM from a point or an entity …" etc. can be answered easily.

Let's see some of these examples in action.

Example 1: Finding persons near a given person, who is connected with others with some specific relation

```
CREATE GRAPH gh1
create geoindex gh1.Person latlon = lat.lon
create geoindex gh1.Company latlon = lat.lon

CREATE (Person:john {"lat":12.8025,"lon":77.4028})-[WORKS_AT]->(Company:Acme {"lat":12.8016,"lon":77.3937})
CREATE (Person:dan {"lat":12.8025,"lon":77.7599})-[WORKS_AT]->(Company:google {"lat":12.8444,"lon":77.5986})
CREATE (Person:mike {"lat":12.8146,"lon":77.6656})-[WORKS_AT]->(Company:Acme {"lat":12.8417,"lon":77.7577})
CREATE (Person:peter {"lat":12.8153,"lon":77.7010})-[WORKS_AT]->(Company:Acme {"lat":12.8543,"lon":77.5882})
CREATE (Person:susan {"lat":12.8187,"lon":77.4169})-[WORKS_AT]->(Company:Acme {"lat":12.8554,"lon":77.5265})
CREATE (Person:andrea {"lat":12.8235,"lon":77.5262})-[WORKS_AT]->(Company:Acme {"lat":12.8585,"lon":77.6706})
CREATE (Person:arti {"lat":12.8281,"lon":77.6464})-[WORKS_AT]->(Company:Acme {"lat":12.8610,"lon":77.5350})
CREATE (Person:teresa {"lat":12.8316,"lon":77.4727})-[WORKS_AT]->(Company:Acme {"lat":12.8696,"lon":77.5991})
```

```
S=>(@p Person:john)-[#RADIUS_OF 1000 WORKS_AT]->(@c Company:*)
+-----------+--------+------------+
|sub        |pred    |        obj|
+-----------+--------+------------+
|Person:john|WORKS_AT|Company:Acme|
+-----------+--------+------------+
```

```
S=>(@p Person:john)-[@r #RADIUS_OF 1000 WORKS_AT]->(@c Company:*); RETURN p.name AS person, r.rel AS works_at, c.name AS company
+-------+------+--------+
|company|person|works_at|
+-------+------+--------+
|Acme   |john  |WORKS_AT|
+-------+------+--------+
```

```
S=>(@p Person:john)-[#RADIUS_OF 1000 WORKS_AT]->(@c Company:*); RETURN COUNT(c.name) AS Num_companies
+-------------+
|Num_companies|
+-------------+
|1            |
+-------------+
```

Example 2: Entities with no relationships, and finding based on geo-loc solely

```
CREATE GRAPH gh2
create geoindex gh2.Person latlon = lat.lon
create geoindex gh2.Company latlon = lat.lon

CREATE (Person:john {"lat":12.8025,"lon":77.4028})
CREATE (Person:laila {"lat":12.8015,"lon":77.4022})
CREATE (Person:madlin {"lat":12.8021,"lon":77.4022})
CREATE (Person:rachel {"lat":12.8022,"lon":77.4023})
CREATE (Person:shiela {"lat":12.8020,"lon":77.4020})
CREATE (Person:amelia {"lat":12.8017,"lon":77.4013})
CREATE (Person:paulin {"lat":12.8023,"lon":77.4029})
CREATE (Person:melinda {"lat":12.8011,"lon":77.4019})
```

Number of people living within radius of 3000m from john

```
S=>(@p Person:john)-[#RADIUS_OF 3000 *]->(@c *); RETURN COUNT(c.name) AS Num_places
+----------+
|Num_places|
+----------+
|8         |
+----------+
```

*Note: * in relation ([#RADIUS_OF 3000 *]), means that there may be any relation between subject and object or no relation at all*

Example 3: Chained query

```
CREATE GRAPH gh3
create geoindex gh3.Person latlon = lat.lon
create geoindex gh3.Theater latlon = lat.lon

CREATE (Person:john {"lat":12.8025,"lon":77.4028})
CREATE (Theater:regent {"lat":12.8015,"lon":77.4022, "rating":3.9})-[RUNNING_PLAY]->(Play:"Romeo and Juliet" {"likes":81})
CREATE (Theater:veena {"lat":12.8021,"lon":77.4022, "rating":4.3})-[RUNNING_PLAY]->(Play:"Merchant of Venice" {"likes":88})
CREATE (Theater:mona {"lat":12.8022,"lon":77.4023, "rating":4.1})-[RUNNING_PLAY]->(Play:"ABC Murder" {"likes":71})
CREATE (Theater:ashok {"lat":12.8020,"lon":77.4020, "rating":3.6})-[RUNNING_PLAY]->(Play:"ABC Murder" {"likes":97})
```

Find a Theater(s) within 3KM from person john, which has rating equal to or more than 4.1 and running the play "ABC Murder"

```
S=>(@p Person:john)-[#RADIUS_OF 3000 *]->(@c Theater:* {rating >= 4.1})-[@r RUNNING_PLAY]->(@pl Play:"ABC Murder"); RETURN c.name AS Theater
+-------+
|Theater|
+-------+
|mona   |
+-------+
```

Find the theatre with more than 4.1 likes and play within 3Km from john, which is running some play which has more than 80 likes

```
S=>(@p Person:john)-[#RADIUS_OF 3000 *]->(@c Theater:* {rating >= 4.1})-[@r RUNNING_PLAY]->(@pl Play:* {likes > 80}); RETURN c.name
AS Theater, pl.name AS Play
+------------------+-------+
|Play              |Theater|
+------------------+-------+
|Merchant of Venice|veena  |
+------------------+-------+
```