Graph Infrastructure benchmark

https://github.com/ldbc/ldbc_snb_implementations

The POC is based on LDBC - The Linked Data Benchmark Council is a joint effort to establish benchmarking practices for evaluating graph data management systems. The examination involves evaluation of query and algorithms performance over several scale factors, measurement of bulk loading time and storage size. In addition we evaluate the management utilities, logging and monitoring capabilities.

Hardware (to be specified by the product specialists)

Specification	Value	Comment
CPU		
RAM		
SSD Disk size		Suitable for at least 2T of graph
GPU?		
# Cluster nodes		

Generate Data

The LDBC Social Network Benchmark (SNB) models a social network graph and introduces two different workloads on this common graph. The Interactive Workload specifies a set of read-only traversals that touch a small portion of the graph and is further divided into interactive short (IS) and interactive complex (IC) queries. The Business Intelligence (BI) Workload explores large portions of the graph in search of occurrences of patterns that combine both structural and attribute predicates of varying complexity.

A data generator tool will be used for creating the test graph according to the size, distribution and complexity requirements.

Definition	Value	Comment
Total Size	2T of nodes	5T if possible

Generated	https://github.com/ldbc/ldbc_snb_datagen
	
	https://github.com/ldbc/ldbc_snb_datagen/wiki/Configuration
	https://github.com/ldbc/ldbc_snb_datagen/wiki/Advanced_Configuration
	• `ldbc.snb.datagen.generator.scaleFactor`
	  o SF-1000 2T nodes 17T relationships
	• `ldbc.snb.datagen.generator.distribution.degreeDistribution`
	  o `ldbc.snb.datagen.generator.distribution.AltmannDistribution`
	  o `ldbc.snb.datagen.generator.distribution.FacebookDegreeDistribution`
	
	Output types
	
	https://github.com/ldbc/ldbc_snb_datagen/wiki/Data-	

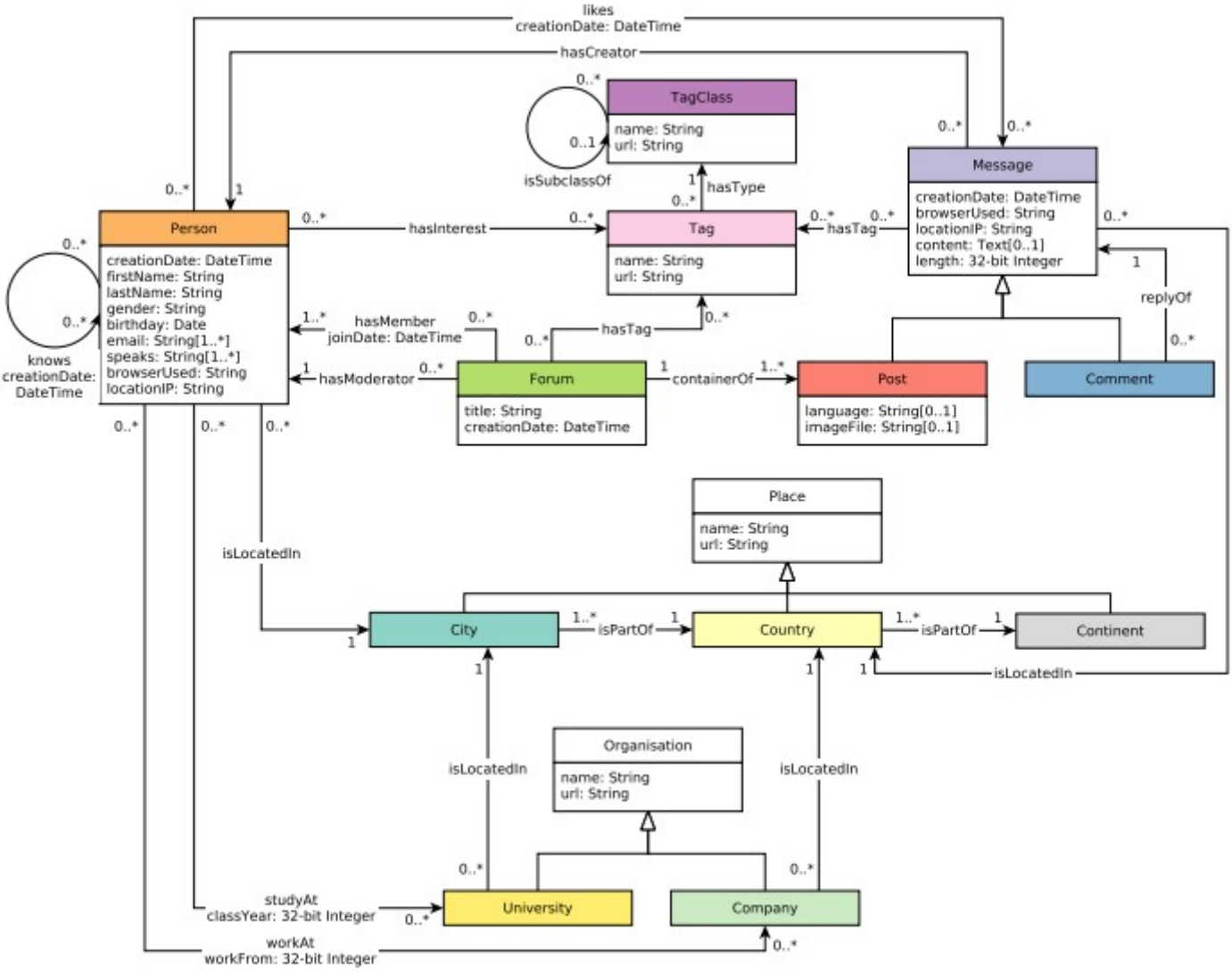
	Output • Use CSV for graph databases that support CSV import.	

Steps to generate the graph (same generated graph will be used across all infra.):

- Generate SF-1000 csv graph
  - git clone https://github.com/ldbc/ldbc_snb_datagen.git
  - `wget http://archive.apache.org/dist/hadoop/core/hadoop-3.2.1/hadoop-3.2.1.tar.gz`
  - `tar xf hadoop-3.2.1.tar.gz`
  - `cp params-csv-basic.ini params.ini`
  - edit run.sh to add
    - `export HADOOP_CLIENT_OPTS="-Xmx2G" – use RAM according to scale factor`
    - `export HADOOP_HOME=`pwd`/hadoop-3.2.1 – point to the path of the hadoop`
    - `export JAVA_HOME=/usr`
  - ./run.sh
  - save ./social_network/static folder in another directory
  - edit params.ini to scalefactor1000
  - ./run.sh again

Ldbc schema

ldbc: https://arxiv.org/pdf/1907.07405.pdf



Data Ingestion

Data ingestion is going to be tested by using any method available (mostly online ingestion), involving several loads, merge performance and graph availability during ingestion.

Data Load

		Comments	Expected?
Frequency	• 1000 • 10000 • 100000	Writes per seconds From X amount of time	
Monitor	• Elapsed time • Total time • GC time • Indexing time	Using infrastructure Demo each infra	

	• % Memory • % CPU • % Disk • % GPU? • % Network In/ Out		
Fault tolerance	• Corrupted CSV • Data not from schema	Readable error log	
Insert by query	• 1000 • 10000 • 100000	Items per second	
Bulk insert	• 1000 • 10000 • 100000 • 1000000	Batch size	
Upsert	• 1000 • 10000 • 100000	Items to be updated at once Nodes/Edges separated	

- Load data to DataStax using Bulk loader
  - git clone https://github.com/ldbc/ldbc_snb_implementations.git
  - copy ./datastax folder (from my laptop)
  - edit ./datastax/convert-csvs.sh, ./datastax /import-to-datastax.sh add (copy from my laptop)
    - `export DSBULK=/home/avi/ldbc/dsbulk-1.4.0/bin`
    - `export DATA_DIR=/home/avi/ldbc/ldbc_snb_datagen/social_network`
    - `export POSTFIX=_0_0.csv`
    - `Comment datetime transformation`
  - `copy headers file from my laptop`
  - run ./datastax/convert-csvs.sh
  - run ./datastax/import-to-datastax.sh
- Load data to DataStax using Spark
  - **TBD**
- Record size on disk(data and index) and load time

Querying

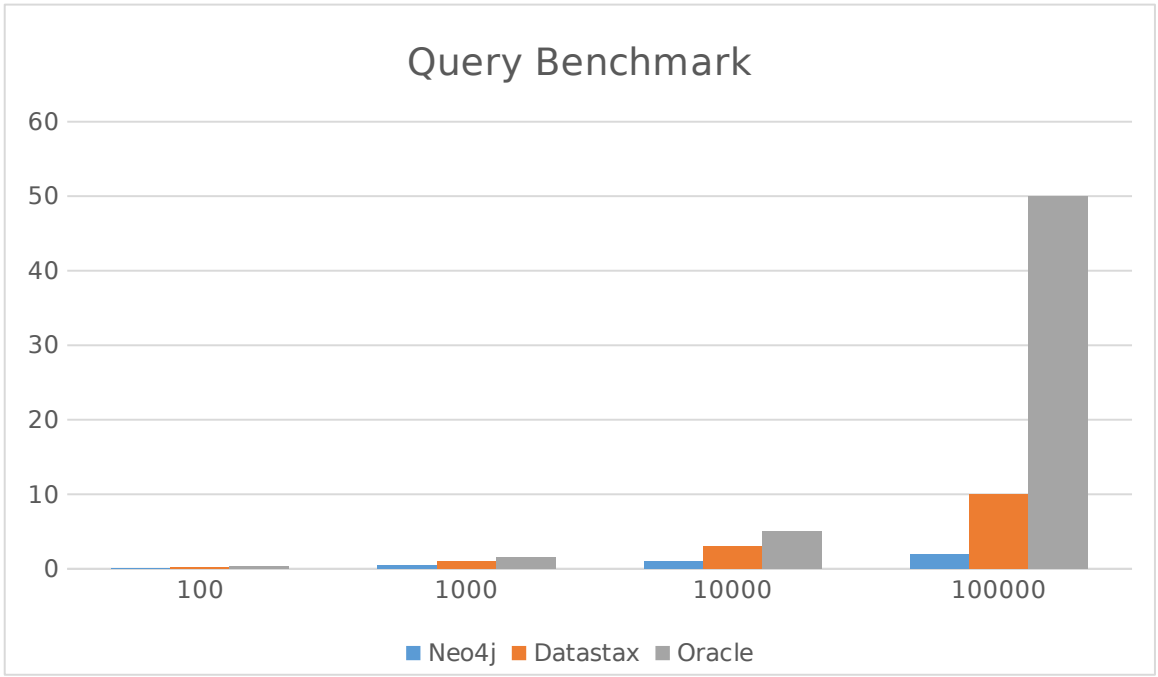
Each query will be executed multiple times in different contexts, for example:

1. Starting from single node
2. Starting from multiple nodes
3. Different size of branching factors (cardinality)
4. Using indexes
5. Sequential execution
6. Parallel execution

The following parameters will be tested:

- Max Time
- Min Time
- Average Time
- Total Time
- Items Returned

The results will be shown using charts that the x axis branch factor(cardinality) y axis is time. For example:



A dedicated java program developed for running the queries sequentially and in parallel with different parameters and save the benchmark result to csv file. The results will be used later on to generate charts to visualize the benchmark and for farther analysis.

- Create parameters to run the benchmark
  - in ldbc_snb_implementation edit and run /interactive-create-validation-parameters.sh
  - `./interactive-validate.sh`
- run java benchmark.jar -datastax 127.0.0.1 /home/avi/ldbc/ldbc_snb_implementations/datastax/queries/ /home/avi/ldbc/ldbc_snb_datagen/substitution_parameters/

Ldbc queries

https://github.com/ldbc/ldbc_snb_implementations/tree/master/cypher/queries

Simple

Given a start Person, retrieve their first name, last name, birthday, IP address, browser, and city of residence.

```
MATCH (n:Person {id:$personId})-[:IS_LOCATED_IN]->(p:Place)
RETURN
  n.firstName AS firstName,
  n.lastName AS lastName,
  n.birthday AS birthday,
  n.locationIP AS locationIP,
  n.browserUsed AS browserUsed,
  p.id AS cityId,
  n.gender AS gender,
  n.creationDate AS creationDate
```

Given a start Person, retrieve the last 10 Messages created by that user. For each Message, return that Message, the original Post in its conversation, and the author of that Post. If any of the Messages is a Post, then the original Post will be the same Message, i.e. that Message will appear twice in that result

```
MATCH (:Person {id:$personId})<-[:HAS_CREATOR]-(m:Message)-[:REPLY_OF*0..]->(p:Post)
MATCH (p)-[:HAS_CREATOR]->(c)
RETURN
  m.id as messageId,
  CASE exists(m.content)
    WHEN true THEN m.content
    ELSE m.imageFile
  END AS messageContent,
  m.creationDate AS messageCreationDate,
  p.id AS originalPostId,
  c.id AS originalPostAuthorId,
  c.firstName as originalPostAuthorFirstName,
  c.lastName as originalPostAuthorLastName
ORDER BY messageCreationDate DESC
LIMIT 10
```

Given a start Person, retrieve all of their friends, and the date at which they became friends

```
MATCH (n:Person {id:$personId})-[r:KNOWS]-(friend)
RETURN
  friend.id AS personId,
  friend.firstName AS firstName,
  friend.lastName AS lastName,
  r.creationDate AS friendshipCreationDate
ORDER BY friendshipCreationDate DESC, toInteger(personId) ASC
```

c. Given a Message, retrieve its content and creation date.

```
MATCH (m:Message {id:$messageId})
RETURN
  m.creationDate as messageCreationDate,
  CASE exists(m.content)
    WHEN true THEN m.content
    ELSE m.imageFile
  END AS messageContent
```

Given a Message, retrieve its author

```
MATCH (m:Message {id:$messageId})-[:HAS_CREATOR]->(p:Person)
RETURN
  p.id AS personId,
```

```
  p.firstName AS firstName,
  p.lastName AS lastName
```

Given a Message, retrieve the Forum that contains it and the Person that moderates that Forum. Since Comments are not directly contained in Forums, for Comments, return the Forum containing the original Post in the thread which the Comment is replying to

```
MATCH (m:Message {id:$messageId})-[:REPLY_OF*0..]->(p:Post)<-[:CONTAINER_OF]-(f:Forum)-
[:HAS_MODERATOR]->(mod:Person)
RETURN
  f.id AS forumId,
  f.title AS forumTitle,
  mod.id AS moderatorId,
  mod.firstName AS moderatorFirstName,
  mod.lastName AS moderatorLastName
```

Given a Message, retrieve the (1-hop) Comments that reply to it. In addition, return a boolean flag knows indicating if the author of the reply knows the author of the original message. If author is same as original author, return false for knows flag.

```
MATCH (m:Message {id:$messageId})<-[:REPLY_OF]-(c:Comment)-[:HAS_CREATOR]->(p:Person)
OPTIONAL MATCH (m)-[:HAS_CREATOR]->(a:Person)-[r:KNOWS]-(p)
RETURN
  c.id AS commentId,
  c.content AS commentContent,
  c.creationDate AS commentCreationDate,
  p.id AS replyAuthorId,
  p.firstName AS replyAuthorFirstName,
  p.lastName AS replyAuthorLastName,
  CASE r
    WHEN null THEN false
    ELSE true
  END AS replyAuthorKnowsOriginalMessageAuthor
ORDER BY commentCreationDate DESC, replyAuthorId
```

Complex

Given a start Person, find Persons with a given first name (firstName) that the start Person is connected to (excluding start Person) by at most 3 steps via the knows relationships. Return Persons, including the distance (1..3), summaries of the Persons workplaces and places of study.

```
MATCH (:Person {id:$personId})-[path:KNOWS*1..3]-(friend:Person)
WHERE friend.firstName = $firstName
WITH friend, min(length(path)) AS distance
ORDER BY distance ASC, friend.lastName ASC, toInteger(friend.id) ASC
LIMIT 20
MATCH (friend)-[:IS_LOCATED_IN]->(friendCity:Place)
OPTIONAL MATCH (friend)-[studyAt:STUDY_AT]->(uni:Organisation)-[:IS_LOCATED_IN]-
>(uniCity:Place)
WITH
  friend,
  collect(
    CASE uni.name
      WHEN null THEN null
      ELSE [uni.name, studyAt.classYear, uniCity.name]
    END
  ) AS unis,
  friendCity,
```

```
  distance
OPTIONAL MATCH (friend)-[workAt:WORK_AT]->(company:Organisation)-[:IS_LOCATED_IN]-
>(companyCountry:Place)
WITH
  friend,
  collect(
    CASE company.name
      WHEN null THEN null
      ELSE [company.name, workAt.workFrom, companyCountry.name]
    END
  ) AS companies,
  unis,
  friendCity,
  distance
RETURN
  friend.id AS friendId,
  friend.lastName AS friendLastName,
  distance AS distanceFromPerson,
  friend.birthday AS friendBirthday,
  friend.creationDate AS friendCreationDate,
  friend.gender AS friendGender,
  friend.browserUsed AS friendBrowserUsed,
  friend.locationIP AS friendLocationIp,
  friend.email AS friendEmails,
  friend.speaks AS friendLanguages,
  friendCity.name AS friendCityName,
  unis AS friendUniversities,
  companies AS friendCompanies
ORDER BY distanceFromPerson ASC, friendLastName ASC, toInteger(friendId) ASC
LIMIT 20
```

Given a start Person, find (the most recent) Messages from all of that Person’s friends, that were created before (and including) a given date (maxDate).

```
MATCH (:Person {id:$personId})-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(message:Message)
WHERE message.creationDate <= $maxDate
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  message.id AS messageId,
  CASE exists(message.content)
    WHEN true THEN message.content
    ELSE message.imageFile
  END AS messageContent,
  message.creationDate AS messageCreationDate
ORDER BY messageCreationDate DESC, toInteger(messageId) ASC
LIMIT 20
```

Given a start Person, find Persons that are their friends and friends of friends (excluding start Person) that have made Posts / Comments in both of the given Countries, CountryX and CountryY, within a given period. Only Persons that are foreign to Countries CountryX and CountryY are considered, that is Persons whose location is neither CountryX nor CountryY.

```
MATCH (person:Person {id:$personId})-[:KNOWS*1..2]-(friend:Person)<-[:HAS_CREATOR]-
(messageX:Message),
(messageX)-[:IS_LOCATED_IN]->(countryX:Place)
WHERE
```

```
  not(person=friend)
  AND not((friend)-[:IS_LOCATED_IN]->()-[:IS_PART_OF]->(countryX))
  AND countryX.name=$countryXName AND messageX.creationDate>=$startDate
  AND messageX.creationDate<$endDate
WITH friend, count(DISTINCT messageX) AS xCount
MATCH (friend)<-[:HAS_CREATOR]-(messageY:Message)-[:IS_LOCATED_IN]->(countryY:Place)
WHERE
  countryY.name=$countryYName
  AND not((friend)-[:IS_LOCATED_IN]->()-[:IS_PART_OF]->(countryY))
  AND messageY.creationDate>=$startDate
  AND messageY.creationDate<$endDate
WITH
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  xCount,
  count(DISTINCT messageY) AS yCount
RETURN
  personId,
  personFirstName,
  personLastName,
  xCount,
  yCount,
  xCount + yCount AS count
ORDER BY count DESC, toInteger(personId) ASC
LIMIT 20
```

Given a start Person (personId), find Tags that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to friends' Posts created within a given time interval, and that were never attached to friends' Posts created before this interval.

```
MATCH (person:Person {id:$personId})-[:KNOWS]-(:Person)<-[:HAS_CREATOR]-(post:Post)-
[:HAS_TAG]->(tag:Tag)
WHERE post.creationDate >= $startDate
   AND post.creationDate < $endDate
WITH person, count(post) AS postsOnTag, tag
OPTIONAL MATCH (person)-[:KNOWS]-()<-[:HAS_CREATOR]-(oldPost:Post)-[:HAS_TAG]->(tag)
WHERE oldPost.creationDate < $startDate
WITH person, postsOnTag, tag, count(oldPost) AS cp
WHERE cp = 0
RETURN
  tag.name AS tagName,
  sum(postsOnTag) AS postCount
ORDER BY postCount DESC, tagName ASC
LIMIT 10
```

Given a start Person, find the Forums which that Person's friends and friends of friends (excluding start Person) became Members of after a given date. For each Forum find the number of Posts that were created by any of these Persons. For each Forum and consider only those Persons which joined that particular Forum after the given date (minDate).

```
MATCH (person:Person {id:$personId})-[:KNOWS*1..2]-(friend:Person)<-
[membership:HAS_MEMBER]-(forum:Forum)
WHERE membership.joinDate>$minDate
    AND not(person=friend)
WITH DISTINCT friend, forum
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)<-[:CONTAINER_OF]-(forum)
WITH forum, count(post) AS postCount
```

```
RETURN
  forum.title AS forumTitle,
  postCount
ORDER BY postCount DESC, toInteger(forum.id) ASC
LIMIT 20
```

Given a start Person and some Tag, find the other Tags that occur together with this Tag on Posts that were created by start Person's friends and friends of friends (excluding start Person). Return top 10 Tags, and the count of Posts that were created by these Persons, which contain both this Tag and the given Tag.

```
MATCH
  (person:Person {id:$personId})-[:KNOWS*1..2]-(friend:Person),
  (friend)<-[:HAS_CREATOR]-(friendPost:Post)-[:HAS_TAG]->(knownTag:Tag {name:$tagName})
WHERE not(person=friend)
MATCH (friendPost)-[:HAS_TAG]->(commonTag:Tag)
WHERE not(commonTag=knownTag)
WITH DISTINCT commonTag, knownTag, friend
MATCH (commonTag)<-[:HAS_TAG]-(commonPost:Post)-[:HAS_TAG]->(knownTag)
WHERE (commonPost)-[:HAS_CREATOR]->(friend)
RETURN
  commonTag.name AS tagName,
  count(commonPost) AS postCount
ORDER BY postCount DESC, tagName ASC
LIMIT 10
```

Given a start Person, find (most recent) likes on any of start Person's Messages. Find Persons that liked (likes edge) any of start Person's Messages, the Messages they liked most recently, the creation date of that like, and the latency in minutes (minutesLatency) between creation of Messages and like. Additionally, for each Person found return a flag indicating (isNew) whether the liker is a friend of start Person. In case that a Person liked multiple Messages at the same time, return the Message with lowest identifier.

```
MATCH (person:Person {id:$personId})<-[:HAS_CREATOR]-(message:Message)<-[like:LIKES]-
(liker:Person)
WITH liker, message, like.creationDate AS likeTime, person
ORDER BY likeTime DESC, toInteger(message.id) ASC
WITH
  liker,
  head(collect({msg: message, likeTime: likeTime})) AS latestLike,
  person
RETURN
  liker.id AS personId,
  liker.firstName AS personFirstName,
  liker.lastName AS personLastName,
  latestLike.likeTime AS likeCreationDate,
  latestLike.msg.id AS messageId,
  CASE exists(latestLike.msg.content)
    WHEN true THEN latestLike.msg.content
    ELSE latestLike.msg.imageFile
  END AS messageContent,
  latestLike.msg.creationDate AS messageCreationDate,
  not((liker)-[:KNOWS]-(person)) AS isNew
ORDER BY likeCreationDate DESC, toInteger(personId) ASC
LIMIT 20
```

Given a start Person, find (most recent) Comments that are replies to Messages of the start Person. Only consider direct (single-hop) replies, not the transitive (multi-hop) ones. Return the reply Comments, and the Person that created each reply Comment.

```
MATCH
  (start:Person {id:$personId})<-[:HAS_CREATOR]-(:Message)<-[:REPLY_OF]-
(comment:Comment)-[:HAS_CREATOR]->(person:Person)
RETURN
  person.id AS personId,
  person.firstName AS personFirstName,
  person.lastName AS personLastName,
  comment.creationDate AS commentCreationDate,
  comment.id AS commentId,
  comment.content AS commentContent
ORDER BY commentCreationDate DESC, toInteger(commentId) ASC
LIMIT 20
```

Given a start Person, find (the most recent) Messages created by that Person’s friends or friends of friends (excluding start Person). Only consider the Messages created before a given date (excluding that day).

```
MATCH (:Person {id:$personId})-[:KNOWS*1..2]-(friend:Person)<-[:HAS_CREATOR]-
(message:Message)
WHERE message.creationDate < $maxDate
RETURN DISTINCT
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  message.id AS messageId,
  CASE exists(message.content)
    WHEN true THEN message.content
    ELSE message.imageFile
  END AS messageContent,
  message.creationDate AS messageCreationDate
ORDER BY message.creationDate DESC, toInteger(message.id) ASC
LIMIT 20
```

Given a start Person with id personId, find that Person’s friends of friends (person) – excluding the start Person and his/her immediate friends –, who were born on or after the 21st of a given month (in any year) and before the 22nd of the following month. Calculate the similarity between each person and the start Person, where commonInterestScore is defined as follows:

- common = number of Posts created by person, such that the Post has a Tag that the start Person, is interested in
- uncommon = number of Posts created by person, such that the Post has no Tag that the start Person, is interested in
- commonInterestScore = common – uncommon

```
MATCH (person:Person {id:$personId})-[:KNOWS*2..2]-(friend:Person)-[:IS_LOCATED_IN]-
>(city:Place)
WHERE
  ((friend.birthday/100%100 = $month AND friend.birthday%100 >= 21) OR
  (friend.birthday/100%100 = $nextMonth AND friend.birthday%100 < 22))
  AND not(friend=person)
  AND not((friend)-[:KNOWS]-(person))
WITH DISTINCT friend, city, person
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
```

```
WITH friend, city, collect(post) AS posts, person
WITH
  friend,
  city,
  length(posts) AS postCount,
  length([p IN posts WHERE (p)-[:HAS_TAG]->(:Tag)<-[:HAS_INTEREST]-(person)]) AS
commonPostCount
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  commonPostCount - (postCount - commonPostCount) AS commonInterestScore,
  friend.gender AS personGender,
  city.name AS personCityName
ORDER BY commonInterestScore DESC, toInteger(personId) ASC
LIMIT 10
```

Given a start Person, find that Person's friends and friends of friends (excluding start Person) who started working in some Company in a given Country, before a given date (year).

```
MATCH (person:Person {id:$personId})-[:KNOWS*1..2]-(friend:Person)
WHERE not(person=friend)
WITH DISTINCT friend
MATCH (friend)-[workAt:WORK_AT]->(company:Organisation)-[:IS_LOCATED_IN]->(:Place {name:
$countryName})
WHERE workAt.workFrom < $workFromYear
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  company.name AS organizationName,
  workAt.workFrom AS organizationWorkFromYear
ORDER BY organizationWorkFromYear ASC, toInteger(personId) ASC, organizationName DESC
LIMIT 10
```

Given a start Person, find the Comments that this Person's friends made in reply to Posts, considering only those Comments that are direct (single-hop) replies to Posts, not the transitive (multihop) ones. Only consider Posts with a Tag in a given TagClass or in a descendent of that TagClass. Count the number of these reply Comments, and collect the Tags that were attached to the Posts they replied to, but only collect Tags with the given TagClass or with a descendant of that TagClass. Return Persons with at least one reply, the reply count, and the collection of Tags.

```
MATCH (:Person {id:$personId})-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-
(comment:Comment)-[:REPLY_OF]->(:Post)-[:HAS_TAG]->(tag:Tag),
  (tag)-[:HAS_TYPE]->(tagClass:TagClass)-[:IS_SUBCLASS_OF*0..]->(baseTagClass:TagClass)
WHERE tagClass.name = $tagClassName OR baseTagClass.name = $tagClassName
RETURN
  friend.id AS personId,
  friend.firstName AS personFirstName,
  friend.lastName AS personLastName,
  collect(DISTINCT tag.name) AS tagNames,
  count(DISTINCT comment) AS replyCount
ORDER BY replyCount DESC, toInteger(personId) ASC
LIMIT 20
```

Given two Persons, find the shortest path between these two Persons in the subgraph induced by the knows relationships. Return the length of this path:

- −1: no path found
- 0: start person = end person
- > 0: regular case

```
MATCH (person1:Person {id:$person1Id}), (person2:Person {id:$person2Id})
OPTIONAL MATCH path = shortestPath((person1)-[:KNOWS*]-(person2))
RETURN
CASE path IS NULL
  WHEN true THEN -1
  ELSE length(path)
END AS shortestPathLength;
```

Given two Persons, find all (unweighted) shortest paths between these two Persons, in the subgraph induced by the knows relationship. Then, for each path calculate a weight. The nodes in the path are Persons, and the weight of a path is the sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated based on their interactions:

- Every direct reply (by one of the Persons) to a Post (by the other Person) contributes 1.0.
- Every direct reply (by one of the Persons) to a Comment (by the other Person) contributes 0.5.

Return all the paths with shortest length, and their weights. Do not return any rows if there is no path between the two Persons

```
MATCH path = allShortestPaths((person1:Person {id:$person1Id})-[:KNOWS*..15]-
(person2:Person {id:$person2Id}))
WITH nodes(path) AS pathNodes
RETURN
  extract(n IN pathNodes | n.id) AS personIdsInPath,
  reduce(weight=0.0, idx IN range(1,size(pathNodes)-1) | extract(prev IN [pathNodes[idx-
1]] | extract(curr IN [pathNodes[idx]] | weight + length((curr)<-[:HAS_CREATOR]-
(:Comment)-[:REPLY_OF]->(:Post)-[:HAS_CREATOR]->(prev))*1.0 + length((prev)<-
[:HAS_CREATOR]-(:Comment)-[:REPLY_OF]->(:Post)-[:HAS_CREATOR]->(curr))*1.0 +
length((prev)-[:HAS_CREATOR]-(:Comment)-[:REPLY_OF]-(:Comment)-[:HAS_CREATOR]-
(curr))*0.5) )[0][0]) AS pathWight
ORDER BY pathWight DESC
```

BI

Given a date, find all Messages created before that date. Group them by a 3-level grouping:

1. by year of creation
2. for each year, group into Message types: is Comment or not
3. for each year-type group, split into four groups based on length of their content
    - 0: 0 <= length < 40 (short)
    - 1: 40 <= length < 80 (one liner)
    - 2: 80 <= length < 160 (tweet)
    - 3: 160 <= length (long)

```
// Q1. Posting summary
/*
  :param { date: 20110721220000000 }
*/
```

```
MATCH (message:Message)
WHERE message.creationDate < $date
WITH count(message) AS totalMessageCountInt // this should be a subquery once Cypher
supports it
WITH toFloat(totalMessageCountInt) AS totalMessageCount
MATCH (message:Message)
WHERE message.creationDate < $date
  AND message.content IS NOT NULL
WITH
  totalMessageCount,
  message,
  message.creationDate/10000000000000 AS year
WITH
  totalMessageCount,
  year,
  message:Comment AS isComment,
  CASE
    WHEN message.length <  40 THEN 0
    WHEN message.length <  80 THEN 1
    WHEN message.length < 160 THEN 2
    ELSE                      3
  END AS lengthCategory,
  count(message) AS messageCount,
  floor(avg(message.length)) AS averageMessageLength,
  sum(message.length) AS sumMessageLength
RETURN
  year,
  isComment,
  lengthCategory,
  messageCount,
  averageMessageLength,
  sumMessageLength,
  messageCount / totalMessageCount AS percentageOfMessages
ORDER BY
  year DESC,
  isComment ASC,
  lengthCategory ASC
```

Select all Messages created in the range of [startDate, endDate] by Persons located in country1 or country2. Select the creator Persons and the Tags of these Messages. Split these Persons, Tags and Messages into a 5-level grouping:

1. name of country of Person,
2. month the Message was created,
3. gender of Person,
4. age group of Person, defined as years between person’s birthday and end of simulation (2013-01-01), divided by 5, rounded down (partial years do not count),
5. name of tag attached to Message.

Consider only those groups where number of Messages is greater than 100

```
// Q2. Top tags for country, age, gender, time
/*
  :param {
    date1: 20091231230000000,
    date2: 20101107230000000,
    country1: 'Ethiopia',
    country2: 'Belarus'
  }
```

```
*/
MATCH
  (country:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(person:Person)
  <-[:HAS_CREATOR]-(message:Message)-[:HAS_TAG]->(tag:Tag)
WHERE message.creationDate >= $startDate
  AND message.creationDate <= $endDate
  AND (country.name = $country1 OR country.name = $country2)
WITH
  country.name AS countryName,
  message.creationDate/100000000000%100 AS month,
  person.gender AS gender,
  floor((20130101 - person.birthday) / 10000 / 5.0) AS ageGroup,
  tag.name AS tagName,
  message
WITH
  countryName, month, gender, ageGroup, tagName, count(message) AS messageCount
WHERE messageCount > 100
RETURN
  countryName,
  month,
  gender,
  ageGroup,
  tagName,
  messageCount
ORDER BY
  messageCount DESC,
  tagName ASC,
  ageGroup ASC,
  gender ASC,
  month ASC,
  countryName ASC
LIMIT 100
```

Find the Tags that were used in Messages during the given month of the given year and the Tags that were used during the next month. For the Tags and for both months, compute the count of Messages.

```
// Q3. Tag evolution
/*
  :param {
    year: 2010,
    month: 10
  }
*/
WITH
  $year AS year1,
  $month AS month1,
  $year + toInteger($month / 12.0) AS year2,
  $month % 12 + 1 AS month2
// year-month 1
MATCH (tag:Tag)
OPTIONAL MATCH (message1:Message)-[:HAS_TAG]->(tag)
  WHERE message1.creationDate/10000000000000   = year1
    AND message1.creationDate/100000000000%100 = month1
WITH year2, month2, tag, count(message1) AS countMonth1
// year-month 2
OPTIONAL MATCH (message2:Message)-[:HAS_TAG]->(tag)
  WHERE message2.creationDate/10000000000000   = year2
    AND message2.creationDate/100000000000%100 = month2
WITH
```

```
  tag,
  countMonth1,
  count(message2) AS countMonth2
RETURN
  tag.name,
  countMonth1,
  countMonth2,
  abs(countMonth1-countMonth2) AS diff
ORDER BY
  diff DESC,
  tag.name ASC
LIMIT 100
```

Given a TagClass and a Country, find all the Forums created in the given Country, containing at least one Post with Tags belonging directly to the given TagClass. The location of a Forum is identified by the location of the Forum's moderator.

```
// Q4. Popular topics in a country
/*
  :param {
    tagClass: 'MusicalArtist',
    country: 'Burma'
  }
*/
MATCH
  (:Country {name: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-
  (person:Person)<-[:HAS_MODERATOR]-(forum:Forum)-[:CONTAINER_OF]->
  (post:Post)-[:HAS_TAG]->(:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass})
RETURN
  forum.id,
  forum.title,
  forum.creationDate,
  person.id,
  count(DISTINCT post) AS postCount
ORDER BY
  postCount DESC,
  forum.id ASC
LIMIT 20
```

Find the most popular Forums for a given Country, where the popularity of a Forum is measured by the number of members that Forum has from the given Country. Calculate the top 100 most popular Forums. In case of a tie, the forum(s) with the smaller id value(s) should be selected. For each member Person of the 100 most popular Forums, count the number of Posts (postCount) they made in any of those (most popular) Forums. Also include those member Persons who have not posted any messages (have a postCount of 0).

```
// Q5. Top posters in a country
/*
  :param { country: 'Belarus' }
*/
MATCH
  (:Country {name: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-
  (person:Person)<-[:HAS_MEMBER]-(forum:Forum)
WITH forum, count(person) AS numberOfMembers
ORDER BY numberOfMembers DESC, forum.id ASC
LIMIT 100
WITH collect(forum) AS popularForums
UNWIND popularForums AS forum
```

```
MATCH
  (forum)-[:HAS_MEMBER]->(person:Person)
OPTIONAL MATCH
  (person)<-[:HAS_CREATOR]-(post:Post)<-[:CONTAINER_OF]-(popularForum:Forum)
WHERE popularForum IN popularForums
RETURN
  person.id,
  person.firstName,
  person.lastName,
  person.creationDate,
  count(DISTINCT post) AS postCount
ORDER BY
  postCount DESC,
  person.id ASC
LIMIT 100
```

Get each Person (person) who has created a Message (message) with a given Tag (direct relation, not transitive). Considering only these messages, for each Person node:

- Count its messages (messageCount).
- Count likes (likeCount) to its messages.
- Count Comments (replyCount) in reply to it messages.

The score is calculated according to the following formula: 1 * messageCount + 2 * replyCount + 10 * likeCount

```
// Q6. Most active Posters of a given Topic
/*
  :param { tag: 'Abbas_I_of_Persia' }
*/
MATCH (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]-
>(person:Person)
OPTIONAL MATCH (:Person)-[like:LIKES]->(message)
OPTIONAL MATCH (message)<-[:REPLY_OF]-(comment:Comment)
WITH person, count(DISTINCT like) AS likeCount, count(DISTINCT comment) AS replyCount,
count(DISTINCT message) AS messageCount
RETURN
  person.id,
  replyCount,
  likeCount,
  messageCount,
  1*messageCount + 2*replyCount + 10*likeCount AS score
ORDER BY
  score DESC,
  person.id ASC
LIMIT 100
```

Given a Tag, find all Persons (person) that ever created a Message (message1) with the given Tag. For each of these Persons (person) compute their “authority score” as follows:

- The “authority score” is the sum of “popularity scores” of the Persons (person2) that liked any of that Person’s Messages (message2) with the given Tag.
- A Person’s (person2) “popularity score” is defined as the total number of likes on all of their Messages (message3).

```
// Q7. Most authoritative users on a given topic
/*
  :param { tag: 'Arnold_Schwarzenegger' }
```

```
*/
MATCH (tag:Tag {name: $tag})
MATCH (tag)<-[:HAS_TAG]-(message1:Message)-[:HAS_CREATOR]->(person1:Person)
MATCH (tag)<-[:HAS_TAG]-(message2:Message)-[:HAS_CREATOR]->(person1)
OPTIONAL MATCH (message2)<-[:LIKES]-(person2:Person)
OPTIONAL MATCH (person2)<-[:HAS_CREATOR]-(message3:Message)<-[like:LIKES]-(p3:Person)
RETURN
  person1.id,
  count(DISTINCT like) AS authorityScore
ORDER BY
  authorityScore DESC,
  person1.id ASC
LIMIT 100
```

Find all Messages that have a given Tag. Find the related Tags attached to (direct) reply Comments of these Messages, but only of those reply Comments that do not have the given Tag. Group the Tags by name, and get the count of replies in each group.

```
// Q8. Related Topics
/*
  :param { tag: 'Genghis_Khan' }
*/
MATCH
  (tag:Tag {name: $tag})<-[:HAS_TAG]-(message:Message),
  (message)<-[:REPLY_OF]-(comment:Comment)-[:HAS_TAG]->(relatedTag:Tag)
WHERE NOT (comment)-[:HAS_TAG]->(tag)
RETURN
  relatedTag.name,
  count(DISTINCT comment) AS count
ORDER BY
  count DESC,
  relatedTag.name ASC
LIMIT 100
```

Given two TagClasses (tagClass1 and tagClass2), find Forums that contain

- at least one Post (post1) with a Tag with a (direct) type of tagClass1 and
- at least one Post (post2) with a Tag with a (direct) type of tagClass2.

The post1 and post2 nodes may be the same Post. Consider the Forums with a number of members greater than a given threshold. For every such Forum, count the number of post1 nodes (count1) and the number of post2 nodes (count2).

```
// Q9. Forum with related Tags
/*
  :param {
    tagClass1: 'BaseballPlayer',
    tagClass2: 'ChristianBishop',
    threshold: 200
  }
*/
MATCH
  (forum:Forum)-[:HAS_MEMBER]->(person:Person)
WITH forum, count(person) AS members
WHERE members > $threshold
MATCH
  (forum)-[:CONTAINER_OF]->(post1:Post)-[:HAS_TAG]->
  (:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass1})
```

```
WITH forum, count(DISTINCT post1) AS count1
MATCH
  (forum)-[:CONTAINER_OF]->(post2:Post)-[:HAS_TAG]->
  (:Tag)-[:HAS_TYPE]->(:TagClass {name: $tagClass2})
WITH forum, count1, count(DISTINCT post2) AS count2
RETURN
  forum.id,
  count1,
  count2
ORDER BY
  abs(count2-count1) DESC,
  forum.id ASC
LIMIT 100
```

Given a Tag, find all Persons that are interested in the Tag and/or have written a Message (Post or Comment) with a creationDate after a given date and that has a given Tag. For each Person, compute the score as the sum of the following two aspects:

- 100, if the Person has this Tag as their interest, or 0 otherwise
- number of Messages by this Person with the given Tag

Also, for each Person, compute the sum of the score of the Person’s friends (friendsScore).

```
// Q10. Central Person for a Tag
/*
  :param {
    tag: 'John_Rhys-Davies',
    date: 20120122000000000
  }
*/
MATCH (tag:Tag {name: $tag})
// score
OPTIONAL MATCH (tag)<-[interest:HAS_INTEREST]-(person:Person)
WITH tag, collect(person) AS interestedPersons
OPTIONAL MATCH (tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person:Person)
         WHERE message.creationDate > $date
WITH tag, interestedPersons + collect(person) AS persons
UNWIND persons AS person
// poor man's disjunct union (should be changed to UNION + post-union processing in the
future)
WITH DISTINCT tag, person
WITH
  tag,
  person,
  100 * length([(tag)<-[interest:HAS_INTEREST]-(person) | interest])
    + length([(tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(person) WHERE
message.creationDate > $date | message])
  AS score
OPTIONAL MATCH (person)-[:KNOWS]-(friend)
WITH
  person,
  score,
  100 * length([(tag)<-[interest:HAS_INTEREST]-(friend) | interest])
    + length([(tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]->(friend) WHERE
message.creationDate > $date | message])
  AS friendScore
RETURN
  person.id,
  score,
  sum(friendScore) AS friendsScore
```

```
ORDER BY
  score + friendsScore DESC,
  person.id ASC
LIMIT 100
```

Find those Persons of a given Country that replied to any Message, such that the reply does not have any Tag in common with the Message (only direct replies are considered, transitive ones are not). Consider only those replies that do no contain any word from a given blacklist. For each Person and valid reply, retrieve the Tags associated with the reply, and retrieve the number of likes on the reply. The detailed conditions for checking blacklisted words are currently as follows. Words do not have to stand separately, i.e. if the word “Green” is blacklisted, “South-Greenland” cannot be included in the results. Also, comparison should be done in a case-sensitive way. These conditions are preliminary and might be changed in later versions of the benchmark.

```
// Q11. Unrelated replies
/*
  :param {
    country: 'Germany',
    blacklist: ['also', 'Pope', 'that', 'James', 'Henry', 'one', 'Green']
  }
*/
WITH $blacklist AS blacklist
MATCH
  (country:Country {name: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-
  (person:Person)<-[:HAS_CREATOR]-(reply:Comment)-[:REPLY_OF]->(message:Message),
  (reply)-[:HAS_TAG]->(tag:Tag)
WHERE NOT (message)-[:HAS_TAG]->(:Tag)<-[:HAS_TAG]-(reply)
  AND size([word IN blacklist WHERE reply.content CONTAINS word | word]) = 0
OPTIONAL MATCH
  (:Person)-[like:LIKES]->(reply)
RETURN
  person.id,
  tag.name,
  count(DISTINCT like) AS countLikes,
  count(DISTINCT reply) AS countReplies
ORDER BY
  countLikes DESC,
  person.id ASC,
  tag.name ASC
LIMIT 100
```

Find all Messages created after a given date (exclusive), that received more than a given number of likes (likeThreshold)

```
// Q12. Trending Posts
/*
  :param {
    date: 20110721220000000,
    likeThreshold: 400
  }
*/
MATCH
  (message:Message)-[:HAS_CREATOR]->(creator:Person),
  (message)<-[like:LIKES]-(:Person)
WHERE message.creationDate > $date
WITH message, creator, count(like) AS likeCount
WHERE likeCount > $likeThreshold
RETURN
```

```
  message.id,
  message.creationDate,
  creator.firstName,
  creator.lastName,
  likeCount
ORDER BY
  likeCount DESC,
  message.id ASC
LIMIT 100
```

Find all Messages in a given Country, as well as their Tags. Group Messages by creation year and month. For each group, find the 5 most popular Tags, where popularity is the number of Messages (from within the same group) where the Tag appears. Note: even if there are no Tags for Messages in a given year and month, the result should include the year and month with an empty popularTags list.

```
// Q13. Popular Tags per month in a country
/*
  :param { country: 'Burma' }
*/
MATCH (:Country {name: $country})<-[:IS_LOCATED_IN]-(message:Message)
OPTIONAL MATCH (message)-[:HAS_TAG]->(tag:Tag)
WITH
  message.creationDate/10000000000000   AS year,
  message.creationDate/100000000000%100 AS month,
  message,
  tag
WITH year, month, count(message) AS popularity, tag
ORDER BY popularity DESC, tag.name ASC
WITH
  year,
  month,
  collect([tag.name, popularity]) AS popularTags
WITH
  year,
  month,
  [popularTag IN popularTags WHERE popularTag[0] IS NOT NULL] AS popularTags
RETURN
  year,
  month,
  popularTags[0..5] AS topPopularTags
ORDER BY
  year DESC,
  month ASC
LIMIT 100
```

For each Person, count the number of Posts they created in the time interval [startDate, endDate] (equivalent to the number of threads they initiated) and the number of Messages in each of their (transitive) reply trees, including the root Post of each tree. When calculating Message counts only consider messages created within the given time interval. Return each Person, number of Posts they created, and the count of all Messages that appeared in the reply trees (including the Post at the root of tree) they created.

```
// Q14. Top thread initiators
/*
  :param {
    startDate: 20120531220000000,
```

```
    endDate: 20120630220000000
  }
*/
MATCH (person:Person)<-[:HAS_CREATOR]-(post:Post)<-[:REPLY_OF*0..]-(reply:Message)
WHERE  post.creationDate >= $startDate
  AND  post.creationDate <= $endDate
  AND reply.creationDate >= $startDate
  AND reply.creationDate <= $endDate
RETURN
  person.id,
  person.firstName,
  person.lastName,
  count(DISTINCT post) AS threadCount,
  count(DISTINCT reply) AS messageCount
ORDER BY
  messageCount DESC,
  person.id ASC
LIMIT 100
```

Given a Country country, determine the “social normal”, i.e. the floor of average number of friends that Persons of country have in country. Then, find all Persons in country, whose number of friends in country equals the social normal value.

```
// Q15. Social normals
/*
  :param { country: 'Burma' }
*/
MATCH
  (country:Country {name: $country})
MATCH
  (country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(person1:Person)
OPTIONAL MATCH
  // start a new MATCH as friend might live in the same City
  // and thus can reuse the IS_PART_OF edge
  (country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(friend1:Person),
  (person1)-[:KNOWS]-(friend1)
WITH country, person1, count(friend1) AS friend1Count
WITH country, avg(friend1Count) AS socialNormalFloat
WITH country, floor(socialNormalFloat) AS socialNormal
MATCH
  (country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(person2:Person)
OPTIONAL MATCH
  (country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(friend2:Person)-[:KNOWS]-(person2)
WITH country, person2, count(friend2) AS friend2Count, socialNormal
WHERE friend2Count = socialNormal
RETURN
  person2.id,
  friend2Count AS count
ORDER BY
  person2.id ASC
LIMIT 100
```

Given a Person, find all other Persons that live in a given Country and are connected to given Person by a transitive trail with length in range [minPathDistance, maxPathDistance] through the knows relation. In the trail, an edge can be only traversed once while nodes can be traversed multiple times (as opposed to a path which allows repetitions of both nodes and edges). For each of these Persons, retrieve all of their Messages that contain at least one Tag belonging to a given TagClass (direct relation not transitive). For each Message, retrieve all of its Tags. Group the

results by Persons and Tags, then count the Messages by a certain Person having a certain Tag. (Note: it is not yet decided whether a Person connected to the start Person on a trail with a length smaller than minPathDistance, but also on a trail with the length in [minPathDistance, maxPathDistance] should be included. The current reference implementations allow such Persons, but this might be subject to change in the future.)

```
// Q16. Experts in social circle
/*
  :param {
    personId: 19791209310731,
    country: 'Pakistan',
    tagClass: 'MusicalArtist',
    minPathDistance: 3,
    maxPathDistance: 5
  }
*/
// This query will not work in a browser as is. I tried alternatives approaches,
// e.g. enabling path of arbitrary lengths, saving the path to a variable p and
// checking for `$minPathDistance <= length(p)`, but these could not be
// evaluated due to the excessive amount of paths.
// If you would like to test the query in the browser, replace the values of
// $minPathDistance and $maxPathDistance to a constant.
MATCH
  (:Person {id: $personId})-[:KNOWS*$minPathDistance..$maxPathDistance]-(person:Person)
WITH DISTINCT person
MATCH
  (person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(:Country {name: $country}),
  (person)<-[:HAS_CREATOR]-(message:Message)-[:HAS_TAG]->(:Tag)-[:HAS_TYPE]->
  (:TagClass {name: $tagClass})
MATCH
  (message)-[:HAS_TAG]->(tag:Tag)
RETURN
  person.id,
  tag.name,
  count(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  tag.name ASC,
  person.id ASC
LIMIT 100
```

For a given country, count all the distinct triples of Persons such that:

- a is friend of b,
- b is friend of c,
- c is friend of a.

Distinct means that given a triple t1 in the result set R of all qualified triples, there is no triple t2 in R such that t1 and t2 have the same set of elements.

```
// Q17. Friend triangles
/*
  :param { country: 'Spain' }
*/
MATCH (country:Country {name: $country})
MATCH (a:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country)
MATCH (b:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country)
MATCH (c:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country)
MATCH (a)-[:KNOWS]-(b), (b)-[:KNOWS]-(c), (c)-[:KNOWS]-(a)
```

```
WHERE a.id < b.id
  AND b.id < c.id
RETURN count(*) AS count
// as a less elegant solution, count(a) also works
```

For each Person, count the number of Messages they made (messageCount). Only count Messages with the following attributes:

- Its content is not empty (and consequently, imageFile empty for Posts).
- Its length is below the lengthThreshold (exclusive, equality is not allowed).
- Its creationDate is after date (exclusive, equality is not allowed).
- It is written in any of the given languages.
    - The language of a Post is defined by its language attribute.
    - The language of a Comment is that of the Post that initiates the thread where the Comment replies to.

  The Post and Comments in the reply tree's path (from the Message to the Post) do not have to satisfy the constraints for content, length and creationDate.

For each messageCount value, count the number of Persons with exactly messageCount Messages (with the required attributes).

```
// Q18. How many persons have a given number of posts
/*
  :param {
    date: 20110722000000000,
    lengthThreshold: 20,
    languages: ['ar']
  }
*/
MATCH (person:Person)
OPTIONAL MATCH (person)<-[:HAS_CREATOR]-(message:Message)-[:REPLY_OF*0..]->(post:Post)
WHERE message.content IS NOT NULL
  AND message.length < $lengthThreshold
  AND message.creationDate > $date
  AND post.language IN $languages
WITH
  person,
  count(message) AS messageCount
RETURN
  messageCount,
  count(person) AS personCount
ORDER BY
  personCount DESC,
  messageCount DESC
```

For all the Persons (person) born after a certain date, find all the strangers they interacted with, where strangers are Persons that do not know person. There is no restriction on the date that strangers were born. (Of course, person and stranger are required to be two different Persons.) Consider only strangers that are

- members of Forums tagged with a Tag with a (direct) type of tagClass1 and
- members of Forums tagged with a Tag with a (direct) type of tagClass2.

The Tags may be attached to the same Forum or they may be attached to different Forums. Interaction is defined as follows: the person has replied to a Message by the stranger B (the

reply might be a transitive one). For each person, count the number of strangers they interacted with (strangerCount) and total number of times they interacted with them (interactionCount).

```
// Q19. Stranger's interaction
/*
  :param {
    date: 19890101,
    tagClass1: 'MusicalArtist',
    tagClass2: 'OfficeHolder'
  }
*/
MATCH
  (:TagClass {name: $tagClass1})<-[:HAS_TYPE]-(:Tag)<-[:HAS_TAG]-
  (forum1:Forum)-[:HAS_MEMBER]->(stranger:Person)
WITH DISTINCT stranger
MATCH
  (:TagClass {name: $tagClass2})<-[:HAS_TYPE]-(:Tag)<-[:HAS_TAG]-
  (forum2:Forum)-[:HAS_MEMBER]->(stranger)
WITH DISTINCT stranger
MATCH
  (person:Person)<-[:HAS_CREATOR]-(comment:Comment)-[:REPLY_OF*]->(message:Message)-
[:HAS_CREATOR]->(stranger)
WHERE person.birthday > $date
  AND person <> stranger
  AND NOT (person)-[:KNOWS]-(stranger)
  AND NOT (message)-[:REPLY_OF*]->(:Message)-[:HAS_CREATOR]->(stranger)
RETURN
  person.id,
  count(DISTINCT stranger) AS strangersCount,
  count(comment) AS interactionCount
ORDER BY
  interactionCount DESC,
  person.id ASC
LIMIT 100
```

For all given TagClasses, count number of Messages that have a Tag that belongs to that TagClass or any of its children (all descendants through a transitive relation).

```
// Q20. High-level topics
/*
  :param { tagClasses: ['Writer', 'Single', 'Country'] }
*/
UNWIND $tagClasses AS tagClassName
MATCH
  (tagClass:TagClass {name: tagClassName})<-[:IS_SUBCLASS_OF*0..]-
  (:TagClass)<-[:HAS_TYPE]-(tag:Tag)<-[:HAS_TAG]-(message:Message)
RETURN
  tagClass.name,
  count(DISTINCT message) AS messageCount
ORDER BY
  messageCount DESC,
  tagClass.name ASC
LIMIT 100
```

Find zombies within the given country, and return their zombie scores. A zombie is a Person created before the given endDate, which has created an average of [0, 1) Messages per month, during the time range between profile’s creationDate and the given endDate. The number of months spans the time range from the creationDate of the profile to the endDate with partial

months on both end counting as one month (e.g. a creationDate of Jan 31 and an endDate of Mar 1 result in 3 months). For each zombie, calculate the following:

- zombieLikeCount: the number of likes received from other zombies.
- totalLikeCount: the total number of likes received.
- zombieScore: zombieLikeCount / totalLikeCount. If the value of totalLikeCount is 0, the zombieScore of the zombie should be 0.0.

For both zombieLikeCount and totalLikeCount, only consider likes received from profiles that were created before the given endDate.

```
// Q21. Zombies in a country
/*
  :param {
    country: 'Ethiopia',
    endDate: 20130101000000000
  }
*/
MATCH (country:Country {name: $country})
WITH
  country,
  $endDate/10000000000000   AS endDateYear,
  $endDate/100000000000%100 AS endDateMonth
MATCH
  (country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(zombie:Person)
OPTIONAL MATCH
  (zombie)<-[:HAS_CREATOR]-(message:Message)
WHERE zombie.creationDate  < $endDate
  AND message.creationDate < $endDate
WITH
  country,
  zombie,
  endDateYear,
  endDateMonth,
  zombie.creationDate/10000000000000   AS zombieCreationYear,
  zombie.creationDate/100000000000%100 AS zombieCreationMonth,
  count(message) AS messageCount
WITH
  country,
  zombie,
  12 * (endDateYear  - zombieCreationYear )
     + (endDateMonth - zombieCreationMonth)
     + 1 AS months,
  messageCount
WHERE messageCount / months < 1
WITH
  country,
  collect(zombie) AS zombies
UNWIND zombies AS zombie
OPTIONAL MATCH
  (zombie)<-[:HAS_CREATOR]-(message:Message)<-[:LIKES]-(likerZombie:Person)
WHERE likerZombie IN zombies
WITH
  zombie,
  count(likerZombie) AS zombieLikeCount
OPTIONAL MATCH
  (zombie)<-[:HAS_CREATOR]-(message:Message)<-[:LIKES]-(likerPerson:Person)
WHERE likerPerson.creationDate < $endDate
WITH
  zombie,
```

```
  zombieLikeCount,
  count(likerPerson) AS totalLikeCount
RETURN
  zombie.id,
  zombieLikeCount,
  totalLikeCount,
  CASE totalLikeCount
    WHEN 0 THEN 0.0
    ELSE zombieLikeCount / toFloat(totalLikeCount)
  END AS zombieScore
ORDER BY
  zombieScore DESC,
  zombie.id ASC
LIMIT 100
```

Consider all pairs of people (person1, person2) such that one is located in a City of Country country1 and the other is located in a City of Country country2. For each City of Country country1, return the highest scoring pair. The score of a pair is defined as the sum of the subscores awarded for the following kinds of interaction. The initial value is score = 0.

1. person1 has created a reply Comment to at least one Message by person2: score += 4
2. person1 has created at least one Message that person2 has created a reply Comment to: score += 1
3. person1 and person2 know each other: score += 15
4. person1 liked at least one Message by person2: score += 10
5. person1 has created at least one Message that was liked by person2: score += 1

Consequently, the maximum score a pair can obtain is: 4 + 1 + 15 + 10 + 1 = 31. To break ties, order by (1) person1.id ascending and (2) person2.id ascending

```
// Q22. International dialog
/*
  :param {
    country1: 'Mexico',
    country2: 'Indonesia'
  }
*/
MATCH
  (country1:Country {name: $country1})<-[:IS_PART_OF]-(city1:City)<-[:IS_LOCATED_IN]-
(person1:Person),
  (country2:Country {name: $country2})<-[:IS_PART_OF]-(city2:City)<-[:IS_LOCATED_IN]-
(person2:Person)
WITH person1, person2, city1, 0 AS score
// subscore 1
OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(c:Comment)-[:REPLY_OF]->(:Message)-
[:HAS_CREATOR]->(person2)
WITH DISTINCT person1, person2, city1, score + (CASE c WHEN null THEN 0 ELSE  4 END) AS
score
// subscore 2
OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(m:Message)<-[:REPLY_OF]-(:Comment)-
[:HAS_CREATOR]->(person2)
WITH DISTINCT person1, person2, city1, score + (CASE m WHEN null THEN 0 ELSE  1 END) AS
score
// subscore 3
OPTIONAL MATCH (person1)-[k:KNOWS]-(person2)
WITH DISTINCT person1, person2, city1, score + (CASE k WHEN null THEN 0 ELSE 15 END) AS
score
// subscore 4
```

```
OPTIONAL MATCH (person1)-[:LIKES]->(m:Message)-[:HAS_CREATOR]->(person2)
WITH DISTINCT person1, person2, city1, score + (CASE m WHEN null THEN 0 ELSE 10 END) AS
score
// subscore 5
OPTIONAL MATCH (person1)<-[:HAS_CREATOR]-(m:Message)<-[:LIKES]-(person2)
WITH DISTINCT person1, person2, city1, score + (CASE m WHEN null THEN 0 ELSE  1 END) AS
score
// preorder
ORDER BY
  city1.name ASC,
  score DESC,
  person1.id ASC,
  person2.id ASC
WITH
  city1,
  // using a list might be faster, but the browser query editor does not like it
  collect({score: score, person1: person1, person2: person2})[0] AS top
RETURN
  top.person1.id,
  top.person2.id,
  city1.name,
  top.score
ORDER BY
  top.score DESC,
  top.person1.id ASC,
  top.person2.id ASC
```

Count the Messages of all residents of a given Country (home), where the message was written abroad. Group the messages by month and destination. A Message was written abroad if it is located in a Country (destination) different than home

```
// Q23. Holiday destinations
/*
  :param { country: 'Egypt' }
*/
MATCH
  (home:Country {name: $country})<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-
  (:Person)<-[:HAS_CREATOR]-(message:Message)-[:IS_LOCATED_IN]->(destination:Country)
WHERE home <> destination
WITH
  message,
  destination,
  message.creationDate/100000000000%100 AS month
RETURN
  count(message) AS messageCount,
  destination.name,
  month
ORDER BY
  messageCount DESC,
  destination.name ASC,
  month ASC
LIMIT 100
```

Find all Messages tagged with a Tag that has the (direct) type of the given tagClass. Count all Messages and their likes grouped by Continent, year, and month.

```
// Q24. Messages by Topic and Continent
/*
  :param { tagClass: 'Single' }
```

```
*/
MATCH (:TagClass {name: $tagClass})<-[:HAS_TYPE]-(:Tag)<-[:HAS_TAG]-(message:Message)
WITH DISTINCT message
MATCH (message)-[:IS_LOCATED_IN]->(:Country)-[:IS_PART_OF]->(continent:Continent)
OPTIONAL MATCH (message)<-[like:LIKES]-(:Person)
WITH
  message,
  message.creationDate/10000000000000   AS year,
  message.creationDate/100000000000%100 AS month,
  like,
  continent
RETURN
  count(DISTINCT message) AS messageCount,
  count(like) AS likeCount,
  year,
  month,
  continent.name
ORDER BY
  year ASC,
  month ASC,
  continent.name DESC
LIMIT 100
```

Given two Persons, find all (unweighted) shortest paths between these two Persons, in the subgraph induced by the knows relationship. Then, for each path calculate a weight. The nodes in the path are Persons, and the weight of a path is the sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated based on their interactions:

- Every direct reply (by one of the Persons) to a Post (by the other Person) contributes 1.0.
- Every direct reply (by one of the Persons) to a Comment (by the other Person) contributes 0.5.

Only consider Messages that were created in a Forum that was created within the timeframe [startDate, endDate]. Note that for Comments, the containing Forum is that of the Post that the comment (transitively) replies to. Return all paths with the Person ids ordered by their weights descending

```
// Q25. Weighted interaction paths
/*
  :param {
    person1Id: 19791209303405,
    person2Id: 19791209308983,
    startDate: 20101031230000000,
    endDate: 20101130230000000
  }
*/
MATCH
  path=allShortestPaths((p1:Person {id: $person1Id})-[:KNOWS*]-(p2:Person {id:
$person2Id}))
UNWIND relationships(path) AS k
WITH
  path,
  startNode(k) AS pA,
  endNode(k) AS pB,
  0 AS relationshipWeights

// case 1, A to B
```

```
// every reply (by one of the Persons) to a Post (by the other Person): 1.0
OPTIONAL MATCH
  (pA)<-[:HAS_CREATOR]-(c:Comment)-[:REPLY_OF]->(post:Post)-[:HAS_CREATOR]->(pB),
  (post)<-[:CONTAINER_OF]-(forum:Forum)
WHERE forum.creationDate >= $startDate AND forum.creationDate <= $endDate
WITH path, pA, pB, relationshipWeights + count(c)*1.0 AS relationshipWeights

// case 2, A to B
// every reply (by ones of the Persons) to a Comment (by the other Person): 0.5
OPTIONAL MATCH
  (pA)<-[:HAS_CREATOR]-(c1:Comment)-[:REPLY_OF]->(c2:Comment)-[:HAS_CREATOR]->(pB),
  (c2)-[:REPLY_OF*]->(:Post)<-[:CONTAINER_OF]-(forum:Forum)
WHERE forum.creationDate >= $startDate AND forum.creationDate <= $endDate
WITH path, pA, pB, relationshipWeights + count(c1)*0.5 AS relationshipWeights

// case 1, B to A
// every reply (by one of the Persons) to a Post (by the other Person): 1.0
OPTIONAL MATCH
  (pB)<-[:HAS_CREATOR]-(c:Comment)-[:REPLY_OF]->(post:Post)-[:HAS_CREATOR]->(pA),
  (post)<-[:CONTAINER_OF]-(forum:Forum)
WHERE forum.creationDate >= $startDate AND forum.creationDate <= $endDate
WITH path, pA, pB, relationshipWeights + count(c)*1.0 AS relationshipWeights

// case 2, B to A
// every reply (by ones of the Persons) to a Comment (by the other Person): 0.5
OPTIONAL MATCH
  (pB)<-[:HAS_CREATOR]-(c1:Comment)-[:REPLY_OF]->(c2:Comment)-[:HAS_CREATOR]->(pA),
  (c2)-[:REPLY_OF*]->(:Post)<-[:CONTAINER_OF]-(forum:Forum)
WHERE forum.creationDate >= $startDate AND forum.creationDate <= $endDate
WITH path, pA, pB, relationshipWeights + count(c1)*0.5 AS relationshipWeights

WITH
  [person IN nodes(path) | person.id] AS personIds,
  sum(relationshipWeights) AS weight

RETURN
  personIds,
  weight
ORDER BY
  weight DESC,
  personIds ASC
```

Updates

Add a Person node, connected to the network by 4 possible edge types

```
MATCH (c:City {id:$cityId})
CREATE (p:Person {id: $personId, firstName: $personFirstName, lastName: $personLastName,
gender: $gender, birthday: $birthday, creationDate: $creationDate, locationIP:
$locationIP, browserUsed: $browserUsed, speaks: $languages, emails: $emails})-
[:IS_LOCATED_IN]->(c)
WITH p, count(*) AS dummy1
UNWIND $tagIds AS tagId
    MATCH (t:Tag {id: tagId})
    CREATE (p)-[:HAS_INTEREST]->(t)
WITH p, count(*) AS dummy2
UNWIND $studyAt AS s
    MATCH (u:Organisation {id: s[0]})
    CREATE (p)-[:STUDY_AT {classYear: s[1]}]->(u)
WITH p, count(*) AS dummy3
UNWIND $workAt AS w
```

```
    MATCH (comp:Organisation {id: w[0]})
    CREATE (p)-[:WORKS_AT {workFrom: w[1]}]->(comp)
```

Add like to post

```
MATCH (person:Person {id:$personId}),(post:Post {id:$postId})
CREATE (person)-[:LIKES {creationDate:$creationDate}]->(post)
```

Add a likes edge to a Comment.

```
MATCH (person:Person {id:$personId}),(comment:Comment {id:$commentId})
CREATE (person)-[:LIKES {creationDate:$creationDate}]->(comment)
```

Add a Forum node, connected to the network by 2 possible edge types.

```
MATCH (p:Person {id: $moderatorPersonId})
CREATE (f:Forum {id: $forumId, title: $forumTitle, creationDate: $creationDate})-
[:HAS_MODERATOR]->(p)
WITH f
UNWIND $tagIds AS tagId
    MATCH (t:Tag {id: tagId})
    CREATE (f)-[:HAS_TAG]->(t)
```

Add a Forum membership edge (hasMember) to a Person.

```
MATCH (f:Forum {id:$forumId}), (p:Person {id:$personId})
CREATE (f)-[:HAS_MEMBER {joinDate:$joinDate}]->(p)
```

Add a Post node to the social network connected by 4 possible edge types (hasCreator, containerOf, isLocatedIn, hasTag).

```
MATCH (author:Person {id: $authorPersonId}), (country:Country {id: $countryId}),
(forum:Forum {id: $forumId})
CREATE (author)<-[:HAS_CREATOR]-(p:Post:Message {id: $postId, creationDate:
$creationDate, locationIP: $locationIP, browserUsed: $browserUsed, content: CASE $content
WHEN '' THEN null ELSE $content END, imageFile: CASE $imageFile WHEN '' THEN null ELSE
$imageFile END, length: $length})<-[:CONTAINER_OF]-(forum), (p)-[:IS_LOCATED_IN]-
>(country)
WITH p
UNWIND $tagIds AS tagId
    MATCH (t:Tag {id: tagId})
    CREATE (p)-[:HAS_TAG]->(t)
```

Add a Comment node replying to a Post/Comment, connected to the network by 4 possible edge types (replyOf, hasCreator, isLocatedIn, hasTag).

```
MATCH
  (author:Person {id: $authorPersonId}),
  (country:Country {id: $countryId}),
  (message:Message {id: $replyToPostId + $replyToCommentId + 1}) // $replyToCommentId is
-1 if the message is a reply to a post and vica versa (see spec)
CREATE (author)<-[:HAS_CREATOR]-(c:Comment:Message {id: $commentId, creationDate:
$creationDate, locationIP: $locationIP, browserUsed: $browserUsed, content: $content,
length: $length})-[:REPLY_OF]->(message), (c)-[:IS_LOCATED_IN]->(country)
WITH c
UNWIND $tagIds AS tagId
```

```
MATCH (t:Tag {id: tagId})
CREATE (c)-[:HAS_TAG]->(t)
```

Add a friendship edge (knows) between two Persons

```
MATCH (p1:Person {id:$person1Id}), (p2:Person {id:$person2Id})
CREATE (p1)-[:KNOWS {creationDate:$creationDate}]->(p2)
```

Algorithms

The time and result will be recorded and compared with other infra.

	DataStax	Execution
PageRank	https://docs.datastax.com/en/dse/6.7/dse-dev/datastax_enterprise/graph/reference/traversal/refTravPageRank.html	
SSSP		
Triangle Counting		
Label Propagation		

Schema changes

A dedicated Java program developed for running schema changes scenario

- Add/remove properties
  - Add random number to all posts
  - And remove the property
- Extract entity from properties(Split nodes)
  - Extract firstName and lastName from Person node to FullName node with name property concatenete firstName and lastName and create relation HAS_FULL_NAME to Person
- Merge nodes
  - Return the FullName node extraction to the original state

Full text search

Besides the ldbc queries full text search feature will be tested. Time and result will be recorded and compared with other infra.

- **TBD**