Interactive Workload

Query definitions were migrated to https://github.com/ldbc/ldbc_snb_docs. Please check that for the latest version.

~~Example Implementations~~

~~For vendor-specific example implementations of these queries refer to the pages below:~~

- ~~Implementation - Virtuoso - SPARQL~~
- ~~Implementation - Virtuoso - SQL~~
- ~~Implementation - Neo4j - Cypher~~

~~Q1 - Friends with certain name~~

- ~~**Description**~~
  - Given a start *Person*, find up to 20 *Persons* with a given first name that the start *Person* is connected to (excluding start *Person*) by at most 3 steps via *Knows* relationships.
    Return *Persons*, including summaries of the *Persons'* workplaces and places of study.
    Sort results by their *distance* from the start *Person*, for *Persons* within the same distance sort by their last name, and for *Persons* with same last name by their identifier
- ~~**Parameter**~~
  - Person.firstName
  - Person.id
- ~~**Result (for each result return)**~~
  - Person.id
  - Person.lastName
  - distance
  - Person.birthday
  - Person.creationDate
  - Person.gender
  - Person.browserUsed
  - Person.locationIP
  - (Person.emails)
  - (Person.language)
  - Person-isLocatedIn->Location.name
  - (Person-studyAt->University.name,
    Person-studyAt->.classYear,
    Person-studyAt->University-isLocatedIn->City.name)
  - (Person-workAt->Company.name,
    Person-workAt->.workFrom,
    Person-workAt->Company-isLocatedIn->City.name)
- ~~**Relevance**~~
  - Choke Points: CP1.7 CP2.3 CP7.1
  - The implementation is to build a hash with the persons whose name matches, then take people within one step, join with the hash, get the names, put in the result. If the result is not yet 20 long, repeat with 2nd degree contacts. If still not 20, continue with 3rd step. Having 20 people or all 3 steps covered, whichever comes first, get the additional details such as places of study, work etc. A declarative query implementation requires reuse of hash join build side and reuse of the 2nd degree friends. The 1st degree friends is a single lookup, hence likely not worth reusing.

    The optimizer is expected not to get stuck in comparing different permutations of single valued attributes, all will be fetched and order does not matter.
    It is also expected to place the functionally dependent (on the person) scalar subqueries after the top k, as these do not enter into the sort and do not change cardinality.
    Interesting mostly for throughput.

~~Q2 - Recent posts and comments by your friends~~

- ~~**Description**~~
  - Given a start *Person*, find (most recent) *Posts* and *Comments* from all of that *Person's* friends, that were created before (and including) a given date.
    Return the top 20 *Posts*/*Comments*, and the *Person* that created each of them.
    Sort results descending by creation date, and then ascending by *Post* identifier.
- ~~**Parameter**~~
  - Person.id
  - maxDate
- ~~**Result (for each result return)**~~
  - Person.id
  - Person.firstName
  - Person.lastName
  - Post.id/Comment.id
  - Post.content/Post.imageFile/Comment.content
  - Post.creationDate/Comment.creationDate
- ~~**Relevance**~~
  - This exercises random lookup with limited locality. Posts are numerous and may be stored in an order that is correlated with their time. Some implementations may use a multi-part key to materialize a time order of posts.

~~Q3 - Friends and friends of friends that have been to countries X and Y~~

- ~~**Description**~~
  - Given a start *Person*, find *Persons* that are their friends and friends of friends (excluding start *Person*) that have made *Posts*/*Comments* in the given *Countries* X and Y within a given period.
    Only *Persons* that are foreign to *Countries* X and Y are considered, that is *Persons* whose *Location* is not *Country* X or *Country* Y.
    Return top 20 *Persons*, and their *Post*/*Comment* counts.
    Sort results descending by total number of *Posts*/*Comments*, and then ascending by *Person* identifier.
- ~~**Parameter**~~
  - Person.id
  - CountryX.name
  - CountryY.name
  - startDate // The beginning of the requested period
  - duration // The duration of the requested period, in days
- ~~**Result (for each result return)**~~
  - Person.id
  - Person.firstName
  - Person.lastName
  - countx // number of Post/Comment from Country X made by Person within the given time interval
  - county // number of Post/Comment from Country Y made by Person within the given time interval
  - count // countx + county
- ~~**Relevance**~~
  - Choke Points: CP3.3
  - This allows exploiting of geographical correlations.
    If one country is large but anti-correlated with the country of self then processing this before a smaller but positively correlated country can be beneficial.

~~Q4 - New topics~~

- ~~**Description**~~
  - Given a start *Person*, find *Tags* that are attached to *Posts* that were created by that *Person's* friends.
    Only include *Tags* that were attached to *Posts* created within a given time interval, and that were never attached to *Posts* created before this interval.
    Return top 10 *Tags*, and the count of *Posts*, which were created within the given time interval, that this *Tag* was attached to.
    Sort results descending by *Post* count, and then ascending by *Tag* name.
- ~~**Parameter**~~
  - Person.id
  - startDate
  - duration // number of days
- ~~**Result (for each result return)**~~
  - Tag.name
  - count // number of Posts made within the given time interval that have this Tag

- **~~Relevance~~**
  - Choke Points: CP2.7
  - If the intervals are short, a hash join may be preferable. If these are long, building a hash of posts for the interval is prohibitive and index will be much better. For the negative interval, building a hash of the distinct tags in the interval is probably best but for this the system needs to know that the hash is used in an anti-join/semijoin, i.e. duplicates need not be kept and that the distinct tags are much fewer than distinct posts.

~~Q5 - New groups~~

- **~~Description~~**
  - 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.
    Return top 20 *Forums*, and the number of *Posts* in each *Forum* that was *Created* by any of these *Persons*.
    Sort results descending by the count of *Posts*, and then ascending by *Forum* name.
- **~~Parameter~~**
  - Person.id
  - minDate
- **~~Result (for each result return)~~**
  - Forum.title
  - count // number of Posts made in Forum that were created by friends
- **~~Relevance~~**
  - What are the groups that your connections (your friends and friends of your friends, excluding you) have joined (hasMember relation) after a certain date order them descending by the number of posts and comments (the total sum of them) they made there, and then ascending by group URI. Group and forum must be the same entity.

~~Q6 - Tag co-occurrence~~

- **~~Description~~**
  - 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 this *Tag*.
    Sort results descending by count, and then ascending by *Tag* name.
- **~~Parameter~~**
  - Person.id
  - Tag.name
- **~~Result (for each result return)~~**
  ~~Relevance~~
  - Tag.name
  - count // number of Posts that were created by friends and friends of friends, which contain this Tag

~~Q7 - Recent likes~~

- **~~Description~~**
  - Given a start *Person*, find (most recent) *Likes* on any of start *Person's Posts*/*Comments*.
    Return top 20 *Persons* that *Liked* your *Post*/*Comment*, the *Post*/*Comment* they liked, the *Like*, and the latency between creation of *Post*/*Comment* and *Like*. Additionally, return a flag indicating whether the liker is a friend of start *Person*.
    Sort results descending by creation time of *Like*, and then ascending by *Person* identifier of liker.
- **~~Parameter~~**
  - Person.id
- **~~Result (for each result return)~~**
  - Person.id
  - Person.firstName
  - Person.lastName
  - Like.creationDate
  - Post.id/Comment.id
  - Post.content/Post.imageFile/Comment.content
  - latency // duration between creation of Post/Comment and Like, in minutes
  - isNew // 0 if liker Person is friend of start Person, 1 otherwise
- **~~Relevance~~**
  - There is both widely scattered indexed access and a clear opportunity for hash join in the existence subquery. Much of the content is accessed only for the top 20 selected posts, hence there is opportunity for late projection.

~~Q8 - Recent replies~~

- **~~Description~~**

    - Given a start *Person*, find (most recent) Comments that are Replies to Posts/Comments of the start Person.
      Return the top 20 reply Comments, and the Person that created each reply Comment.
      Sort results descending by creation date of reply Comment, and then ascending by identifier of reply Comment.
- ~~**Parameter**~~
    - Person.id
- ~~**Result (for each result return)Relevance**~~
    - Person.id
    - Person.firstName
    - Person.lastName
    - Comment.creationDate
    - Comment.id
    - Comment.content
- ~~**Relevance**~~
    - There is temporal locality between the replies being accessed. Thus the top k order by this can interact with the selection, i.e. do not consider older posts than the 20th oldest seen so far.

~~Q9 - Recent posts and comments by friends or friends of friends~~

- ~~**Description**~~
    - Given a start *Person*, find the (most recent) Posts/Comments created by that *Person's* friends or friends of friends (excluding start *Person*).
      Only consider the Posts/Comments created before a given date (excluding that date).
      Return the top 20 Posts/Comments, and the Person that created each of those Posts/Comments.
      Sort results descending by creation date of Post/Comment, and then ascending by Post/Comment identifier.
- ~~**Parameter**~~
    - Person.id
    - maxDate
- ~~**Result (for each result return)**~~
    - Person.id
    - Person.firstName
    - Person.lastName
    - Post.id/Comment.id
    - Post.content/Post.imageFile/Comment.content
    - Post.creationDate/Comment.creationDate
- ~~**Relevance**~~
    - This is a harder variant on Q2 with almost 50x the data. In addition to the Q2 choke points there is CP1.6 for the distinct operator.

~~Q10 - Friend recommendation~~

- ~~**Description**~~
    - Given a start *Person*, find that *Person's* friends of friends (excluding start *Person*, and 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 of these *Persons* and start *Person*, where similarity for any *Pers on* is defined as follows:
      - *common* = number of *Posts* created by that *Person*, such that the *Post* has a *Tag* that start *Person* i s *Interested* in
      - *uncommon* = number of *Posts* created by that *Person*, such that the *Post* has no *Tag* that start *Pers on* is *Interested* in
      - *similarity* = *common* - *uncommon*
      Return top 10 *Persons*, their *Location*, and their *similarity* score.
      Sort results descending by *similarity* score, and then ascending by *Person* identifier
- ~~**Parameter**~~
    - Person.id
    - month1 // between 1-12
    - month2 // month1 + 1, but 12 + 1 = 1
- ~~**Result (for each result return)**~~

    - Person.firstName
    - Person.lastName
    - Person.id
    - Person.gender
    - Person-isLocatedIn->Location.name // City
    - similarity
- ~~**Relevance**~~
    - Choke Points: CP2.7 CP2.6 CP4.2a, CP5 CP6.3, CP7.1
    - The query does widely scattered graph traversal, one expects no locality of in friends of friends, as these have been acquired over a long time and have widely scattered identifiers.

      The join order is simple but one must see that the anti-join for "not in my friends" is better with hash.
      Also the last pattern in the scalar subqueries joining or anti-joining the tags of the candidate's posts to interests of self should be by hash. The rest is quite obviously by index.
      For SQL systems there is possibility for late projection of the first name, last name, gender.
      For RDF systems this is not obvious since the predicates are not guaranteed single valued.
      For RDF systems it is significant to translate the ids of the "dependent columns" (e.g. last name) into strings only after the top k order by.

~~Q11 - Job referral~~

- ~~**Description**~~
    - 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).
      Return top 10 *Persons*, the *Company* they worked at, and the year they started working at that *Company*.
      Sort results ascending by the start date, then ascending by *Person* identifier, and lastly by *Organization* name
- ~~**Parameter**~~
    - Person.id
    - Country.name
    - -workAt->.worksFrom
- ~~**Result (for each result return)**~~
    - Person.id
    - Person.firstName
    - Person.lastName
    - Person-worksAt->.worksFrom
    - Person-worksAt->Organization.name
- ~~**Relevance**~~
    - Choke Points: CP2.7
    - There are selective joins and top k order by.
      There is a pattern on a two-part primary key on the relationship work_at with a condition on the company, on the person as well as n attribute of the relationship, i.e. the start date of the employment.

~~Q12 - Expert search~~

- ~~**Description**~~
    - Given a start *Person*, find the *Comments* that this *Person's* friends made in reply to *Posts*.
      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.
      Return top 20 *Persons*, the reply count, and the collection of *Tags*.
      Sort results descending by *Comment* count, and then ascending by *Person* identifier
- ~~**Parameter**~~
    - Person.id
    - TagClass.name
- ~~**Result (for each result return)**~~
    - Person.id
    - Person.firstName
    - Person.lastName
    - {Tag.name}
    - count // number of reply Comments
- ~~**Relevance**~~
    - Choke Points: CP7.2 CP7.3
    - The chain from original post to the reply is transitive.

The traversal may be initiated at either end, the system may note that this is a tree, hence leaf to root is always best. Additionally, a hash table can be built from either end, e.g. from the friends of self, from the tags in the category, from the or other.

~~Q13 - Single shortest path~~

- ~~**Description**~~
  - 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.
- ~~**Parameter**~~
  - Person.id // first Person
  - Person.id // second Person
- ~~**Result (for each result return)**~~
  - length
- ~~**Relevance**~~
  - The query measures graph traversal and generally presupposes some built-in support for bidirectional shortest path. The performance of the shortest path implementation is measured, along with index-based access of posts.

~~Q14 - Weighted paths~~

- ~~**Description**~~
  - Given two *Persons*, find all weighted paths of the shortest length between these two *Persons* in the subgraph induced by the *Knows* relationship.
    The nodes in the path are *Persons*. Weight of a path is sum of weights between every pair of consecutive *Person* nodes in the path.
    The weight for a pair of *Persons* is calculated such that every reply (by one of the *Persons*) to a *Post* (by the other *Person*) contributes 1.0, and every reply (by ones of the *Persons*) to a *Comment* (by the other *Person*) contributes 0.5.
    Return all the paths with shortest length, and their weights.
    Sort results descending by path weight.
- ~~**Parameter**~~
  - Person.id // person 1
  - Person.id // person 2
- ~~**Result (for each result return)**~~
  - [Person.id] // Identifiers representing an ordered sequence of the Persons in the path
  - weight
- ~~**Relevance**~~
  - The query measures graph traversal and generally presupposes some built-in support for bidirectional shortest path. The performance of the shortest path implementation is measured, along with index-based access of posts.
  - Implementations are not allowed to materialize connection weights for this query. Implementations may deploy optimizations taking advantage of the fact that many of the edges on the resulting paths may occur on several paths.
  - Implementations may vary in the presentation of results but the result set must contain all intermediate vertex id's on each path, the weight of each individual edge on each path, as well as the sum of weights for the path.

~~S1 - Person Profile~~

- ~~**Description**~~
  - Given a start Person, retrieve their first name, last name, birthday, IP address, browser, and city of residence.
- ~~**Parameter**~~
  - Person.id
- ~~**Result (1 result)**~~
  - Person.firstName
  - Person.lastName
  - Person.birthday
  - Person.locationIp
  - Person.browserUsed
  - Person-isLocatedIn->Place.id
  - Person.gender
  - Person.creationDate

~~S2 - Person Recent Messages~~

- ~~**Description**~~
  - Given a start Person, retrieve the last 10 Messages (Posts or Comments) created by that user.
  - For each message, return that message, the original post in its conversation, and the author of that post.
  - Order results descending by message creation date, then descending by message identifier
- ~~**Parameter**~~

    - Person.id
- **Result (for each result return)**
    - Message.id
    - Message.content OR Post.imageFile
    - Message.creationDate
    - Message-replyOf*->Post.id
    - Message-replyOf*->Post-hasCreator->Person.id
    - Message-replyOf*->Post-hasCreator->Person.firstName
    - Message-replyOf*->Post-hasCreator->Person.lastName

S3 - Person Friends

- **Description**
    - Given a start Person, retrieve all of their friends, and the date at which they became friends.
    - Order results descending by friendship creation date, then ascending by friend identifier
- **Parameter**
    - Person.id
- **Result (for each result return)**
    - Person.id
    - Person.firstName
    - Person.lastName
    - Knows.creationDate

S4 - Message Content

- **Description**
    - Given a Message (Post or Comment), retrieve its content and creation date.
- **Parameter**
    - Message.id
- **Result (1 result)**
    - Message.creationDate
    - Message.content OR Message.imageFile

S5 - Message Creator

- **Description**
    - Given a Message (Post or Comment), retrieve its author.
- **Parameter**
    - Message.id
- **Result (1 result)**
    - Message-hasCreator->Person.id
    - Message-hasCreator->Person.firstName
    - Message-hasCreator->Person.lastName

S6 - Message Forum

- **Description**
    - Given a Message (Post or Comment), retrieve the Forum that contains it and the Person that moderates that forum.
    - As comments are not directly contained in Forums, for Comments return the Forum containing the original Post of the thread which the Comment is replying to.
- **Parameter**
    - Message.id
- **Result (1 result)**
    - Message<-containerOf-Forum.id
    - Message<-containerOf-Forum.title
    - Message<-containerOf-Forum-hasModerator->Person.id
    - Message<-containerOf-Forum-hasModerator->Person.firstName
    - Message<-containerOf-Forum-hasModerator->Person.lastName

S7 - Message Replies

- **Description**
    - Given a Message (Post or Comment), retrieve the (1-hop) Comments that reply to it.
    - In addition, return a boolean flag 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
    - Order results descending by comment identifier, then ascending by author identifier

- ~~**Parameter**~~
  - Message.id
- ~~**Result (for each result return)**~~
  - Message<-replyOf-Comment.id
  - Message<-replyOf-Comment.content
  - Message<-replyOf-Comment.creationDate
  - Message-hasCreator-Person.id
  - Message-hasCreator-Person.firstName
  - Message-hasCreator-Person.lastName
  - Boolean: "original message author knows reply author"

~~Categorization~~

		I1	I2	I3.1	I3.2	I3.3	I3.4	I4	I5	I6	I7	I8	I9	I10	I11	I12
PROJECTION	node attr	*		*	*	*	*	*		*	*	*	*	*	*	*
	edge attr														*	
	GROUP_COUNT	*														
	subqueries	*					*							*		
	expr						sum							sub		
	count(1)						*	*		*	*		*	*		*
	as						*							*		
SUBQUERIES													*	*		
FILTER	node attr			*	*	*	*	*			*			*		
	edge attr														*	
	aggr						*									
	exists			*	*	*		*			*	*		*		*
	not							*			*			*		*
	and			*	*	*	*	*			*					
	or											*				
	comparison				*	*	*	*						*	*	
	between							*								
	if													*		
TRAVERSAL	k-hop	3		2	2	2	2	2		2	2	2	1	2	2	3
	kleen star											*				
SET	UNION			*	*	*	*			*					*	
	MINUS													*		
GROUP_BY		*					*	*		*	*		*			*
ORDER_BY		A					D	D		D	D	A	D	D	A	D
LIMIT		*					*	*		*	*	*	*	*	*	*
Virtuoso	bif:concat	*														
	bif:dateadd					*	*									
	bif:month													*		
	bif:dayofmonth													*		