LDBC

Cooperative Project

FP7 – 317548

Social Network Benchmark (SNB) Task Force Progress Report

Coordinator: [Norbert Martínez (Sparsity)]

With contributions from: [Peter Boncz (VUA), Josep Lluís Larriba (UPC), Renzo Angles (VUA), Alex Averbuch (NEO), Orri Erling (OGL), Andrey Gubichev (TUM), Mirko Spasić (OGL), Arnau Prat (UPC)]

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M12
Actual delivery date:	M13
Version:	0.2
Total number of pages:	56
Keywords:	benchmark, choke points, dataset generator, graph database, query set, RDF, task force, technical user community, workload

Abstract

The Social Network Benchmark (SNB) is a LDBC benchmark effort intended to test various functionalities of systems used for graph data management. For this, the recognizable scenario of using graph-shaped data in a large social network is used.

The SNB consists of three sub-benchmarks, or workloads, that focus on different functionalities. In this report, most attention is put into the Interactive Workload, which contains small lookup queries. The other workloads, which are in preliminary stage of development only, are the Business Intelligence Workloads (with analytical queries), and the Graph Analytics Workload (with graph algorithms).

The first goal of the Task Force was to develop the Social Network Benchmark dataset generator used to produce synthetic data (directed labeled graphs) that mimic the characteristics of the real data. This dataset generator is to be used by all three workloads, and therefore its design is being influenced by all three workloads. This task has been supported by the Technical User Community (TUC) input and feedback.

The second goal has been the formulation of the Interactive Workload. In order to arrive at a well-chosen set of queries, so called "choke point" analysis has been carried out using the expertise of database architects to identify the most important technical challenges that should be tested by the queries. This resulting Interactive SNB workload has been also executed and analyzed in several of the vendor's platforms.

The document further contains an explanation of some of the open issues in the design and the necessary steps to complete the remaining workloads in the SNB.

Document Information

IST Project Number	FP7 – 317548	Acronym	LDBC
Full Title	LDBC		
Project URL	http://www.ldbc.eu/		
Document URL	`https://svn.sti2.at/ldbc/trunk/TaskForces/sib/sib_report_1st_year.tex/`		
EU Project Officer	Carola Carstens		

Deliverable	Number		Title	Social Network Benchmark (SNB) Task Force Progress Report
Work Package	Number	N/A	Title	Social Interactive Benchmark Task Force

Date of Delivery	Contractual	M12	Actual	M13
Status	version 0.2		final ☐	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

Authors (Partner)	Norbert Martínez (Sparsity)			
Responsible Author	Name	Norbert Martínez	E-mail	norbert@sparsity-technologies.com
	Partner	Sparsity	Phone	+34934010967

Abstract (for dissemination)	The Social Network Benchmark (SNB) is a LDBC benchmark effort intended to test various functionalities of systems used for graph data management. For this, the recognizable scenario of using graph-shaped data in a large social network is used. The SNB consists of three sub-benchmarks, or workloads, that focus on different functionalities. In this report, most attention is put into the Interactive Workload, which contains small lookup queries. The other workloads, which are in preliminary stage of development only, are the Business Intelligence Workloads (with analytical queries), and the Graph Analytics Workload (with graph algorithms). The first goal of the Task Force was to develop the Social Network Benchmark dataset generator used to produce synthetic data (directed labeled graphs) that mimic the characteristics of the real data. This dataset generator is to be used by all three workloads, and therefore its design is being influenced by all three workloads. This task has been supported by the Technical User Community (TUC) input and feedback. The second goal has been the formulation of the Interactive Workload. In order to arrive at a well-chosen set of queries, so called "choke point" analysis has been carried out using the expertise of database architects to identify the most important technical challenges that should be tested by the queries. This resulting Interactive SNB workload has been also executed and analyzed in several of the vendor's platforms. The document further contains an explanation of some of the open issues in the design and the necessary steps to complete the remaining workloads in the SNB.
Keywords	benchmark, choke points, dataset generator, graph database, query set, RDF, task force, technical user community, workload

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
17/09/2013	0.1	Norbert Martinez	First draft
17/11/2013	0.2	Norbert Martinez	Second draft for final review

Table of Contents

List of Tables

1 Introduction

1.1 Motivation for the benchmark

SNB is intended to be a plausible look-alike of all aspects of operating a social network site, from the online transactions to business intelligence and graph analytics. SNB aims at capturing the essential features of these usage scenarios while abstracting away details of specific business deployments. All the SNB scenarios share a common scalable synthetic data set. The data generation is specially designed to reproduce features of real data, including realistic data distributions and correlations.

The scenario of the benchmark is chosen with the following goals in mind:

- it should be understandable to a large audience, and this audience should also understand the relevance of managing such data
- the scenario in the benchmark should cover the complete range of interesting challenges, according to the benchmark scope
- the query challenges in it should be realistic in the sense that, though synthetic, similar data and workloads are encountered in practice

1.2 Relevance to industry

SNB is intended to provide the following value to different stakeholders:

- For **end users** facing graph processing tasks, SNB provides a recognizable scenario against which it is possible to compare merits of different products and technologies. By covering a wide variety of scales and price points, SNB can serve as an aid to technology selection.
- For **vendors** of graph database technology, SNB provides a checklist of features and performance characteristics that helps in product positioning and can serve to guide new development.
- For **researchers**, both industrial and academic, the SNB dataset and workload provide interesting challenges in multiple choke-point areas, such as query optimization, (distributed) graph analysis, transactional throughput, and provides a way to objectively compare the effectiveness and efficiency of new and existing technology in these areas.

The technological scope of the SNB comprises all systems that one might conceivably use to perform social network data management tasks:

- **Graph database systems** (e.g. neo4j, InfiniteGraph, DEX, Titan)
  - these systems store directed, labeled graphs; and support traversals via APIs
  - they often also support a query language (e.g. Gremlin, Cypher), but this may not be the primary interface
  - queries/programs/tasks programmed against the graph data often involve updating a state specific to the task attached to potentially all nodes/edges
  - these systems often support value-based indexes to quickly locate nodes/edges
  - these systems often support transactional queries, with some degree of consistency
  - typically single-machine architecture (non-cluster)

- **Graph programming frameworks** (e.g. Giraph, Signal/Collect, Graphlab, Green Marl)
  - the core of this system is a language/API that allow to create graph manipulations focused on successive actions on sets of nodes, executing in parallel or lockstep
  - these systems often interface (or take the form of a library) inside a programming language, such that graph manipulation steps and custom logic are intertwined
  - these frameworks typically target global graph computations
    - with long latency
    - involving many graph nodes/edges
    - which often compute approximation answers of problems that are often NP-complete
  - both single-machine and cluster systems exist
- **RDF database systems** (e.g. OWLIM, Virtuoso, BigData, Jena TDB, Stardog, Allegrograph)
  - these systems implement the SPARQL1.1 query language similar in complexity to SQL1992, which allows for structured queries, and simple traversals
  - RDF database system often come with additional support for simple reasoning (sameAs,subClass) , text search and geospatial predicates
  - RDF database systems generally support transactions, but not always with full concurrency and serializability
  - RDF database systems supposed strength is integrating multiple data sources (e.g. DBpedia)
  - both single-machine and cluster systems exist
- **Relational database systems** (e.g. Postgres, MySQL, Oracle, DB2, SQLserver, Virtuoso, MonetDB, Vectorwise, Vertica, but also Hive and Impala)
  - data is relational, and queries are formulated in SQL and/or PL/SQL
  - both single-machine and cluster systems exist
  - relational systems do not normally support recursion, or stateful recursive algorithms, which makes them not at home in the graph analytics workloads
- **noSQL database systems** (e.g. key-value stores such as HBase, REDIS, MongoDB, CouchDB, or even MapReduce systems like Hadoop and Pig).
  - all these systems are cluster-vbased and scalable.
  - the key-value stores could possibly implement the Interactive Workload, though its navigational aspects would pose some problems as potentially many key-value lookups are needed.
  - the MapReduce systems could be suited for the Graph Analytics workload.
  - Pure MapReduce would probably have query latency that is so high that the Business Intelligence workload would not make sense, though we note that some of the key-value stores (e.g. MongoDB) provide a MapReduce query functionality on the data that it stores which could make it suited for the BI workload.

We can further add to this list noSQL database system (HBase, Redis, MongoDB, CouchDB, etc), though such systems would most likely be restricted to handling the Interactive Workload. Those systems with a MapReduce based query functionality could possibly also

1.3 Processes

The LDBC is organized in three platforms:

- The *board* of LDBC members
- *Technical User Community* (TUC): an open organization that brings together users of graph and RDF technologies, researchers, industry participants, and delegates of the LDBC members in physical events (TUC meetings) to discuss about possible benchmark use cases and scenarios and to assess the quality of the benchmark proposals and the adequacy to their needs.
- *Task Forces*: an internal LDBC structure that carries the development of a benchmark from beginning to end.

The benchmark development is initiated by board decision, developed by a task force, and supported by TUC input and feedback. The result, a benchmark specification, consists of both textual documentation and - insofar possible - a standard implementation (i.e. data generator, workload generator and test driver). LDBC software is open source, and is disseminated through GitHub (see: https://github.com/ldbc). The development of a benchmark results in the creation of four main elements:

1. the *data schema*, which defines the structure of the data used by the benchmark
2. the *workload*, which defines the set of operations that the system under benchmarking has to perform during the benchmark execution;
3. *performance metrics*, which are used to measure (quantitatively) the performance of the systems
4. *execution rules*, which are defined to assure that the results from different executions of the benchmark are valid and comparable.

The SNB Task Force is composed by five of the LDBC Consortium members. The roles of each one[1] during this first year has been:

- **OGL:** identification and analysis of choke points for relational and RDF; design of the interactive query set; BIBM query driver based on BSBM; validation scale factor and parameters for the interactive query set; performance analysis of the interactive query set in Virtuoso.
- **NEO:** leading of the methodology for query specification; design and specification of the graph schema; validation tool for the synthetic datasets generated by the dbgen; query driver for graph databases based on the YCSB platform.
- **TUM:** identification and analysis of choke points for relational, RDF and graph databases.
- **UPC:** leading of the task force; leading of the construction of the dbgen tool based on the S3G2 generator (CWI); design and specification of the graph schema; analytical tool to measure the synthetic graph generated by dbgen; validation of the interactive query set using the DEX graph database engine.
- **VUA:** leading of the methodology, design and specification of the graph schema; methodology for query specification; validation of the SPARQL queries of the interactive query set; measurement and adjust of the generator correlations and distributions.

[1] In lexicographical order and not by importance.

1.4 Output values

The SNB is in fact three distinct benchmarks with a common dataset, since there are three different workloads. Each workload produces a single metric for performance at the given scale and a price/performance metric at the scale. The full disclosure further breaks down the composition of the metric into its constituent parts, e.g. single query execution times.

- **Interactive Workload**. The Interactive SNB workload tests a system's throughput with relatively simple queries with concurrent updates.
  - The workload test ACID features and scalability in an online operational setting.
  - The system under test (SUT) is expected to run in a steady state, providing durable storage with smooth response times.
  - Updates are typically small, affecting a few nodes at a time, e.g. uploading of a post and its tags.
  - Transactions may require serializability, e.g. verifying that something does not exist before committing the transaction.
  - Updates will conflict a small percentage of the time.
  - Reads do not typically require more than read committed isolation.
  - Queries typically touch a small fraction of the database, up to hundreds of thousands of values.
  - The count of values accessed by the queries does not significantly increase as the database is scaled up.
- **Business Intelligence Workload**. The workload consists of complex structured queries for analyzing online behavior of users for marketing purposes.
  - The workload stresses query execution and optimization.
  - Queries typically touch a large fraction of the data and do not require repeatable read.
  - The queries are concurrent with trickle load.
  - The queries touch more data as the database grows.
- **Graph Analytics Workload**. This workload tests the functionality and scalability of the SUT for graph analytics that typically cannot be expressed in a query language.
  - The analytics is done on most of the data in the graph as a single operation.
  - The analysis itself produces large intermediate results.
  - The analysis is not expected to be transactional or to have isolation from possible concurrent updates.

Thus, each different system or platform will support the following workloads:

Table 1.1: Workloads supported by each system

System	Interactive	Business Intelligence	Graph Analytics
Graph databases	Yes	Yes	Maybe
Graph programming frameworks	-	Yes	Yes
RDF databases	Yes	Yes	-
Relational databases	Yes	Yes	Maybe, by keeping state in temporary tables, and using the functional features of PL-SQL
NoSQL Key-Value	Maybe	Maybe	-
NoSQL MapReduce	-	Maybe	Yes

1.5 Participation of industry and academia

The broader dissemination policies include public relations by the LDBC board (e.g. press releases), the periodic progress statements of task forces, and the TUC internal and external (public) information, disseminated via portals. LDBC also organizes events, mainly TUC meetings[2] where LDBC members meet the LDBC audience (IT practitioners, researchers, industry). Two of such TUC meetings were held already in November 2012 and April 2013, with a third TUC meeting coinciding with Neo Technology's Graph-Connect in London (November 19, 2013). Additionally, LDBC sponsors and co-organizes scientific workshops. In 2013, it organized the 1st International workshop on Benchmarking RDF Systems (BeRSys), co-located with ESWC'2013; and the 1st International workshop on Graph Data Management Experiences and Systems (GRADES), co-located with SIGMOD/PODS 2013, as well as the GraphLab 2013 workshop held in San Francisco, an event focusing on graph programming frameworks. Finally, LDBC members publish technical aspects of benchmark development in scientific literature [1, 2, 3, 4].

[2] http://ldbc.eu:8090/display/TUC/Events

2 Development

2.1 Requirements

SNB is designed around the following objectives:

- **Rich coverage**. SNB is intended to cover most demands encountered in operating a social network.
- **Modularity**. Not all products and technologies will fit all workloads. For this reason SNB is broken into parts that can be individually addressed. In this manner SNB stimulates innovation without imposing an overly high threshold for participation.
- **Reasonable implementation cost**. For a product offering relevant functionality, the effort for obtaining initial results with SNB should be small, on the order of days.
- **Relevant selection of challenges**. Benchmarks are known to direct product development in certain directions. SNB is informed by the state of the art in database research so as to offer optimization challenges for years to come while not having a prohibitively high threshold for entry.
- **Reproducibility and documentation of results**. SNB specifies rules for full disclosure of benchmark execution and for auditing of benchmark runs. The benchmarks may be run on any equipment but the exact configuration and price of the hardware and software must be disclosed.

2.2 Choke points

2.2.1 Elements of performance

As detailed in deliverable D3.3.3, when comparing algorithmically equivalent solutions to a problem, performance is generally determined by how well the implementations exploit parallelism and locality.

Parallelism is of two main types, i.e. *thread level parallelism*, where independent instruction streams execute in parallel on different processor cores or threads, and *instruction level parallelism*, where execution of consecutive instructions on the same thread overlaps. For the present context, the latter is most importantly exemplified by automatic prefetching of out of cache memory in out of order execution.

Locality can be either spatial or temporal. Spatial locality is achieved when nearby memory requests are serviced by one physical access. At the CPU level this occurs when two fields of the same structure fall on the same cache line. In a database this may occur when two consecutively needed records fall on the same page. Temporal locality occurs when data in the unit of memory hierarchy, e.g. cache line or database buffer page, is re-accessed soon after its initial access. A linear scan of a large table has high spatial locality and a recurring update of specific hot spots has high temporal locality.

The price of missing locality is increased latency, i.e. the time computation dependent on a data item is blocked. Even within one thread of control, the entire thread need not be wholly blocked by one instance of latency, as there can exist data independent parallelizable interleaved instruction sequences. However, in all other cases, the occurrence of latency blocks the thread.

Graph shaped data generally have less natural spatial locality than for example relational data. Items which are accessed together are generally not stored together. This differs from a typical data warehouse containing a history of business events. Graph workloads are characterized by a predominance of random access with no or unpredictable locality. Access is often navigational, i.e., one only knows the next step on an access path and this step must be completed before the one after this is known. A breadth first traversal may alleviate this by having multiple edges to traverse concurrently, allowing overlapping of access latency.

Different workloads have very different characteristics as concerns parallelism and latency tolerance. Varying the scale of the data will cover a range of different platform types, including commodity servers, clusters of such and shared memory scale-up solutions. Workloads need to stress parallelism and locality, for example under the following conditions:

- *Random lookup followed by lookup* of related items. A very short query has little intrinsic parallelism and little locality, since having an edge between vertices generally does not guarantee locality. A 1:n join does have some possibility of parallel execution, certainly in the event of scale-out where it is critical to issue remote operations in parallel and not in series. Testing throughput under these conditions requires large numbers of concurrent operations. These operations may collectively exhibit locality whereas they do not do so individually.

- *Large graph traversals* that touch several percent of the data can be exercised in different ways:

  1. as a breadth-first traversal with filtering,. e.g. distinct persons 5 hop social environment born between two dates
  2. as a star-schema style scan with selective hash joins.

  Workloads should contain queries that preferentially lend themselves to either style of processing.

- *Graph analytics* operations that typically would be implemented in a graph analytics framework. Even though the graph structure may exhibit arbitrary connectedness the fact of touching most vertices with no restrictions on order of access provides both parallelism and locality. Further, simple per-vertex operations offer opportunities for vectoring.

- *Updates* occur primarily in the following scenarios:

  1. Bulk load of data: it has no isolation requirements
  2. trickle updates, e.g. adding new posts, new connections between persons during a benchmark run. It may require serializability, e.g. checking the absence of an edge before inserting it may have to be atomic, plus durability.
  3. graph analytics runs that have large, possibly database resident intermediate state that needs to be kept between iterations. It requires little isolation as the work may be partitioned into non-overlapping chunks but may require serializability for situations like message combination.

Data placement will most likely play a significant role in lookup performance. Thus scale-out implementations will be incited to co-locate items that are frequently accessed together, e.g. a person and the person's posts. A relational schema may imply co-location by sharing a partitioning column between primary and foreign keys. The matter of data location also impacts single server situations but is less acute there, due to lower latency.

2.2.2 Interactive choke points

CP1 - AggregationPerformance. Performance of aggregate calculations.

Identifier	CP1.2 (QOPT)
Title	Interesting Orders
Description	Apart from clustered indexes providing key order, other operators also preserve or even induce tuple orderings. Sort-based opera- tors create new orderings, typically the probe-side of a hash join conserves its order, etc.

Identifier	CP1.6 (QEXE)
Title	High cardinality group by performance, e.g. partitioning on grouping keys in order to avoid having to add up per thread partial group by's
Description	If an aggregation produces significant numbers of groups, for instance Q15 with 1 million groups at 100GB, with intra query parallelization each thread may make its own partial aggregation. To produce the result, these have to be re-aggregated. In order to avoid this, the tuples entering the aggregation operator may be partitioned by a hash of the grouping key and directed to the appropriate partition. The partition may have its own thread so that only this thread writes the aggregation, hence avoiding costly critical sections. A high cardinality distinct modifier in a query is a special case of this choke point. It is amenable to the same solution with intra query parallelization and partitioning as the group by. We further note that scale-out systems have an extra incentive for partitioning since this will distribute the CPU and memory pressure over multiple machines, yielding better platform utilization and scalability.

Identifier	CP1.7
Title	Complex aggregate performance, e.g. concatenation
Description	Many databases offer user defined aggregates and more complex aggregation operations than the basic count, sum, man and min. For example SPARQL has a standard string concatenation aggregation operator. These types of aggregates are expected to benefit from the same optimizations than the basic built-in ones, for example partitioning.

CP2 - JoinPerformance. Voluminous joins, with or without selections.

Identifier	CP2.3 (QOPT)
Title	Rich Join Order Optimization.
Description	The execution times of different join orders differ by orders of magnitude. Therefore, finding an efficient join order is important, and, in general, requires enumeration of all join orders, e.g., using dynamic programming. The enumeration is complicated by operators that are not freely reorderable like semi, anti, and outer joins. Because of this difficulty most join enumeration algorithms do not enumerate all possible plans, and therefore can miss the optimal join order.

Identifier	CP2.4 (QOPT)
Title	Late Projection.
Description	In column stores, queries where certain columns are only used late in the plan, can typically do better by omitting them from the original table scans, to fetch them later by row-id with a separate scan operator which is joined to the intermediate query result. Late projection does have a trade-off involving locality, since late in the plan the tuples may be in a different order, and scattered I/O in terms of tuples/second is much more expensive than sequential I/O. Late projection specifically makes sense in queries where the late use of these columns happens at a moment where the amount of tuples involves has been considerably reduced; for example after an aggregation with only few unique group-by keys, or a top-N operator.

Identifier	CP2.6 (QEXE)
Title	Unpredictable, widely scattered indexed access pattern (graph walk)
Description	The efficiency of index lookup is very different depending on the locality of keys coming to the indexed access. Techniques like vectoring non-local index access by simply missing the cache in parallel on multiple lookups vectored on the same thread may have high impact. Also detecting absence of locality should turn off any locality dependent optimizations if these have overhead when there is no locality. A graph neighborhood traversal is an example of an operation with random access without predictable locality.

Identifier	CP2.7 (QOPT)
Title	Join type - correct choice of hash vs. index based on cardinality on either side
Description	Specially with stores, where one usually has an index on everything, deciding to use a hash join requires a good estimation of cardinalities on both the probe and build sides. In TPC-H, the use of hash join is almost a foregone conclusion in many cases, since an implementation will usually not even define an index on foreign key columns. There is a break even point between index and hash based plans, depending on the cardinality on the probe and build sides.sides. This choke point tests whether this break-even point is correctly modeled and whether the cost model produces accurate estimates as input to this choice.

CP3 - DataAccessLocality. Non-full-scan access to (correlated) table data.

Identifier	CP3.3 (QOPT)
Title	Detecting Correlation
Description	If an schema rewards creating clustered indexes, the question then is which of the three date columns to use as key. In fact it should not matter which column is used, as range-propagation between correlated attributes of the same table is relatively easy. One way is through creation of multi-attribute histograms after detection of attribute correlation. With MinMax indexes, range-predicates on any column can be translated into qualifying tuple position ranges. If an attribute value is correlated with tuple position, this reduces the area to scan roughly equally to predicate selectivity.

Identifier	CP3.5 (STORAGE)
Title	Dimensional clustering - assigning pk/fk/uri/vertex ids in function of attributes of the entity concerned
Description	A data model where each entity has a unique synthetic identifier, e.g. RDF or graph models, has some choice in assigning a value to this identifier. The properties of the entity being identified may affect this, e.g. type (label), other dependent properties, e.g. geographic location, date, position in a hierarchy etc, depending on the application. Such identifier choice may create locality which in turn improves efficiency of compression or index access.

CP4 - ExpressionCalculation. Efficiency in evaluating (complex)

Identifier	CP4.2a (QOPT)
Title	Common Subexpression Elimination
Description	A basic technique helpful in multiple queries is common subexpression elimination (CSE). CSE should recognize also that average aggregates can be derived afterwards by dividing a SUM by the COUNT when those have been computed.

CP5 - CorrelatedSubqueries. Efficiently handling dependent subqueries.

Identifier	CP5.1 (QOPT)
Title	Flattening Subqueries (into join plans)
Description	Many queries have correlated subqueries and their query plans can be flattened, such that the correlated subquery is handled using an equi-join, outer-join or anti-join. To execute queries well, systems need to flatten both subqueries, the first into an equi-join plan, the second into an anti-join plan. Therefore, the execution layer of the database system will benefit from implementing these extended join variants. The ill effects of repetitive tuple-at-a-time subquery execution can also be mitigated in execution systems use vectorized, or block-wise query execution, allowing to run sub-queries with thousands of input parameters instead of one. The ability to look up many keys in an index in one API call, creates the opportunity to benefit from physical locality, if lookup keys exhibit some clustering.

Identifier	CP5.3 (QEXE)
Title	Overlap between Outer- and Subquery
Description	In some queries, the correlated subquery and the outer query have the same joins and selections. In this case, a non-tree, rather DAG-shaped query plan would allow to execute the common parts just once, providing the intermediate result stream to both the outer query and correlated subquery, which higher up in the query plan are joined together (using normal query decorrelation rewrites). As such, the benchmark rewards systems where the optimizer can detect this and whose the execution engine sports an operator that can buffer intermediate results and provide them to multiple parent operators.

CP6 - Parallelism and Concurrency. Making use of parallel computing resources.

Identifier	CP6.3 (QEXE)
Title	Result Re-use
Description	Sometimes with a high number of streams a significant amount of identical queries emerge in the resulting workload. The reason is that certain parameters, as generated by the workload generator, have only a limited amount of parameters bindings. This weakness opens up the possibility of using a query result cache, to eliminate the repetitive part of the workload. A further opportunity that detects even more overlap is the work on recycling, which does not only cache final query results, but also intermediate query results of âĂIJhigh worthâĂİ. Here, worth is a combination of partial-query result size, partial-query evaluation cost, and observed (or estimated) frequency of the partial-query in the workload.

CP7 - RDF and Graph Specifics.

Identifier	CP7.1 (QOPT)
Title	Translation of internal id's into external ones. Translate at point of minimum cardinality, e.g. after top k order by
Description	RDF and possibly graph models often use a synthetic integer identifier for entities, e.g. URI's . For presentation to the client applications, these identifiers must be translated to their original form, e.g. the URI string that was used when loading the data. This should be done as late as possible, or at the point of minimal cardinality in the plan.

Identifier	CP7.2 (QOPT)
Title	Cardinality estimation of transitive paths, order of traversal (1:n, nm:1 n:n)
Description	A transitive path may occur in a 'fact table' or a 'dimension table' position. A transitive path may cover a tree or a graph, e.g. descendants in a geographical hierarchy vs. graph neighborhood or transitive closure in a many-to-many connected social network. In order to decide proper join order and type, the cardinality of the expansion of the transitive path needs to be correctly estimated. This could for example take the form of executing on a sample of the data in the cost model or of gathering special statistics, e.g. the depth and fan-out of a tree. In the case of hierarchical dimensions, e.g. geographic locations or other hierarchical classifications, detecting the cardinality of the transitive path will allow one to go to a star schema plan with scan of a fact table with a selective hash join. Such a plan will be on the other hand very bad for example if the hash table is much larger than the 'fact table' being scanned.

Identifier	CP7.3 (QEXE)
Title	Execution of a transitive step
Description	Graph workloads may have transitive operations, for example finding a shortest path between vertices. This involves repeated execution of a short lookup, often on many values at the same time, while usually having an end condition, e.g. the target vertex being reached or having reached the border of a search going in the opposite direction. For best efficiency, these operations can be merged or tightly coupled to the index operations themselves. Also parallelization may be possible but may need to deal with a global state, e.g. set of visited vertices. There are many possible tradeoffs between generality and performance.

2.3 Social Network Data Generator (SNDG) schema

The data schema of a benchmark defines the structure of the data (used by a benchmark) in terms of *entities*, *relationships* between entities, and *attributes* (defined for entities and relationships). Figure 2.1 presents the UML [1] diagram of the SNDG data schema:

In this UML diagram, a class of entity is represented with a box containing three blocks: the upper block contains the name of the entity class; the middle block holds the attributes[2]; The definition of an attribute includes its name, data type and cardinality. A *relationship* (or *association*) is represented by an edge connecting two entity classes in the diagram, and is labeled with the name of the relationship. Relationships may be directed, and in this case, arrow heads are used to denote the source and target entity of the relationship. A UML class diagram can also record the cardinality of a class in a relationship - a class can participate in zero or one $(0..1)$, one only (1), zero or more $(0..*)$, one or more $(1..*)$ times in a relationship. A relationship can also have attributes (called *attributed relationship*); such attributes provide valuable information about the relationship. Such a relationship is represented by means of an association class.

An instance of a social graph is comprised of *objects* in the graph that are instances of the entities depicted by boxes in the schema shown in Figure 2.2.

The main concept in the SNDG schema is *Person*. Instances of this concept are *avatars* of real world persons and these are created when a person joins the network. The avatars record information regarding the *real world person* this avatar represents, as well as network information. Attributes *firstName*, *lastName*, *gender*, *birthday*, *email* and *speaks* fall into the first category whereas *browserUsed*, *locationIP*, *creationDate* fall into the second. The first connection of the person to the Social Network registers the IP address and the browser used. A person has one or more *email* addresses, *speak* one or more languages that are official or commonly spoken

[1] UML: http://www.uml.org/

[2] UML boxes used to represent classes also contain a third block that stores the methods of the class. This block will not be used since the SNDG Schema does not model methods.

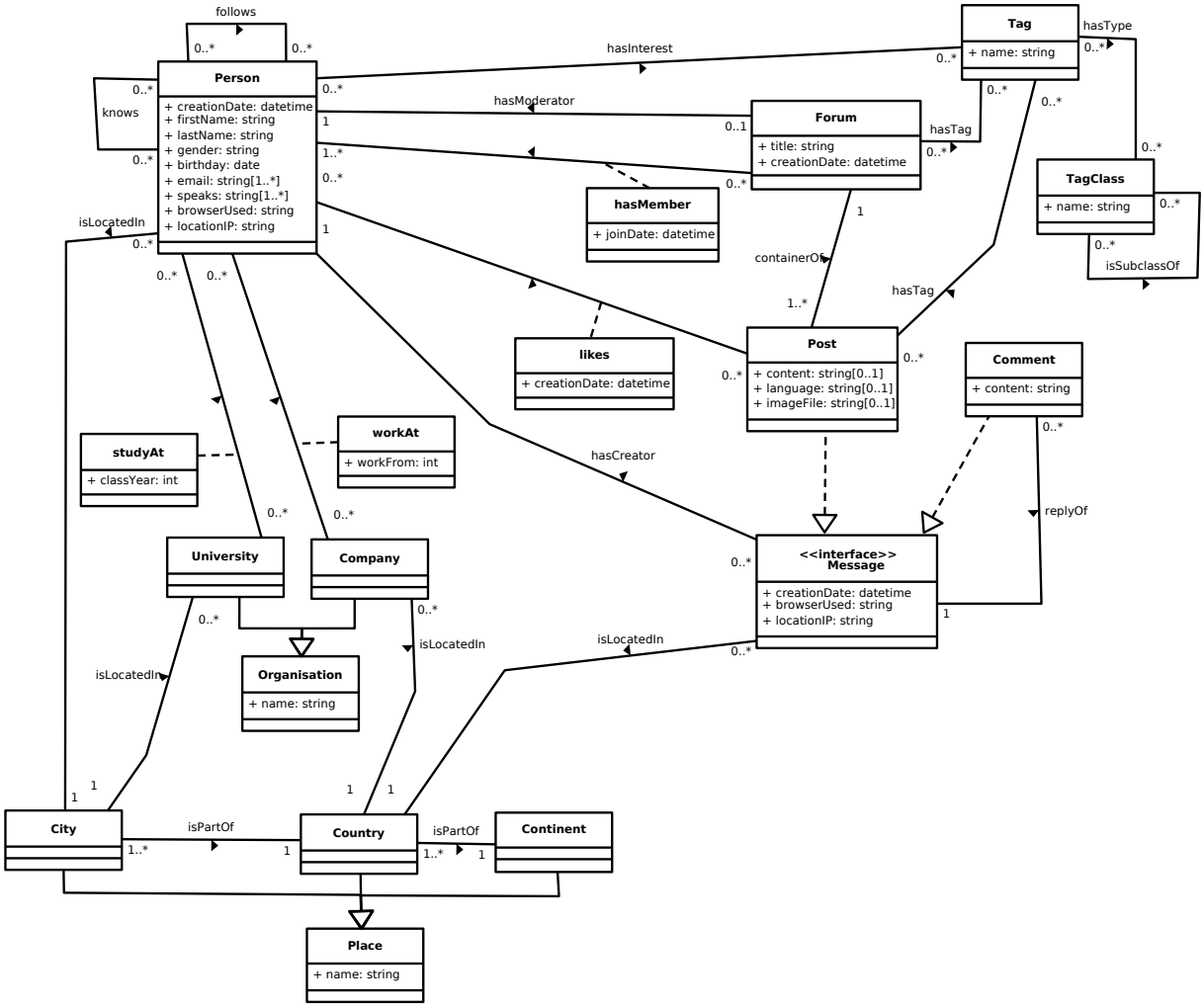

Figure 2.1: Social Intelligence Benchmark Data Schema

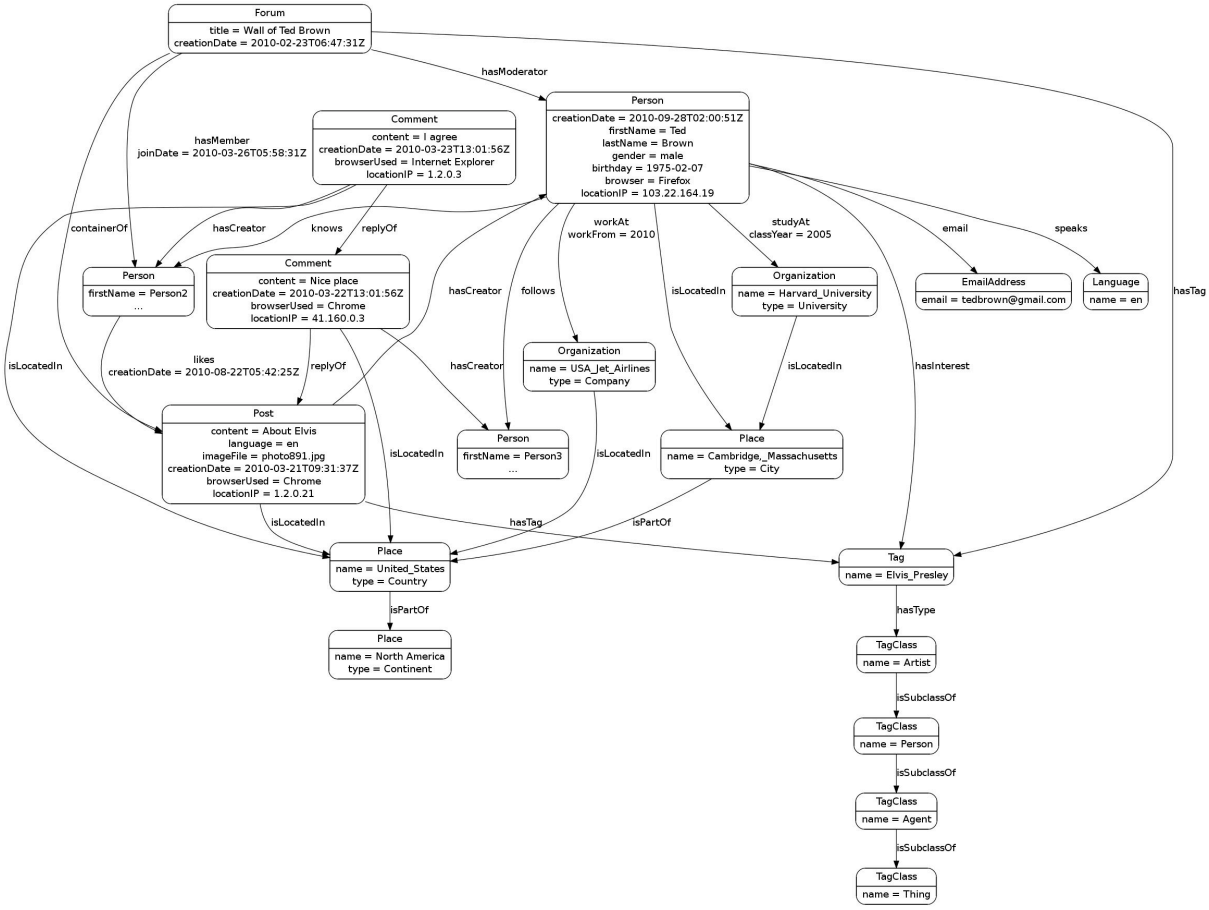

Figure 2.2: An example of SNDG

in her country, or even foreign languages. People in the social graph are connected to others by establishing *friendships*. The facts that a person may know and follow zero or more persons, are recorded by the relationships *knows* (symmetric relationship) and *follows*. Each person is *locatedIn* a city. There is a hierarchy of *places*: concepts *City*, *Country* and *Continent* are defined as sub-concepts of *Place*. The SNDG schema records the regional correlation between the entities using the transitive *partOf* relationship.

A person has studied (relation *studyAt*) and worked (relation *workAt*) at zero or more universities and companies respectively. The former records the graduation year, and the latter the year the person started working for the company. Universities and companies are instances of entities *University* and *Company* respectively, sub-entities of *Organization* and located in (attribute *isLocatedIn*) a city and a country respectively.

A person has zero or more interests (attribute *hasInterest*), each interest (instance of entity *Tag*) can have different types (entity *TagClass*) that form a hierarchy of types. A person is the moderator of a forum (entity *Forum*). A forum can have at most one moderator (this is recorded by attribute *hasModerator*), and can have at least one person as member (indicated by relationship *hasMember*). It also has a title (attribute *title*) and date of creation (attribute *creationDate*). The SNDG schema records the date (attribute *joinDate*) that the person has joined the forum. A forum can be tagged with zero or more tags that record the topics of interest of the forum. A forum might have at least one published post (entity *Post*, attribute *containerOf*). A post has a *content*, is written in some *language* and/or is associated with an image (attribute *imageFile*). Similar to forums, posts are tagged (attribute *tag*) with zero or more tags. Persons *like* zero or more posts - the date the person indicated that she likes the post is recorded using the *creationDate* attribute. People can comment posts where a comment is just a simple text (attribute *content*). Posts and comments are messages and carry information such as the creation date (attribute *creationDate*), the browser used (attribute *browserUsed*) and the IP address (attribute *locationIP*). Comments can reply to zero or more posts and comments (attribute *replyOf*).

2.4 Data generation process

An important component of the Social Network benchmark is the data generator used to produce *synthetic data* that mimic the characteristics of the real data. The Social Network Data Generator is based on the S3G2 data generator [5]. S3G2 (Scalable Structure-correlated Social Graph Generator) targets at modeling huge correlated directed labeled graphs. S3G2 and hence SNDG, use a *graph model* that is essentially a *correlated property graph*. The building blocks of the data generator are *property dictionaries*, *simple subgraph generation*, and *edge generation* along *correlation dimensions*.

2.4.1 Property dictionary

Property values for each literal property are generated following a *property dictionary model* that is defined by *i)* a *dictionary* D, a *ranking function* R and a *probability density function* F.

Dictionary D is a *fixed set of values* (e.g., terms from DBPedia[3], GeoNames[4]). The ranking function R is a bijection that assigns to each value in a dictionary a unique *rank* between 1 and $|D|$. The probability density function specifies how the data generator chooses values from dictionary D using the rank for each term in the dictionary. The idea to have a separate ranking and probability function is motivated by the need of generating correlated values: in particular, the ranking function is typically parameterized by some parameters: different parameter values result in different rankings. For example, in the case of a dictionary of property *firstName*, the popularity of first names, might depend on the *gender*, *country* and *birthDate* properties. Thus, the fact that the popularity of first names in different countries and times is different, is reflected by the different ranks produced by function $R()$ over the full dictionary of names.

The S3G2 data generator contains a dictionary for each literal property, as well as ranking functions for all literal properties (independently of the class they are defined in). It is important that when designing correlation parameters for a ranking function, the amount of parameter combinations (see the example above) stays limited, in order to keep the representation of such functions compact.

[3]DBPedia: `http://el.dbpedia.org/`

[4]`GeoNames:http://www.geonames.org/`

2.4.2 Graph generation

The idea behind a simple graph generation algorithm is to create new edges and nodes in one pass, starting from an existing node n and start creating new nodes to which n will be connected. The degree distribution function is used to guide the above process by determining the number of descendants that a node will have. It has been observed that in a number of social networks, the amount of edges that start from a node, follows a *power law* distribution. In S3G2 it is possible to have such a degree distribution function from which the degree of each node is generated, correlated with properties of the node. For instance, people with many friends in a social network will typically post more pictures than people with few friends (hence, the amount of friend nodes in our use case influences the amount of posted comment and picture nodes in the graph). The disadvantage of this process is that it might lead to isolated graphs that are simply dangling from node n the whole process was initiated from.

The S3G2 data generator generates correlated and highly connected graph data by creating edges *after* generating many nodes. This approach, when compared to a naive graph generation algorithm, is computationally harder than generating edges towards new nodes. The reason is that if node properties influence their connectivity, a naive implementation would have to compare the properties of all existing nodes with all nodes, something that would lead to quadratic computational cost and a random access pattern, so the generation algorithm would only be fast as long as the data fits in RAM (to avoid a random I/O access pattern).

The use of data correlation provides a solution to this problem: the probability that two nodes are connected is typically skewed with respect to some similarity between the nodes. That is, for a node n and for a small set of nodes that are somehow similar to it, there is a high connectivity probability, whereas for most other nodes, this probability is quite low. This knowledge can be exploited by a data generator by identifying correlation dimensions.

For a certain edge label $e \in P^{E(x,y)}$ between node classes O^x and O^y, a correlation dimension $CD_e(M^x, M^y, F)$ consists of two *similarity metric* functions $M^x : n \rightarrow [0, \infty]$, $M^y : n \rightarrow [0, \infty]$, and a probability distribution F :[1,$W.t$]→[0,1], where $W.t$ is a window size, of W tiles with each t nodes. For instance, in the case of friends in a social network, both start and end of the edges are of the class person ($O^x = O^y$), so a single metric function would typically be used. The description below assumes that $M = M^x = M^y$.

The similarity metric is computed by invoking the similarity function M on all nodes; nodes are subsequently sorted on this score. The consequence of this approach is that the similar nodes are grouped together, and consequently, the larger the distance between two nodes indicates a monotonic increase in their similarity difference. In order to choose the nodes to connect, S3G2 uses a geometric probability distribution for F that provides a probability for picking nodes to connect with that are between 1 and $W.t$ positions apart in this similarity ranking [5]. In order for F to be a geometric distribution, S3G2 should not cut short at $W.t$ positions apart, but consider even further apart nodes. The advantage of this window shortcut is that after sorting the data, it allows S3G2 to generate edges using a fully sequential access pattern that needs little RAM resources since it only buffers $W.t$ nodes.

Similarity functions and probability distribution functions over ranked distance drive *what* kind of nodes will be connected with an edge, not *how many*. This decision on the degree of a node is made before generating the edges, using the *degree function* N (recall that in social networks this would be a power-law function). The edges that will be connected to a node n (i.e., the edges that determine the degree of a node), are selected by randomly picking the required number of edges according to the correlated probability distributions as discussed before. In the case that multiple correlations exist, another probability function is used to divide the intended number of edges between the various correlation dimensions. Thus, S3G2 has a a power-law distributed node degree, and a predictable (but not fixed) average split between the reasons for creating edges.

Another parameter that is taken into consideration when generating the graph is *random dimension*. Generating edges between the most similar nodes in all the correlation dimensions is very restrictive: unlikely connections in a social network that the data model would not explain or make plausible, will occur in practice. This random noise is introduced in the generated data by creating a special correlation dimension where a random function can be used as a similarity metric along with a uniform probability function.

2.4.3 Implementation

The S3G2 data generator algorithm is implemented using a MapReduce algorithm that is built on the building blocks discussed earlier: (a) correlated data dictionaries, (b) naive graph generation and (c) edge generation using correlation dimensions. The idea is that a Map function runs on different parts of the input data, in parallel and on many clusters. This function processes the input data, produces for each result a key. Reduce functions then obtain this data and Reducers run in parallel on many cluster machines. The produced key simply determines the Reducer to which the results are sent.

This key is also used to sort the data for which edges will be generated according to the previously discussed similarity metric of the correlation dimension to be used next. Since there are many correlation dimensions, there will be multiple successive MapReduce phases. Map and Reduce functions can perform simple graph generation as previously discussed in addition to generating correlated property values using dictionaries.

The main task of the Reduce function is to (a) sort on the correlation dimension (b) generate edges between nodes using an algorithm that is based on *sliding windows*. The idea behind this approach to correlated edge generation is that when generating edges, only the edges that connect similar (according to the similarity metric) nodes should be considered. By ordering these nodes, the algorithm can keep a small window of nodes (properties and edges) in RAM. Subsequently, only the nodes that are cached in this window are considered for edge generation. Note that different correlations can influence the generation of an edge. To handle this, multiple sorts and sequential sliding window passes are needed, leading to multiple MapReduce jobs.

The sliding window approach is implemented by dividing the sorted nodes conceptually in tiles of t nodes: when a Reducer gets a data item, then it adds it to the tile that is processed (an in memory data structure). When the tile is full and there are W tiles in memory, then the oldest tile is dropped. Before it is dropped, the Reduce function generates edges for all the nodes in the tile, implementing the windowing approach and generating edges along a correlation dimension.

In principle, simple graph generation only requires local information (the current node), and can be performed as a Map task, but also as a post-processing job in the Reduce function. Note that node generation also includes the generation of the (correlated) properties of the new nodes. It comes without saying that data correlations introduce *dependencies*, that impose constraints on the order in which generation tasks have to be performed. So, the correlation rules one introduces, naturally determine the amount of MapReduce jobs needed, as well as the order of actions inside the Map and Reduce functions.

2.4.4 Dataset validation

Due to the complex nature of the data generation it is important that the generated content is validated for correctness, in terms of: data content, data format, graph topology. One of the ways these correctness criteria are maintained is via a tool we have developed, currently named **ldbc_datachecker**[5]. This tool is new and, as such, still lacks certain functionality. However, it already supports the following types of dataset validation:

- File existence: check that all expected files were written to the data generator's output directory
- Configuration parameters: for example, if the parameters specify that the network should contain 2000 `Person` nodes, check that this is so
- Type check: check that values in the generated .csv columns conform to documented types (`int`, `long`, `string`, `finite set`, etc.)
- Data format: performs checks such as no illegal characters in strings, email addresses structured correctly, etc.
- Graph topology: tests inconsistencies in the graph structure such as relationships that do not connect exactly two nodes, cardinality constraints (e.g., only one `KNOWS` relationship exists between any two `Person` nodes), etc.

[5] `https://github.com/ldbc/ldbc_datachecker`

Data generator development and improvements are ongoing, to maintain correctness of the output in the presence of these constant changes it is important that the validation process is fast and easy, and simple to extend in line with future data generator changes. These were the goals when developing the `ldbc_datachecker` too, and will continue to be moving forward.

2.4.5 Graph statistics overview

In Section 2.1.1 of LDBC Deliverable we summarized some properties that often appear in real graphs, and that will be useful to characterize them. Table 2.1 shows a comparison of the expected values for some of these metrics in a real social network, and the measured metrics of a socialnet_dbgen synthetic dataset describing the activities of 100k users during 1 year. In this case, the analysis has been done for *person-[knows]->person*, which is the most important relationship in a social network graph.

Table 2.1: person-[knows]->person sf=100K1Y statistics

Metric	**Description**	**Expected**	**socialnet_dbgen**
community structure	a very large connected component	at least 80% to 90% of the nodes	99.78%
small world property	average path length	smaller than 5 or 6	3.93
degree of transivity of the graph	average clustering coefficient	greater than 0.3	0.11
diameter	largest distance between two nodes	up to 9 or 11	11

There are also two important diagrams that characterize the *knows* relationship: the *hop plot* in Figure 2.3, a visualization of the distribution of pairwise distances that grows between 2 and 5 as expected; and the *edge degree distribution* in Figure 2.4 that shows a power-law distribution, almost linear in a log-log diagram[6].

Table 2.2: Global Statistics

N	80,767,146	number of nodes
E	350,352,746	number of edges
V	500,108,979	number of attribute values

Table 2.3: User-Knows Subgraph Statistics

N	100,000	number of nodes
E	2,887,796	number of edges
N_c	99,778	number of nodes in largest connected component
N_c/N	1.00	fraction of nodes in largest connected component
$\overline{C}$	0.11	average clustering coefficient
D	6	diameter
$\overline{D}$	3.93	average path length

2.5 Query drivers

2.5.1 QGEN - BIBM

BIBM (Business Intelligence Benchmark) is a set of test tools to benchmark and validate relational database management systems (tables and/or property graph variants). It can run BSBM (Berlin SPARQL Benchmark) and TPC-H test suites. This driver has been updated, so it is able to run SNB, as well.

[6]Note that the x-axis has been computed with $log_{10}(degree + 1)$ to represent in a logarithmic scale all the range of degree values

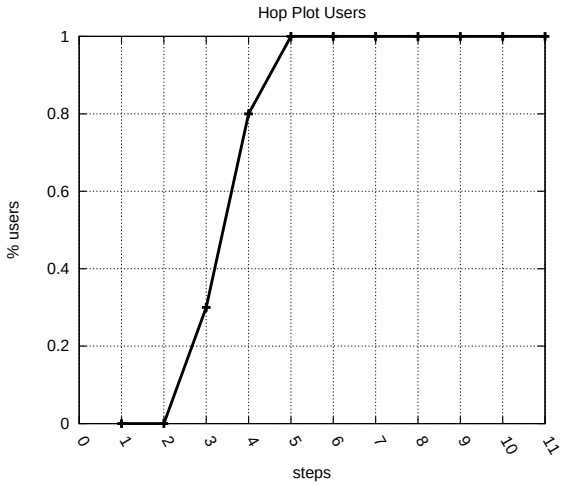

Figure 2.3: Hop-plot of the *knows* relationship

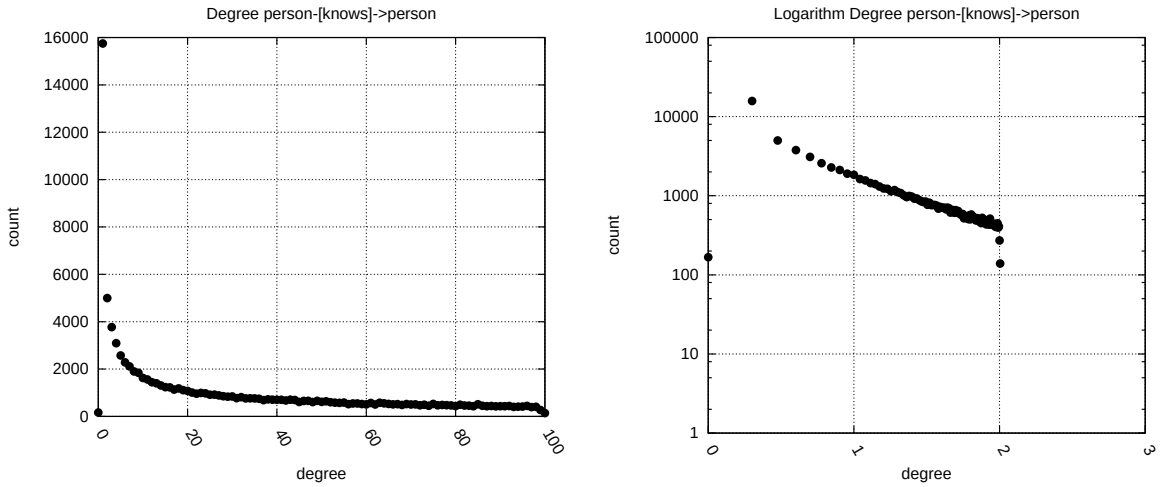

Figure 2.4: Degree distribution of the *knows* relationship

Main driver components

- `TestDriver`: This is the main part of the BIBM. It contains loading configuration files, processing the command line arguments, initialization of the other driver components, launching them, and writing the results into the output files (log files, qualification files, benchmark result files, etc).
- `ClientManager`: As the name suggests, this component is responsible for managing clients (creating them, starting with or without warm-up phase, waiting for them and summarizing the results).
- `ClientThread`: This part is managed by the previous one. It implements one single client. It connects to the server with exposed SPARQL endpoint via the SPARQL protocol, runs all the queries from the current query mix, and collects the results.
- `SIBQueryMix`: This is the collection of the queries, permuted randomly, representing a query mix.
- `SIBParameterPool` and `SIBFormalParameter`: The assignment of these components is to deal with parameters of the queries. The query templates are provided, and their abstract parameters are replaced with the formal ones (random or default) by these components.

- `QueryStatistics` and `QueryMixStatistics`: These components have responsibility of calculating all kinds of statistics for the queries and query mixes (maximum/minimum/average query execution time, maximum/minimum number of returned results, number of timeouts, etc).

Features

Some of the key features of the driver is listed bellow:

- Capability of running multiple clients concurrently
- Generation of the output qualification file
- Comparison of the result sets
- Warm-up runs
- Triggering the profile query views
- Running driver with default query parameters
- Initialization of the driver with specific seed
- Specifying the timeouts in ms

This benchmark is used to simulate a social network site, and running queries should mimic a user browsing this site. A lot of those queries return person URIs, and the expected behavior of user is to be interested in some of them, asking for their details (first and last name, location, friends, etc) by clicking on specific URIs. In order to provide this use case, the driver is capable of triggering the profile queries, for randomly chosen persons that are returned by some of the queries from the initial query mix.

2.5.2 LDBC_DRIVER

The ldbc_driver project started out as a fork of YCSB[7], a popular benchmarking framework for performance comparison of key-value stores. Initially the goal was to generalize YCSB from the key-value model to a more generic model, encompassing the graph model. However, it was soon realized that this would require major changes to YCSB, so the ldbc_driver project continued under its own name - it can no longer be seen as a fork of YCSB as almost every aspect of YCSB has been removed or extensively modified.

Driver components

- `Client`: Loads configuration from command line and files, initializes other main components, initiates the benchmark, provides feedback upon benchmark completion.
- `Operation`: Abstraction that represents the queries/operations executed against the database being benchmarked. An Operation is an abstract class, which serves two purposes: (1) identifies a query, e.g., "LDBC Interactive Query 1" (2) contains the parameters for that query instance.

  As opposed to YCSB, where the set of queries is limited to CRUD operations, the Operation construct allows ldbc_driver to support a much larger range of queries, including imperative (graph) algorithms like Dijkstra, PageRank, etc.
- `Workload`: A programmatic benchmark definition, a Workload creates the stream of Operations to be executed against the database being benchmarked. In addition, it populates these Operations with substitution parameters and assigns to them scheduled start times.

[7] `https://github.com/brianfrankcooper/YCSB/wiki`

- `OperationResult`; Expected format and content for the result of a given Operation, used to validate that a database is returning correct results. Every Operation implementation is associated with an OperationResult implementation.
- `WorkloadRunner`: Among other parameters, WorkloadRunner accepts a Workload definition as input and is responsible for running the Operations generated by that Workload. It coordinates such tasks as thread scheduling, Operation execution, metrics (time, result, etc.) gathering, and logging.
- `Db`: An abstract notion of a database connector. Each vendor must extend the Db class to implement a database connector to their database.
- `OperationHandler`: Contains logic for executing an Operation. Every OperationHandler is associated with one Operation type. An OperationHandler takes as input an Operation instance and executes it against the database being benchmarked. It is the responsibility of a Db implementation to provide OperationHandler implementations for every Operation type in the Workload definition.

Contributions and features

In addition to those already discussed, the key features of ldbc_driver include:

1. Load generation based on *scheduled start times* rather than simple throughput
2. Measurement of its own performance (in addition to Operation latencies)
3. Asynchronous multi-threaded execution

The execution model of ldbc_driver differs from those of YCSB, LinkBench, and many other benchmark drivers in that it supports multi-threaded asynchronous execution.

Most other drivers have a blocking multi-threaded model in which each worker thread gets a queue of tasks/operations and executes those tasks in a blocking fashion, waiting for one to terminate before starting the next. The error in this approach is it allows the database to dictate the maximum throughput, limiting the driver's ability to stress the database.

In contrast, by asynchronously executing operations ldbc_driver never blocks, it simply keeps putting operations on the execution queue when they are ready to be executed. The driver collects a number of time-related metrics (*scheduled start time*, *actual start time*, and *execution duration*) and supports a range of policies for dealing with certain failure conditions. For example, if *actual start time* consistently deviates from *scheduled start time* by more than a given threshold it is an indicator that the benchmark driving machine has insufficient resources, and therefore can not run the benchmark as intended. For such cases ldbc_driver can be instructed to terminate benchmark execution.

2.6 Parameter Generation

A typical benchmark consists of two parts:

1. the data generator that creates syntactic dataset of the desired scale
2. the workload generator that issues queries against the generated data based on the pre-defined *query templates*.

A query template is an expression in the query language (e.g., SPARQL) with *substitution parameters* that have to be replaced with real bindings by the workload generator, for example:

```
select * where {
       ?person sn:firstName %name .
       ?person sn:knows ?friend.
}
```

The goal of the workload generator is to generate a set of parameter bindings per query that would guarantee similar execution times of this query with these parameters. In a nutshell, parameter generation for a specific query becomes a data mining process of clustering the entire domain of parameter values into classes based on the runtime of the query. An imporant aspect of the workload generation is that the workload generator can also modify the dataset (by e.g., removing or inserting tuples) to adjust the parameter clusters if needed. The result of the data and workload generators is the dataset and the set of queries with generated parameters. The dataset is then loaded into the system, and it executes the queries with given parameters.

In this section we consider parameter generation for queries in our benchmark. We demonstrate that conventional uniform sampling is not providing stable and comprehensive benchmark results. We present the formal problem of clustering the parameters domain into classes that yield the same query plan with regards to the sum of intermediate results; the solution of this clustering problem will ensure comparable and stable results of benchmark queries. We are currently working on developing an effective and efficient solution for this problem.

In our example query above, the workload generator would produce a number of different bindings (say, 100) for the `%name` parameter, then execute the resulting queries and report an aggregate value of the observed runtime distribution per query (usually, the average runtime per query template). This aggregated score serves two audiences: First, the users can evaluate how fit a specific system is for their use-case (choosing, for example, between systems that are good in complex analytical processing and those that have the highest throughput for lookup queries). Second, the database architects can use the score to analyze their systems' handling of certain technical challenges. That is the case with the classical TPC-H benchmark, where each query tests the specific system capability like handling multiple interesting orders (Q3,4,18), sparse foreign key joins (Q2 and others) or arithmetic operators (Q1).

In order for the aggregate runtime to be a useful measurement of the system's performance, the selection of parameters should guarantee the following properties of the queries:

P1: the query runtime has a bounded variance: the average runtime should correspond to the behavior of the majority of the queries

P2: the runtime distribution is stable: a different sample of 100 parameter bindings should result in an identical runtime distribution

P3: the runtime distribution is normal: this would allow us to have guarantees on the average runtime computed from 100 query runs

P4: the query plan for all the parameters is the same: this ensures that a specific query tests the system's behavior under the well-chosen technical difficulty (e.g., handling voluminous joins or proper cardinality estimation for subqueries etc.)

The standard way to get the parameter bindings for `%name` is to sample the values (uniformly, at random) from all the possible names in the dataset. This is, for example, how the classical TPC-H benchmark creates its workload; since the TPC-H data is generated with simple uniform distribution of values, the uniform sample of parameters guarantees the properties P1-P4.

However, this technique does not work for RDF benchmarks. First, even the datasets generated with uniform distribution of values still have transitive properties not present in any relational benchmark (like the type hierarchy of products in the BSBM benchmark). Furthermore, the benchmarks that use real-world datasets (YAGO, DBpedia) or generate datasets with real-world distributions and correlations (like our own LDBC SNB) require more sophisticated selection of query parameters.

2.6.1 Why uniform sampling does not work?

In this section we illustrate the statement that uniform selection of parameters leads to unpredictable behavior of the queries, thus making interpretation of the results very difficult.

We use two RDF benchmarks, Berlin SPARQL benchmark (BSBM) and LDBC Social Network benchmark (Interactive Workload). For BSBM, we consider queries from the Business Intelligence (BSBM-BI) use case.

For both BSBM and LDBC, the corresponding data generators were used to create datasets with ca. 100 Million triples each. We employ Virtuoso 7 (Column Store) as a SPARQL query engine, and run our experiments on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 64 Gb RAM, 1 Tb SAS-HD, Redhat Enterprise Linux with 2.5.37 kernel.

E1: Runtime distribution has high variance When drawing parameters uniformly at random, we encounter a very skewed runtime distribution even for queries over uniformly distributed syntactic data. In BSBM-BI benchmark, for example, the runtime of the Query 4 (*finds the feature with the highest ratio between price with that feature and price without that feature*) has the variance of $674 \cdot 10^6$. This is caused by the fact that the parameter of the query is the product type: depending on how generic the type is (how high it is in the type hierarchy), the amount of data touched by the query differs greatly.

This issue becomes even more important for the LDBC benchmark, where the data generator seeks to mimic some of the properties of the real-world data: the generated data has correlations and skewed data distributions. In this case, naturally, the randomly generated parameter bindings result in a very skewed runtime distribution.

Moreover, the distribution of runtime is far from normal: the Kolmogorov-Smirnov test that measures the distance between the runtime distribution of BSBM-BI Query 2 (*finds the top 10 products most similar to a specific product*) and the normal distribution, results in the distance of 0.89 (with p-value of 10^{-21}). This value indicates that the observed runtime distribution is extremely non-uniform.

E2: Sampling is not stable A single query in the benchmark is typically being executed several times with different randomly chosen parameter bindings. It is therefore interesting to see how the reported average time changes when we draw a different sample of parameters. In order to study this, we take Query 2 of the LDBC SN benchmark that *finds the newest 20 posts of the user's friends*. We sample 4 independent groups of parameter bindings (100 bindings in each group), run the query with these parameters and report the aggregated runtime numbers within individual groups (q_{10} and q_{90} are the 10th and the 90th percentiles, respectively).

Time	Group 1	Group 2	Group 3	Group 4
q_{10}	0.14 s	0.07 s	0.08 s	0.09 s
Median	1.33 s	0.75 s	0.78 s	1.04 s
q_{90}	4.18 s	3.41 s	3.63 s	3.07 s
Average	1.80 s	1.33 s	1.53 s	1.30 s

We see that uniform at random generation of query parameters in fact produces unstable results: if we were to run 4 workloads of the same query with 100 different parameters in each workload, the deviation in reported average runtime would be up to 40%, with even stronger deviation on the level of percentiles and median runtime (up to 100%).

We observe similar behavior of the BSBM-BI Query 2: when it is executed with different groups of 100 random parameter binding each, the mean runtime difference is up to 15% between the groups, and the median value can vary up to 25%.

E3: Average runtime is not representative In addition to being far from uniform (**E1**), the query runtime distribution can also be "clustered": depending on the parameter binding, the query runs either extremely fast or surprisingly slow, and the average across the runtimes does not correspond to any actual query performance. We consider Query 4 of the BSBM BI workload that *finds the feature with the highest ratio between price with that feature and price without that feature*, depending on the input parameter *ProductType*. The product types in BSBM form a hierarchy: the higher the type is, the more general it is and therefore the more products with this type exist. A short summary of statistical properties of the runtime distribution is given in the table below (where q_{95} is the 95th percentile):

Min	Median	Mean	q_{95}	Max
59 ms	354 ms	3.6 s	17.6 s	259 s

Depending on the parameter selection, the query finishes in either 300 ms to 400 ms, or in more than 17 seconds, with almost no query in between those two groups. This way, the arithmetic mean is over 10 times

larger than the median. Moreover, there is no actual query with the runtime close to the mean: all of them are either much faster, or significantly slower.

E4: Different plans for different parameters Finally, the uniformly generated parameter bindings can lead to completely different plans for the same query template. It happens because the cardinalities of the subqueries naturally depend on the parameter bindings, and sometimes on the combination of the parameters. For instance, the optimal plan for the LDBC SNB Query 3 (*finds the friends within two steps that have been to countries X and Y*) can start either with finding all the friends within two steps from the given person, or from all the people that have been to countries X and Y: if X and Y are Finland and Zimbabwe, there are supposedly very few people that have been to both, but if X and Y are USA and Canada, this intersection is very large.

We note that the plan variability is not a bad property *per se*: indeed, this query forces the query optimizer to accurately estimate the cardinalities of subqueries depending on input parameters. However, the generated parameters should be sampled independently from two different classes (countries that are rarely and frequently visited together), to allow a fair and complete comparison of different query optimization strategies.

2.6.2 Generating Parameter Bindings

Here we define the formal problem of generating the parameter binding for RDF benchmark. In order to compare two query plans (e.g., optimal plans for the same query with different parameter bindings), we use the classical cost function that takes into account the sum of intermediate results produced during the plan's execution. It is formally defined as follows:

$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a scan} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

where $T_1 \bowtie T_2$ is a self-join in the triple store, or a join between two clustered tables in clustered-property storage; the cost function is oblivious to the underlying storage model.

In our experiments, the cost function C_{out} of the query strongly correlates with its running time (ca.85% Pearson correlation coefficient); therefore, if two queries have the same optimal logical plans (with regards to C_{out}), they are expected to have very similar running time.

We consider query Q (with parameters $p_1, \ldots, p_n$) against the RDF dataset D. Every parameter p_i has the domain P_i, and the domain of all the parameters is $P = P_1 \times \ldots \times P_n$. Now, the formal problem of finding the parameter binding is formulated as follows:

PARAMETERS FOR RDF BENCHMARKS: Split P into subsets $S_1, \ldots, S_k$ such that for every S_i holds:

a: $\forall (p_1, \ldots, p_n) \in S_i$ the query Q has the same optimal query plan w.r.t.C_{out}

b: query plan for parameters from $S_k, k \neq i$, is different from the query plan for S_i

Intuitively speaking, we want to cluster the parameters domain into disjoint classes, such that for every class the query Q has the same optimal plan (condition *a*), and these optimal plans are different across different parameter classes (condition *b*).

Since the cost function correlates with running time, queries with identical optimal plans w.r.t.C_{out} are expected to have close runtime, so the properties P1-P4 hold within each set of parameters S_i. Then, the workload generator can produce separate parameter bindings by sampling them from every parameter class independently, thus effectively splitting the query into several cases. For example, BSBM-BI Query 4 would turn into two queries, Q4a (where type parameter denote a very specific product's type) and Q4b (with parameter being a generic type of many products). Alternatively, the benchmark authors can decide to tune the workload generator such that it does not generate parameters from the certain class S_j (if, for example, the total number of distinct classes is too high). Reporting aggregated runtime only within these automatically identified parameter classes will make the results more comprehensible for both users and vendors.

Note that discovery of these sets of parameters is far from trivial. In particular, even if we are given the set of parameters S_j from the candidate solution, checking that it satisfies the condition *a* would require finding the optimal join order for the query Q for all binding from S_j, i.e. it boils down to solving multiple $\mathcal{NP}$-hard join ordering problems. We can, therefore, only aim at a heuristic for it, which is the subject of our future work.

2.7 Instructions

2.7.1 DBGEN

The Social Network dataset generator is a tool built using Apache Hadoop version 1.0.3 under the GPLv3 license. The compilation process uses Apache Maven to automatically detect and download the necessary dependencies. It generates two jars in the target folder: one called ldbc_socialnet_dbgen-.jar that does not include libraries, and the ldbc_socialnet_dbgen.jar containing all the dependencies inside the jar.

To run the generator it is necessary to configure the Hadoop machine or cluster and to setup several generator parameters:

- numtotalUser: The number of users the social network will have. It should be bigger than 1000.
- startYear: The first year.
- numYears: The period of years.
- serializerType: The serializer type has to be one of this three values:
  - ttl : Turtle RDF format
  - n3 : N3 RDF format:
  - csv : generic coma separated value.
- rdfOutputFileName: The base name for the files generated in rdf format (Turtle and N3)

There is also a run.sh script with an example on how to setup the paths and how to execute the generator.

2.7.2 QGEN - BIBM

Compilation

This driver is developed in Java 1.6. The compilation uses Apache Ant to automatically build the application. To compile the project, from the project directory, the following command should be executed:

```
ant
```

Before that, the script classpath.sh has to be executed, as well:

```
. ./classpath.sh
```

This will generate a jar file in the lib folder (bibm.jar).

Execution

Before running the driver, a user should execute the script extract_td_data.sh from the td_data folder, in order to prepare real parameters used in benchmark. All the queries require substitution of formal parameters with the real ones, but the selection of them should be done in such a way that the query results should not be empty. For example, a query, that has to find friends from a specified person that have replied the most to the posts with a tag in a given category, has two formal parameters (person and tag category) that have to be replaced with the real values, guaranteeing us that the result will not be empty. The previously mentioned script runs a lot of statistic queries against the database that will be tested, and store the results in the textual files, used in the later phase by test driver.

To execute the driver, the run.sh script is provided. A user can modify it, depending on his needs. Some of the most common command line arguments are summarized bellow:

```
Openlink BIBM Test Driver 0.7.7
Usage: com.openlinksw.bibm.SIB.TestDriver <options> endpoints...
endpoint: The URL of the HTTP SPARQL or SQL endpoint
    -defaultparams
        use default query parameters
    -dg <default graph>
```

```
    add &default-graph-uri=<default graph> to the http request
-dry-run
    makes parameter substitution in queries but does not actually runs them.
    Implies -printres option
-err-log
    log file name to write error messages
    default: print errors only to stderr
-help
    prints this help message
-idir <data input directory>
    The input directory for the Test Driver data
    default: td_data
-mt <Number of clients>
    Run multiple clients concurrently.
    default: 1
-o <benchmark results output file>
    default: benchmark_result.xml
-pq
    print queries before runs
-printres
    include results into the log
-pvq
    trigger the profile view query for returned persons
-q
    generate output qualification file with the default name (run.qual)
-qcf <input qualification file name>
    To turn on comparison of results.
    default: none.
-qf <output qualification file name>
    generate output qualification file with this name
-qrd <query root directory>
    Where to look for the directories listed in the use case file.
    default: current working directory
-rpvq
    trigger the profile view query for some of the returned persons
-runs <number of query mix runs>
    default: the number of clients
-seed <Long Integer>
    Initialize the Test Driver with another seed than the default.
    default: 808080
-t <timeout in ms>
    Timeouts will be logged for the result report.
    default: 0
-uc <use case query mix directory>
    Directly specifies a query mix directory.
    Can be used several times, and with -ucf optionDirectly specifies a query mix directory.
-ucf <use case file name>
    Specifies where the use case description file can be found.
-w <number of warm up runs before actual measuring>
    default: 0
```

2.7.3 LDBC_DRIVER

All programming language used for developing ldbc_driver is Java 1.6. It is compiled using Apache Maven, but also depends on one project that is not yet available via a public Maven repository. To ease the compilation process, a `build.sh` script is provided, which downloads and builds all non-Maven dependencies, then integrates them into the Maven build by creating *in-project Maven repositories* for each of them. Ultimately the build packages ldbc_driver into one file named core-*VERSION*.jar (this will be renamed to a more descriptive name in future).

The driver can then be run from Java command line using:

```
java -cp core-VERSION.jar com.ldbc.driver.Client -db <classname> -l | -t -oc <count>
```

```
[-P <file1:file2>] [-p <key=value>] -rc <count> [-s]  [-tc <count>] -w <classname>

   -db,--database <classname>       classname of the DB to use
                                    (e.g. com.ldbc.driver.db.basic.BasicDb)
   -l,--load                        run the loading phase of the workload
   -oc,--operationcount <count>     number of operations to execute
   -P <file1:file2>                 load properties from file(s) - files will be loaded
                                    in the order provided
   -p <key=value>                   properties to be passed to DB and Workload - these
                                    will override properties loaded from files
   -rc,--recordcount <count>        number of records to create during load phase
   -s,--status                      show status during run
   -t,--transaction                 run the transactions phase of the workload
   -tc,--threadcount <count>        number of worker threads to execute with
   -w,--workload <classname>        classname of the Workload to use
                                    (e.g. com.ldbc.driver.workloads.simple.SimpleWorkload)
```

3 Status of the Benchmark

3.1 Interactive Query Set

The Interactive workload tests system throughput with relatively simple queries and concurrent updates. In the first version there are twelve queries, and the transactional updates will be defined during the second year. The current queries are:

Q1: Given a person's first name, return up to 10 people with the same first name sorted by last name. Persons are returned (e.g. as for a search page with top 10 shown), and the information is complemented with summaries of the persons' workplaces and places of study.

Simple lookup query. 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: Find the newest 20 posts from all of your friends, but created before a certain date (including that date). Id of a friend, Id of a post, and creationDate are returned, sorted descending by creationDate, and then ascending by post URI.

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: Find Friends and Friends of Friends of the user A that have made a post in the foreign countries X and Y within a specified period. We count only posts that are made in the country that is different from the country of a friend. The result should be sorted descending by total number of posts, and then by person URI. Top 20 should be shown. The user A (as friend of his friend) should not be in the result.

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: Find top 10 most popular topics-tags (by total number of posts by tag) that your friends have been talking about in last 24 hours, but not before that. The query finds tags that are discussed among one's own friends in time interval 1 and not in time interval 2. Typically the first interval is a short span and interval 2 is the time from the start of the dataset to the start if the first interval. The query will quite often come out empty, however this depends on the size of the intervals. The time interval 1 is closed, the second one is open. It two tags have the same count, sort them ascending by tag URI.

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: 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: Find 10 most popular interests (hashtags) of people that are connected to you via friendship path and talk about topic 'X' Among posts by friends and friends of friends (excluding you), find the tags most commonly occurring together with a given tag. Sort the result descending by count, and then ascending by tag URI.

Q7: For the specified person, this query gets the most recent likes of any of the person's posts. Likes from outside the direct connections are specially gratifying, hence these are flagged. The latency between the post and the like is also reported. Speed of reaction is a potential correlate of interest. The results are ordered descending by creation date of the like, and then ascending by liker URI, and the top 20 are returned.

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: This query retrieves the most recent (20) replies to all the posts of the specified person. Order them descending by creation date, and then ascending by reply URI.

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: Find the newest 20 posts from all of your friends, or from friends of your friends (excluding you), but created before a certain date (excluding that date). Id of a friend, Id of a post, and creationDate are returned, sorted descending by creationDate, and then ascending by post URI.

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: Users like me that I do not know but my friends do. The query looks for a friend of a friend (excluding you) who posts much about the interests of the user and little about topics that are not in the interests of the user. The search is restricted by the candidate's horoscope sign. The result should contain 10 FOFs, where the difference between the total number of their posts about the interests of the specified user and the total number of their posts that are not in the interests of the user, is as large as possible. Sort the result descending by the difference mentioned in the previous item, and then ascending by FOF URI.

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: Find a friend of the specified person, or a friend of his friend (excluding the specified person), who has long worked in a company in a specified country. Sort ascending by start date, and then ascending by person URI. Top 10 should be shown.

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: Find friends of a specified user that have replied the most to posts with a tag in a given category. The result should be sorted descending by number of replies, and then ascending by friend's URI. Top 20 should be shown.

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.

Table 3.1 shows the choke points covered by each of these queries.

Choke point	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
CP1.2		*							*			
CP1.6									*			
CP1.7	*											
CP2.3	*											
CP2.4		*					*					
CP2.6					*		*	*		*		
CP2.7		*		*	*		*		*	*	*	
CP3.3			*									
CP3.5		*						*	*			
CP4.2a										*		
CP5.1										*		
CP5.3										*		
CP6.3										*		
CP7.1	*									*		
CP7.2						*						*
CP7.3												*

3.1.1 SNB Update Workload

SNB aims at testing a broad range of update behaviors likely to occur in a graph context. These include non-transactional bulk load for initializing the dataset plus handling update streams while processing queries under different transaction profile constraints. Finally the graph analytics workloads offer cases with very large update volume and high concurrency but relaxed latency requirements.

The update scenarios are characterized by the following dimensions:

- ACID semantics (AC)
- Latency (LA)
- Throughput (TP)
- Update in place with contention (UP)

The update scenarios in the SIB workloads are as follows, each entry is prefixed with the abbreviation of the aspect(s) it exercises.

- **TP - Bulk load:** No update conflicts occur and the bulk load does not have to be recorverable, i.e. it is accepted that a failure during the load will require a full restart of the load. Insert throughput is paramount. The load must end with a durable consistent state of the database.
- **AC, LA - Interactive SN settings:** For example connecting or unfriending. The effect must be immediately seenm, the response time fast but the likelihood of conflicting updates is minimal. Some transactions will perform a check before update, e.g. does a connection already exist before inserting a connection. Such does in principle involve serializability since an insertt into a place that would have happened on a serializable read nust be detected. However, the read set is small and the transaction duration is short and there is low probability of conflict. Full ACID semantics are required.
- **TP, AC - Interactive SN posting:** Most SN interactive writing consists of inserts, mostly in a locally ascending or 'append' pattern. These do not encounter contention in the transactional sense as each can be completed independently. Contention may occur at the page level, for example if an index is kept with time as the primary ordering key, the right edge of the index will see high activity.

  It is conceivable that a test sponsor were allowed to combine writes from many simulated users into a single transaction on the SUT, but this matter is open.

- **TP, AC - BI trickle load:** This is like interactive updates, except that now the test sponsor may decide to combine many transactions on the online system into fewer but larger transactions on the history warehouse.
- **TP, UP - Graph analytics:** Graph analytics algorithms are often expressed in a vertex centric manner where each iteration updates the state of the vertices involved. This creates high update pressure even for the simplest algorithms. Further, since these algorithms are often expressed in a bulk synchronous model and involve essentially random connections where any vertex may add input to any other vertix's next step with possible message combining, we will not only see inserts but also updates in place. The throughput requirement will be very high but the latency of any single operation will not be very important, hence there will be large room for operation reordering and other optimizations. On the ACID axis, atomicity and isolation are emphasized, durability not as much. No logging with intermittent checkpoints or no checkpoints at all are valid options.

Of the workloads the only one likely to be dominated by update in place is graph analytics. This may also be the only one seriously requiring repeatable read, for example for incrementing edge or vertex weights from many neighbors at the same time or for doing message combining between iterations of a BSP model.

The two transaction types of the online workload pose the additional requirement of being able to be sustained indefinitely with relatively constant latency. This means that durability by means of transaction log plus delta store alone is not sufficient, the database must be checkpointed from time to time, which will typically increase latency for the checkpoint duration.

3.2 Benchmark results

3.2.1 Virtuoso

In this section we present benchmark results on Virtuoso.

The dataset in question is the dataset generated by the Data Generator, using the validation parameters. It mimics the activities of 10k users during 3 years period. The results should be considered as exploratory, because the query results are not validated yet. There is no bad query plans, but for some queries, these might not be the ideal ones either. The additional enhancement is needed, in order to get the final results. The specification of the used hardware follows: 24 x Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz CPU, 193 GB of RAM. Operating system, the Virtuoso is running on, is CentOS release 6.3 with 2.6.32-279.14.1.el6.x86_64.

We run ten query mixes, each containing one instance of each query, with random parameters, after warm-up phase. The results are summarized in Figure 3.1, that shows the average execution times of the queries. It is obvious that queries 4, 5, and 9 cannot be considered as interactive, because the average execution time exceeds several times a second, that can be treated as reasonable limit for execution time of an interactive query. Minimum and maximum execution times, per query, and their standard deviations are shown in Figure 3.2 and Figure 3.3. From the second figure, one can conclude that the queries with the execution time over 1s have high standard deviation. This proves how the dataset, or more precisely, the data distribution "irregular" is, reflecting the reality of the social networks. This further complicates the selection of query parameters in the benchmark. Currently, they are chosen in a way that guarantees that every pattern in the SPARQL queries will not cause the result of the query to be empty. But on the other hand, their combination might produce the empty result set, as stated in Table 3.1.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Min	1	20	20	0	20	0	0	4	20	6	0	1
Avg	4.3	20	20	3	20	6.8	18	17.7	20	9.6	4.5	10
Max	10	20	20	10	20	10	20	20	20	10	10	20

Table 3.1: Minimal, average and maximal number of results for each of the queries.

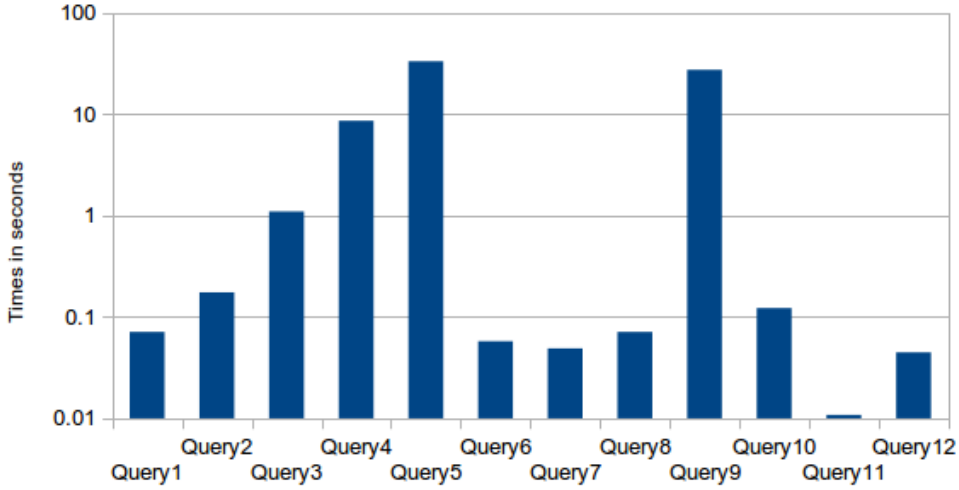

Figure 3.1: Average query execution times on Virtuoso.

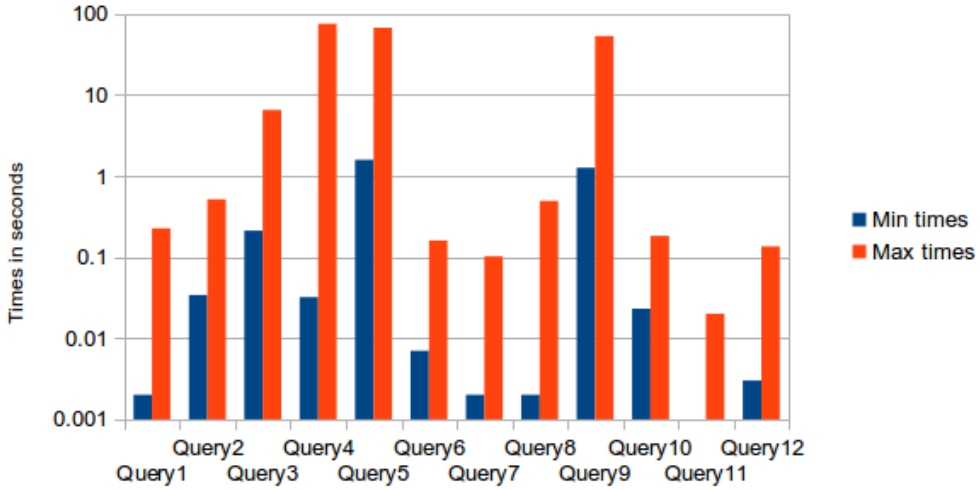

Figure 3.2: Minimal and maximal query execution times on Virtuoso.

This section presents a preliminary analysis of running SNB interactive on a dataset of 100K users, as well. From the Figure 3.4, it can be concluded that the same queries (4, 5, and 9) are the most time consuming in both cases.

Data Sizes

Using the Virtuoso column store the 100K user dataset takes approximately 60GB in allocated database pages. Of these, the interactive workload touches 2/3. In the RDF data model the literals are slightly more space consuming than the quads.

One could run a 1 million user benchmark with reasonable performance in 192 or 256GB of RAM with the rest on SSD. This is common on single servers. 10M users is definitely a scale out size. 100M users is a data center size, e.g. cluster of 2x100+ servers. At this scale, running partitions in duplicate becomes important. 1 billion users will require special measures, possibly splitting the functionality into many physical databases.

Date Access Behavior

In Table 3.2, for each query the counts of index lookup and sequentially scanned rows are given. The last

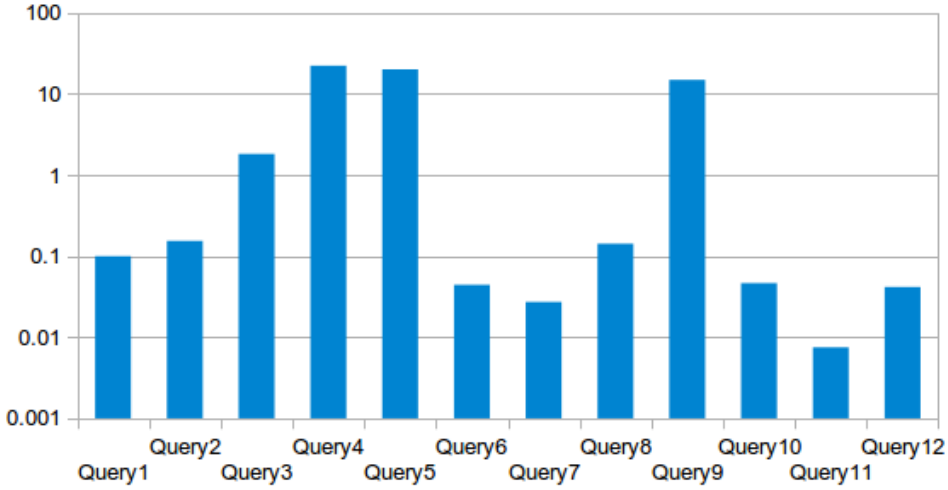

Figure 3.3: Standard deviation of query execution times on Virtuoso.

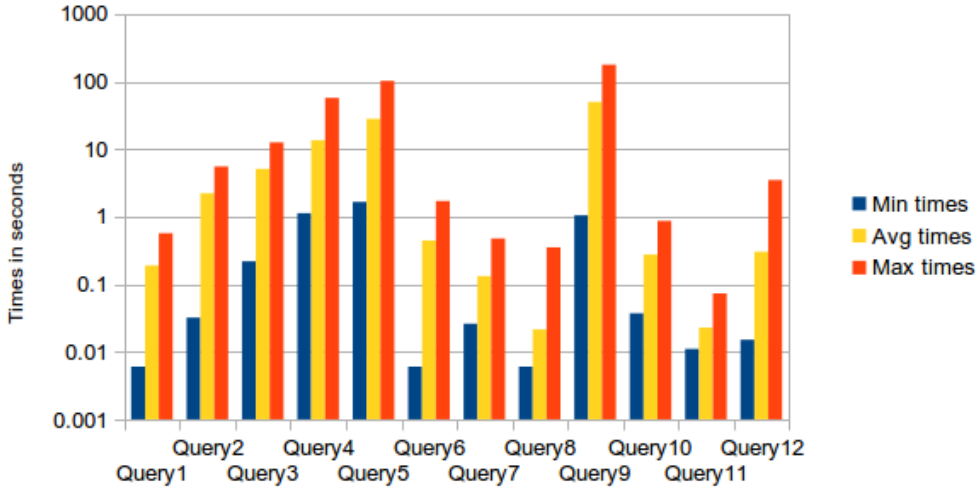

Figure 3.4: Minimal, average and maximal query execution times on Virtuoso (100k of users).

number is a metric of locality, the percentage of times when the next lookup falls within the same column store segment than the previous one. A segment is on the average 18000 rows. There are 200 executions of each query.

Query Compilation Time

Figure 3.5 shows a breakdown of the time for compiling the 12 queries. Each of the times contains 200 compilations without reuse. The times are measured in clocks elapsed between start and end of compilation scaled down to seconds based on a clock of 2.3GHz.

Plan Variability

Figure 3.6 shows the count of distinct execution plans for each of the interactive queries. Different parameters result in different join orders and join types.

Analysis of Workload Characteristics

We notice high variability in query execution times between invocations. This is expected, as the amount of data touched is usually relative to the two-hop social environment of the user making the query. Many of the

	Lookup	Seq scanned rows	Locality
Q5	32798509	28754542	99.78918356088631
Q9	25707233	11921403	66.58320284457644
Q3	6242993	9702354	99.167708780952647
Q4	4411810	19727865	99.708607354933606
Q2	593535	394986	69.192744388052567
Q12	439809	422627	99.290424426818172
Q10	394553	898371	99.08851337781436
Q6	192749	5268111	98.907901143424421
Q7	30287	8122	66.820028645178161
Q8	5087	3819	95.232310485103489
Q1	3079	2349	49.920366783613578
Q11	10	4	0

Table 3.2: Number of index lookup, sequentially scanned rows, and locality for each of the queries.

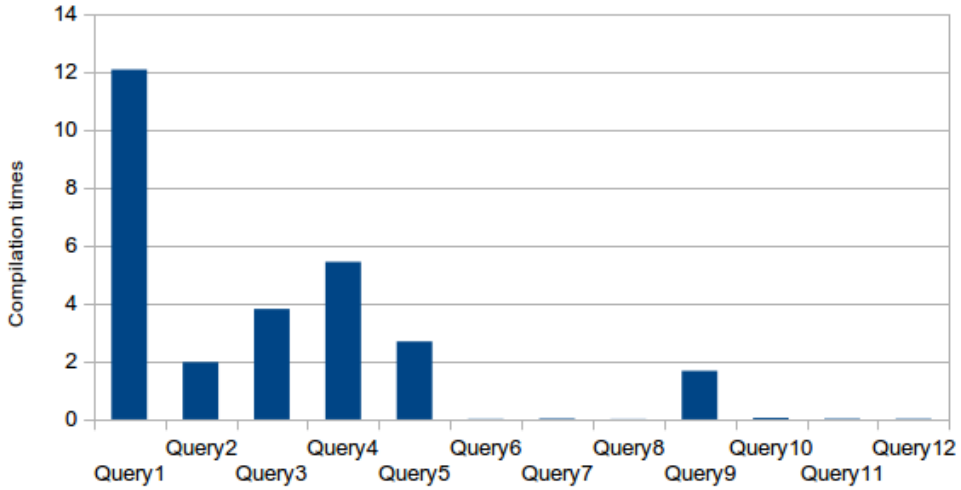

Figure 3.5: Query compilation times.

queries are quite heavy for an interactive workload. The queries are still of a lookup nature and the volume of data touched is expected to remain relatively constant as the database is scaled up. The only query that is not scoped to a person is Q1, which will hit more data as the database grows. This is undesirable and Q1 should consequently also be scoped to a social environment, e.g. 3 hops.

In keeping with the RDF model, the count of random accesses is high relative to sequential access. This is natural since each property lookup counts as a random access. We do see significant locality in data access, e.g. 99% of index lookups fall within a 100 rows or so of the previous one with Q5. We expect this locality to drop as the data size grows. It will however remain significant, so that benefits of vectorized index access over tuple at a time access will continue to be large. We notice that Q9 takes more time than Q5 even though Q5 accesses more rows of data. This is because Q5 has a much more local access pattern: 99+% lookups fall on the same segment as the previous, whereas only 66% fall on the previous in Q9.

We see large variability in query plans driven by parameter choices. The extreme case is Q3 that has 42 differentt plans for 200 executions. The number is artificially inflated due to the union in the query that causes the query too be wholly executed for the one hop and then the two hop connections. The join types are influenced by the cardinalities, which depend on the social rank of the person and on the size of the countries mentioned in the query. Thus some extra work is done.

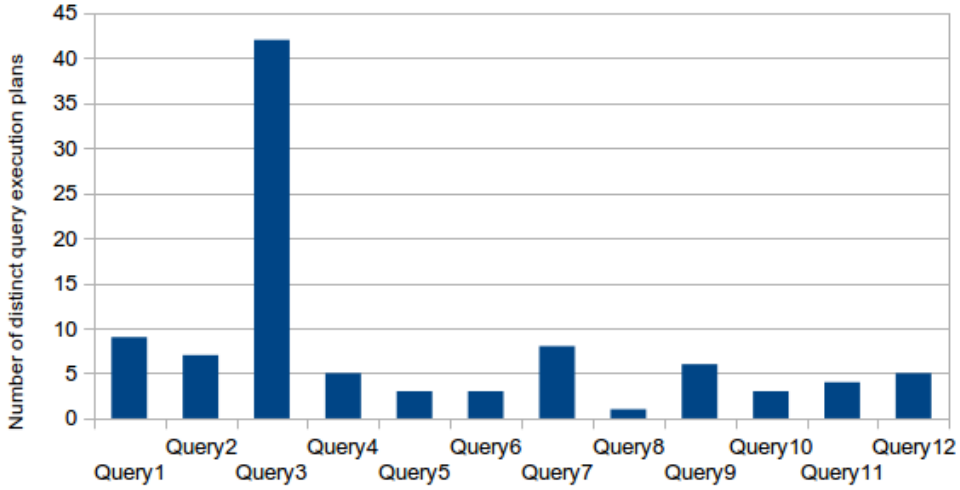

Figure 3.6: Number of distinct execution plans per query.

It remains an objective of the workload to exercise query optimization so as to drive plan selection by parameter values. However the parameter choices need to be made from predictable buckets. Any implementation that supports multiple join types will have a break-even point in the cost model where an index lookup changes into a hash join plan or vice versa. The queries end up testing the correctness of the break - even by having continuously varying parameters such as date ranges.

Q9, which selects the most recent posts from the 2 hop environment of a person takes most of the execution time. As an example, it would benefit from late projection since the author names only need to be retrieved for the posts actually returned. However, an RDF model cannot do that without extra logic since it is not known that all persons have one last name and one first name. Also, the identifiers of URIâĂŹs and literals need to be converted into the returned string for only the 20 posts actually returned. Thus doing the top k order by earlier in the plan would reduce the execution time by around 75%.

In some instances of Q9 the first name and last name of the author of posts are accessed via hash join if it is determined that there will be many posts, i.e. the person has many outbound edges and by index if there will be few posts. It would be better still to only retrieve first names and last names for the authors of the actually returned posts but this would require corrective action if it turned out that result tuples were dropped after the top k order by due to these not having a first or last name.

An SQL implementation may here do late projection with greater ease. Conversion of internal identifiers into external names can be a serious choke point in RDF lookup queries.

3.2.2 DEX

DEX API usage analysis

In this section, we analyze how the benchmark exercices the DEX's API. Table 3.3 shows, for each query, the different API methods used. The first result extracted from this table, is that all used methods are read-only. This result is expected since currently, the interactive query workload does not contain any update query. Second, we see that there are two main categories of methods: those aimed at exploring the graph, and those used to manipulate collections of objects/entities. We see that the methods that fall into the first category are considerably more used than those that fall in the second. This category includes *Graph.getAttribute*, which is the only method used in all the queries and is used to obtain the data to be output during the projection. Similarly, *Graph.findObject* is used in those queries where the input parameter is a specific entity, which are all the queries except query 1. This method finds that object that matches an URI specified by a parameter, i.e. a person. This entity is then used as a starting point for the query. Finally, we find *Graph.select*, *Graph.neighbors (oid)* and *Graph.neighbors (objects)*. These methods are the core methods on any query. The first performs a simple select operation, that is, it obtains the subset of objects that fulfill a simple condition on one of their attributes. The next two operations (neighbors), are those which allow us to navigate through the edges of the graph. They are one of the main features of DEX and graph databases in general, and hence they are highly used. The methods in the second category allow us to manipulate the collections of objects returned by the methods in the first group. Conceptually, they are used to extend the simple selects to implement more complex join operations. These are used in several queries and usually in groups of more than one. Finally, note that there is not a method in the table to sort the results. DEX does not provide any mechanism to sort the data that is output, and hence this has to be performed outside of the engine. In Appendix A.2, we show the implementation of query 5 as an example of how a query in DEX is actually looks like.

To sum up, the benchmark is not currently performing updates, and hence only a subset of read only methods of the DEX API are exerciced. However, according to our expertise, these methods cover almost all of the data retrieval mechanisms provided by DEX, so they are representative enough from an evaluation perspective.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Graph.getAttribtute	X	X	X	X	X	X	X	X	X	X	X	X
Graph.getEdgeData	X						X				X	
Graph.findObject		X	X	X	X	X	X	X	X	X	X	X
Graph.select	X	X	X	X	X				X		X	
Graph.neighbors (oid)	X	X	X	X	X	X	X	X	X	X	X	
Graph.neighbors (objects)		X	X	X	X	X		X	X	X	X	
Graph.explode (oid)	X											
Graph.explode (objects)					X		X				X	
Graph.tails					X							
Graph.exists										X		
Objects.remove			X		X	X			X	X	X	
Objects.union			X		X	X					X	
Objects.intersection				X	X	X			X		X	
Objects.combineIntersection			X	X								X
Objects.difference				X	X					X		

Table 3.3: API used by the different queries.

Query execution evaluation

We present execution results on DEX, using the validation configuration and execution parameters. We take these results as preliminary because 1) further tests are needed in order to validate that the query results are correct and 2) the actual implementations might not be optimal for such queries. We want to identify which

are the bottlenecks of these queries and use them as guidelines for future optimizations. For each query, we have executed an instance for warming up the database, followed by ten executions of the query, which have been averaged to produce the final result. The machine used have the following characteristics: 2 x Intel Xeon E-2609 @ 2.40 Ghz CPU, 128 GB of RAM, Debian Linux with 2.6.32-5-amd64 and 2 2.5" HDDs with 2 TB at 7200 rpm each. Figure 3.7 shows the execution times of the queries using DEX.

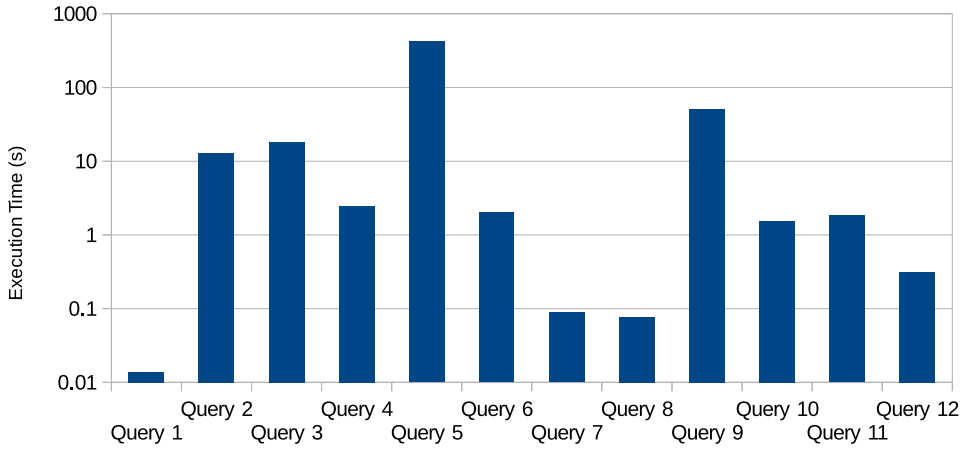

Figure 3.7: Execution times on DEX.

For each query, our goal is to identify those most time consuming operations that become the bottlenecks of such query. Furthermore, we want to see whether these operations are shared by different queries or not. The value of this information is twofold: on the one hand, it will allow us to define better practices when implementing queries in DEX, by avoiding the usage of those more expensive operations. On the other hand, this information will indicate which operations to focus on when optimizing the system. Table 3.4 shows a selection of the most time consuming and widely used operations used by the selected queries, and what percentage of the total execution time is actually spent on them. Note that, from one hand we only include in the table those most time consuming operations, and hence, the sum of the percentages may be below 100. Furthermore, we have aggregated some operations by concepts, i.e. in the case of neighbors we aggregate the two versions of the operation. Finally, the projection includes both getting the attribute by means of *Graph.GetAttribute* as well as the time spent in iterating the results.

Broadly speaking, we see that considering the interactive nature of the benchmark, there are some queries that take too long. We identify four possible bottlenecks: selects, neighbors, explode and the projection. On the other hand it seems that at least, for these queries, the operations involving collections of objects does not suppose a problem. We see that, for queries 1, 2, 3 and 4, the bulk of the execution time is spent in selects. In these queries, the search space of the select is very large, incurring in a prohibitive cost. Therefore, selects are a critical operation to be optimized. Alternatively, having new DEX API operations such as a select restricting the search space (an operation not currently supported) might improve significantly the performance of such queries if it could be used.

On the other hand, for queries 5, 6, 10 and 12, the bulk of the execution time is spent in the neighbors operation. In contrast to queries 1, 2, 3 and 4, neighbors is, in general, a cheap operation according to our results. However, in queries 5, 6, 10 and 12 it becomes a bottleneck because it is called thousands of times (over 200K times for query 5). Therefore, although the neighbors queries are in general cheap, improving them is critical if we cannot reduce the number of calls to them.

Explode is an operation very similar to neighbors, but instead of returning the objects at the other end of the edges, it returns the edges itself. Similarly to neighbors, this is in general a cheap operation. However, due to the cardinality of the temporary collections returned by the calls to explode in query 11, this becomes a bottleneck.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Graph.neighbors	-	0.73	-	3.86	**95.12**	**99.02**	0.027	4.35	0.07	**65.18**	0.11	**73.24**
Graph.select	**42.18**	**91.23**	**88.82**	**95.46**	2.86	-	-	-	37.08	-	0.12	24.08
Graph.explode	-	-	-	-	-	-	-	-	-	-	**99.58**	
Graph.tails	-	-	-	-	0.48							
Objects.union	-	-	2.69	-	0.49	-	-	-	-	-	-	-
Objects.intersection	-	0.33	-	0.11	0.09	0.64	-	-	0.54	-	-	-
Objects.difference	-	-	-	-	0.16	-	-	-	-	-	-	
Projection	37.94	5.03	10.97	0.02	1.02	0.44	**89.74**	**86.70**	**55.82**	34.69	-	1.72

Table 3.4: Percentage of execution time spent in each operation type.

	1	2	3	4	5	6	7	8	9	10	11	12
σ	4.93%	1.62%	0.11%	0.07%	0.37%	0.41%	16.19%	7.72%	23.08%	1.29%	0.11%	0.31%

Table 3.5: Standard deviation for each of the queries.

This is a clear case where the query could possibly be implemented in another way, more efficiently. This suggests that having a query plan optimizer, something that DEX does not currently have, would be appreciated.

Finally, for queries 7 8 and 9, we see that the bulk of the execution is split in the projection (and select in the case of query 9). The programming paradigm used in DEX, forces us to implement the projection as a loop with successive calls to *Graph.getAttribute*. This is a potential source of bottlenecks in queries, mainly for two reasons: the first is the amount of calls to *Graph.getAttribute* to be performed, since this incurr in a large overhead. The second, is the order in which these calls are performed, which might not be the best from a cache locality perspective. Organizing the internal data structures in order to improve this type of DEX's usage would be on of the possible optimizations to perform. The other possibility is clearly performing the projection in some kind of batches, to reduce the amount of calls overhead.

Table 3.5 shows the standard deviation of the executions of each query. Having executions with a low execution time variability is a requirement, since the evaluated queries are expected to be interactive. Broadly speaking, all the queries except 7, 8 and 9, have a low variability (below 5%). For queries 7 and 8, such a large standard deviation is not critical since the queries are fast enough. However, for query 9, such a large variability can become a problem, since may push the execution time of some executions above what can be considered interactive.

Hence, from these preliminary analysis of the execution of the benchmark on DEX, we conclude the following:

- We have identified four possible sources of bottlenecks: selects, neighbods, explode and the projection.
- For selects, these are problematic if the cardinality of the search space is large. Therefore, they have to be optimized or alternatively, be avoided or executed on smaller sets. The later is the most immediate improvement over the current implementation which would potentially improve the execution times. However, it would require adding new and more specialized operations to the DEX API, such as making a selection over a specific set of objects (currently, a select have to be performed over all the objects with a particular attribute type). Having such operation would potentially increase the performance of queries 2, 3, 4 and 9.
- Some queries require too many calls to the Neighbors operation. These queries should be reshaped in order to reduce the number of calls or alternatively improve the neighbors operation to perform faster.
- Explode can become a bottleneck since the result can be very large. An alternative, would be to perform smaller but faster calls to explode, with smaller return collection sizes. In this case, having a query plan optimizer for DEX that could decide the best way to perform the queries would be appreciated.
- The way the projection is performed is has a lot of overhead, and is not cache friendly. Having the possibility to perform batches as well as organizing the internal data structures to be aware of the way the projection is performed could improve the results.

- Finally, query 9 has too much variability in its execution time. First of all, the query should be optimized, and if the variability remains, determine where this variability comes from.

First optimization steps

Improving Selects: Improving the Select operation is the first action to performed after the analysis of Section 3.2.2. We have implemented a simple test program, that emulates the select performed on query 2, using the same dataset. Table 3.6 shows how much time is spent in the select operation, split into the two largest blocks of code: the iteration of the results and the copy of the bitmaps. We see that the most of the time is spent in iterating for each possible value in the index. Hence, the cost of the select is directly proportional to the size of the candidate set where the select is performed over.

Iterator	96.66%
Copy Bitmap	3.33%

Table 3.6: Internal time spent in select.

After analyzing where the time spent in the iteration block comes from, we realize that it is spent mainly in two functions: GetBitmap and AddBitmap. Table 3.7 sumarizes the time spent in executing GetBitmap and AddBitmap over the total execution time of the iteration block.

GetBitmap	36%
AddBitmap	57%

Table 3.7: Internal time spent in iterator.

From these analysis, we have proposed an improvement related to GetBitmap. Figure 3.8 shows the relative execution time of the new select implementation over the existing one. We see that the new is roughly a 18% faster.

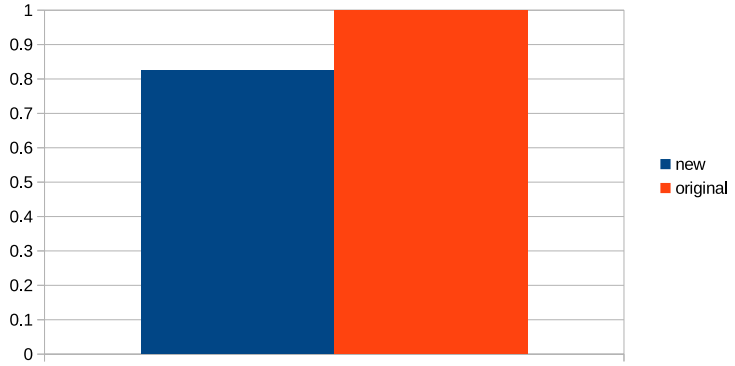

Figure 3.8: Normalized execution time of new select vs the old implementation.

The Restricted Select: Although the performance's improvement of the existing select operation has been significant, for the purpose of the queries of the benchmark it is not sufficient. As stated above, the cost of the select operation seems directly proportional to the size of the select search space. By observing the queries of the benchmark, we realize that having a restricted select, that is, a function where the search space is restricted to a subset of objects, would be appreciated. For instance, in query 2, we can first obtain the subset of posts created by our friends, and then perform a restricted select over this set. We have added the restricted select variant into the API. Figure 3.9(a) shows the normalized execution time of query 2 with respect to the original

query. We see that, thanks to this new API function, the time has been reduced by more than a 90%. However, further experimentation have shown that this particular API is not useful for queries 3 and 9. The reason is that in these queries, we are not able to restrict the search space enough. This result shows that this new API function is only useful when the search space is small. Furthermore, performing a restricted select over a too large serach space, is more expensive than performing the original select over the whole data set.

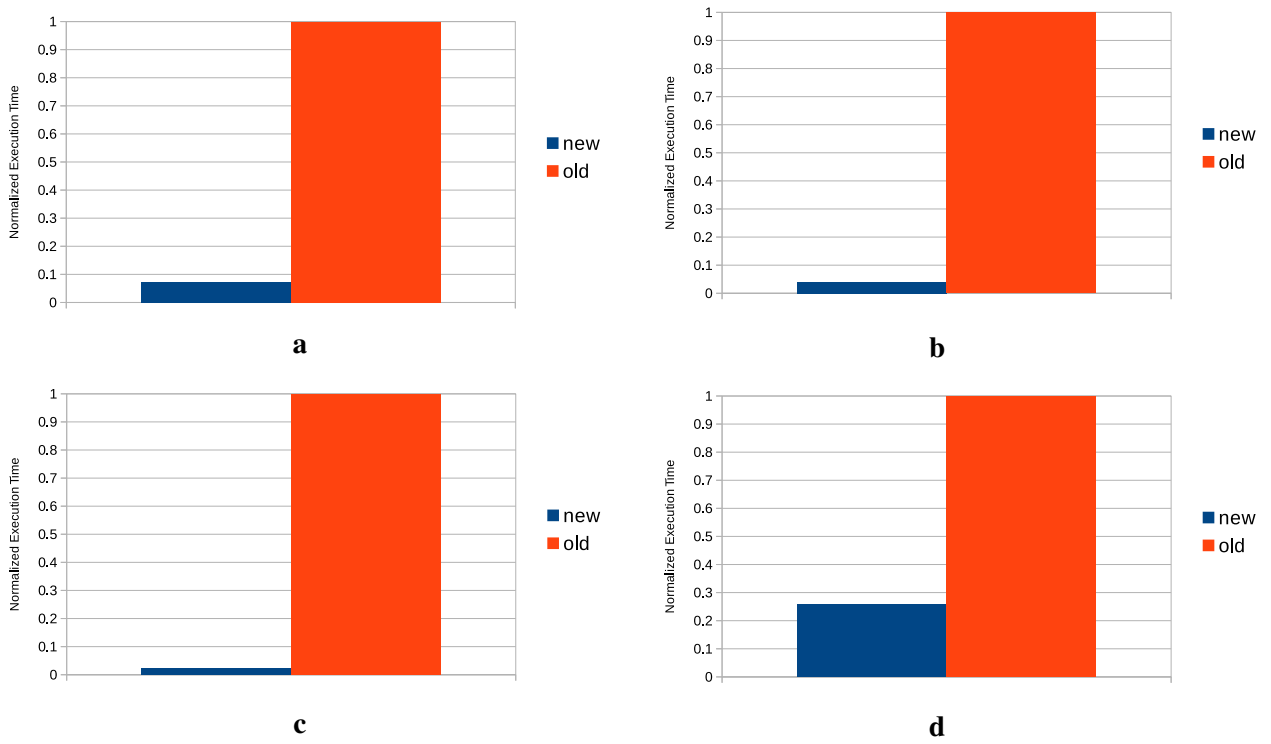

Figure 3.9: **a)** Normalized execution time of query 2 with and without restricted selec. **b)** Normalized execution time of query 9 with and without the pseudo top-k implemented. **c)** Normalized execution time of query 5 compared to the original implementation. **d)** Normalized execution time of query 3 compared to the original implementation.

Top-k: After successfully improving the time of query 2, we have moved our attention towards query 9. Despite its similarities with query 2, with query 9 we can not follow the same approach, since we are not able to restrict the select space enough. By analyzing query 9, we see that it could be potentially improved if we had an efficient top-k query available. Since implementing a tok-k function efficiently into the core of DEX is not trivial, we have performed an implementation of query 9 that performs a top-k at the java layer. Hence, we validate the potential of having this function as part of the core API. This implementation consists on repeatedly making restricted selects using consecutive intervals by the attribute used in the top-k, which in this case is a Date attribute, until k results are obtained. Hence we avoid performing a hugh select, and then performing the top-k over all the search space. Furthermore, we reduce the cost of the projection. Figure 3.9(b) shows the normalized execution time of query 9 with respect to its original implementation. We see that the execution time has been improved by more than a 95%. Hence, we conclude that having such a query in the DEX API would be highly appreciated.

Neighbors improvement: After that, we have moved to query 5, which is the most expensive one. In the analysis performed initially, we have seen that the bulk of the execution time is spent into neighbors. We have realized that, when performing a Neighbors operation of a set of objects, we are creating an unnecessary temporary data structure when the set of objects have a cardinality of 1. Figure 3.9(c) shows the normalized execution time of query 5 after fixing this issue with respect to the original implementation. We see that this new implementation reduces the execution time by more than 97%.

Having a query plan optimizer: Finally, we have analyzed query 3. In this case, the usage of restricted select was not initially useful. The reason was that we were not able to reduce the search space enough, and the cost of the restricted select was larger than without restricting the search space. Hence, we opted by reorganizing the query, by moving the bulk of the implementation inside a loop, and then performing restricted selects on reduced search spaces. More concretely, we iterate over the friends of the query person, and over the posts of each friend, perform a restricted select. Figure 3.9(d) shows the normalized execution times of query 3 with respect to the original implementation. We see that the time has been improved in about 75%. This result ilustrates the potential benefit of having a query plan optimizer for DEX, that would perform this optimization transparently from the programmer.

New execution times

Figure 3.10 shows the execution times for all the queries of the benchmark. Although the times for some of the queries can stil be considered high according to the interactive nature of the benchmark, we see a huge improvement with all the queries running in less than 10 seconds. Furthermore, there is stil room for optimizations. For instance, having a query plan optimizer or the top-k function as part of the core, would potentially improve the times not only for queries 3 and 9. Furthermore, since the current execution times have changed with respect to original ones, the possible bottleneck operations for each query might have changed, openning a broad range of new optimization possibilities.

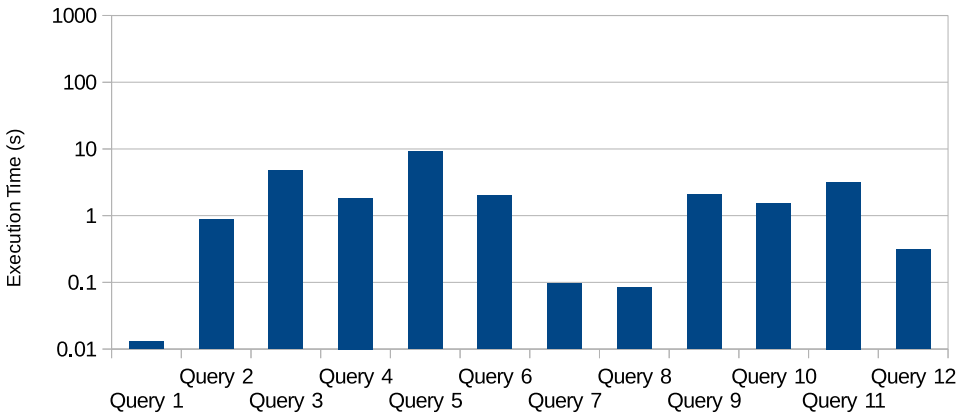

Figure 3.10: New execution times for the optimized queries.

Finally, Table 3.8 shows the standard deviation of the new executions. We see that there are still queries with a high execution time variability. This is something that has to be further analyzed since as explained above, in the context of interactive queries, this variability can negatively affect the user's satisfaction.

	1	2	3	4	5	6	7	8	9	10	11	12
σ	5.72%	18.82%	0.07%	0.15%	0.7%	0.24%	14.24%	7.20%	0.40%	6.05%	0.34%	0.9%

Table 3.8: Standard deviation for each of the optimized queries.

4 Conclusion

The first year of the LDBC project has establish the foundations for the construction of a set of benchmarks to test various functionalities of systems used for graph data management using graph-shaped data as in a large social network. It has provided also the first preliminary benchmark for interactive queries over a synthetic social network graph created by a dataset generator. This benchmark has been succesfully tested and analyzed by some vendors, and the conclusions of this work plus other two benchmarks are going to be the start point for the work of the second year.

One interesting challenge for this task force has been to begin the collaboration of different teams with different approaches and skills. From one side there are several University research teams such as VUA, UPC or TUM, which provides their formal and experimental knowledge on database theory, data management, database performance, graph analytics and benchmark design. On the other side there are different IT companies such as OpenLink, Neo and Sparsity that have different technologies for the management and exploitation of graph-like data. This has been another challenge for the team: graph data is actually represented in different forms such as RDF triples or property graphs, and while there are well-known and widely used standards for RDF/SPARQL, the graph database community still lacks on standardized ways to define and query graphs. Thus, it has been necessary to find a common way to express and represent graph data and graph queries, and in the next months we will need to improve and extend even more the graph formalization and queries by proposing graph query definition and manipulation languages. For all of this, the collaboration of the Technical User Community (TUC) will be very helpful. During the first year we have organized two interesting meetings with multiple academic and industrial organizations that exploit, analyze or consume graph-like data, but we still have not been able to fully integrate them to our definition and validation process. One goal for the next year is to improve the collaboration with the TUC to get more feedback and proposals; this will provide a broader scope of the project, a closer approach to the industry and market requirements, and a better quality on the results of the Task Force, e.g. the SNB benchmark and the benchmarking methodology.

In particular, the design and benchmarking methodology has made its first steps but still needs a lot of work. The first task after setting a common framework has been to identify the benchmark elements, procedures, actors and roles. This knowledge will be used in the next months to define more clearly the benchmarking procedure, the final requirements for the dataset generator and query drivers, and the auditing rules for benchmark publication. The methodology for graph data and query specification has been also an active area of development. We have now some proposals on how to formally model the graph, how to express graph queries and how to run and validate the query results. In the next months also it is expected that the feedback from the TUC community as well as more detailed results of experiments from the industrial partners will allow the Task Force team to obtain a more adequate and portable specification method that perhaps will be suitable for a future standard recommendation.

One important theoretical contribution of the first year has been an exhaustive “choke points” analysis that has been carried out using the expertise of database architects to identify the most important technical challenges that should be tested by the queries. After a detailed work over the classical TPC-H OLTP benchmark, where several groups of distinct choke points have been identified, classified and documented, the activity moved to interactive queries for graphs. While some of the choke points have been identified as the same as for TPC-H, new ones have been proposed based on graph-data characteristics, relationships and correlations. All of these choke points have been used to define and build the first interactive query set, and in the next months they will be used to define the two new benchmarks, BI and graph analytics. Also, it is expected that in the future the TUC will provide some feedback on the current choke point list and most probably new ones will be proposed based on the real experiences of graph database users and vendors.

Perhaps the main effort of the first six months has been the construction of the dataset generator. The work started by adapting the S3G2 data generator algorithm that models a huge correlated property graph. This task has been supported by the TUC input and feedback. The new SNB generator is able to generate a complex synthetic social network with more than 10 entities (graph nodes) and more than 20 relationship types (edge labels). For this, it uses property dictionaries, simple subgraph generation and edge generation along correlation dimensions. Some of these have been tuned to match the requirements of the interactive choke points. Also, the MapReduce implementation allows for the construction of huge graphs with billions of vertices and edges in a few minutes using a distributed architecture. The first analysis done over the generated show that the resulting graph has several of the main properties of real graphs but stil some adjustments will be required during the next months to get even more realistic datasets. Besides, the BI and graph analytical choke points for the new benchmarks will require new extensions to the graph schema, graph data correlations and distributions, and the query drivers will need statistics or sample data provided by the generator to be used in the generation of the substitution parameters for queries.

The first benchmark defined and implemented on top of the datasets generated by SNB has been the Interactive workload that tests system throughput with relative simple queries and concurrent updates. During this first part of the project twelve read-only queries have been defined, documented and tested in two different scenations (RDF/SPARQL and graph databases). The first experiments show that most of the queries are adequate to validate the expected choke points, while a few are too exhaustive or complex to be considered interactive. Thus, the first work of next year will be to tune this query set to guarantee that it is truly interactive. At the same time the effort will be to define a set of concurrent update queries to test the broad range of update behaviors likely to occur in a graph context, with specific focus on transactions and ACID semantics. Also, an important area of activity will be the heuristics for the generation of substitution parameters for the queries because the discovery of the sets of parameters is far from trivial and a uniform selection of parameters probably leads to unpredictable behavior of the queries, thus making intepretation of the results very difficult.

The SNB consists of two more sub-benchmarks, or workloads, that focus on other different functionalities such as Business Intelligence Workloads (with analytical queries), and the Graph Analytics Workload (with graph algorithms). These two benchmarks have been out of the scope of the first year, but the Task Force team have made some adjustements to the dataset generator based on the expected choke points and query sets. Also a preliminary workload for BI has been sketched and it will be available at the beginning of the second year. The Graph Analytics will require an exhaustive enumeration and analysis of the different graph algorithms (clustering, ranking, viral propagation, pattern matching, etc.), their classification and the identification of their own choke points before the benchmark is designed. For this, the collaboration of the TUC and, in particular, from the graph analytical frameworks (e.g. vertic-centric) is a must.

In conclusion, the first year of the LDBC has proposed and tested the first read-only interactive benchmark for Social Networks. For this, some tools such as the dataset generator, the dataset validator and two query drivers have been provided to the Technical User Community and are available in a public source code repository (GitHub). The experience of this year also has clarified the scenario for the remaining tasks to be developed during the second year of the project, where we expect to finish the first benchmark, to create two new ones, and all of this by improving the collaboration with the TUC.

REFERENCES

[1] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, page 15. ACM, 2013.

[2] P. A. Boncz, T. Neumann, and O Erling. TPC-H Analyzed: Hidden Messages And Lessons Learned From An Influential Benchmark. In M. Poess and R. Niambar, editors, *Proceedings of the TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC, 2013)*, August 2013.

[3] Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch. Time-varying social networks in a graph database: a neo4j use case. In *First International Workshop on Graph Data Management Experiences and Systems*, page 11. ACM, 2013.

[4] Andrey Gubichev, Srikanta J Bedathur, and Stephan Seufert. Sparqling kleene: fast property paths in rdf-3x. In *First International Workshop on Graph Data Management Experiences and Systems*, page 14. ACM, 2013.

[5] M-D. Pham, P.A. Boncz, and O. Erling. S3G2: a Scalable Structure-correlated Social Graph Generator. In *TPCTC*, 2012.

A Interactive Query Set Implementations

A.1 Virtuoso

- Query 1

```
select ?person ?last ?bday ?since ?gen ?browser ?locationIP
       ((select group_concat (?email, ", ")
         where {
            ?person snvoc:email ?email
         }
         group by ?person)) as ?email
       ((select group_concat (?lng, ", ")
         where {
            ?person snvoc:speaks ?lng
         }
         group by ?person)) as ?lng
       ?based
       ((select group_concat ( bif:concat (?o_name, " ", ?year, " ", ?o_country), ", ")
         where {
            ?person snvoc:studyAt ?w . ?w snvoc:classYear ?year .
            ?w snvoc:hasOrganisation ?org . ?org snvoc:isLocatedIn ?o_countryURI .
            ?o_countryURI foaf:name ?o_country . ?org foaf:name ?o_name
         }
         group by ?person)) as ?studyAt
       ((select group_concat ( bif:concat (?o_name, " ", ?year, " ", ?o_country), ", ")
         where {
            ?person snvoc:workAt ?w . ?w snvoc:workFrom ?year .
            ?w snvoc:hasOrganisation ?org . ?org snvoc:isLocatedIn ?o_countryURI .
            ?o_countryURI foaf:name ?o_country . ?org foaf:name ?o_name
         }
         group by ?person)) as ?workAt
{
    ?person a snvoc:Person .
    ?person snvoc:firstName "%Name%" . ?person snvoc:lastName ?last .
    ?person snvoc:isLocatedIn ?basedURI . ?basedURI foaf:name ?based .
    ?person snvoc:birthday ?bday . ?person snvoc:creationDate ?since .
    ?person snvoc:gender ?gen . ?person snvoc:browserUsed ?browser .
    ?person snvoc:locationIP ?locationIP .
}
order by ?last ?person
limit 10
```

- Query 2

```
select ?fr ?first ?last ?post ?content ?date
where {
    %Person% snvoc:knows ?fr.
    ?fr snvoc:firstName ?first. ?fr snvoc:lastName ?last .
    ?post snvoc:hasCreator ?fr.
    { {?post snvoc:content ?content } union { ?post snvoc:imageFile ?content }} .
    ?post snvoc:creationDate ?date. filter (?date <= "%Date0%"^^xsd:date).
}
order by desc (?date) ?post
limit 20
```

- Query 3

```
select ?fr ?first ?last ?ct1 ?ct2 (?ct1 + ?ct2) as ?sum
where {
    {select ?fr
```

```
        (((select count (*)
        where {
            ?post snvoc:hasCreator ?fr . ?post snvoc:creationDate ?date .
            ?post snvoc:isLocatedIn %Country1% .
            filter (?date >= "%Date0%"^^xsd:date &&
                    ?date < bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date)) .
        }))
        as ?ct1)
        ((select count (*)
        where {
            ?post2 snvoc:hasCreator ?fr . ?post2 snvoc:creationDate ?date2 .
            ?post2 snvoc:isLocatedIn %Country2% .
            filter (?date2 >= "%Date0%"^^xsd:date &&
                    ?date2 < bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date)) .
        })
        as ?ct2)
    where {
        {%Person% snvoc:knows ?fr.} union
        {%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr. filter (?fr != %Person%)} .
        ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last . ?fr snvoc:isLocatedIn ?city .
        filter(!exists {?city snvoc:isPartOf %Country1%}).
        filter(!exists {?city snvoc:isPartOf %Country2%}).
    }
    }.
    filter (?ct1 > 0 && ?ct2 > 0) .
}
order by desc(6) ?fr
limit 20
```

- Query 4

```
select ?tag count (*)
where {
    ?post snvoc:hasCreator ?fr . ?post snvoc:hasTag ?tag .
    ?post snvoc:creationDate ?date . %Person% snvoc:knows ?fr .
    filter (?date >= "%Date0%"^^xsd:date &&
            ?date <= bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date) ) .
    filter (!exists {
        %Person% snvoc:knows ?fr2 . ?post2 snvoc:hasCreator ?fr2 . ?post2 snvoc:hasTag ?tag .
        ?post2 snvoc:creationDate ?date2 . filter (?date2 < "%Date0%"^^xsd:date)})
    }
group by ?tag
order by desc(2) ?tag
limit 10
```

- Query 5

```
select ?group count (*)
where {
    {select distinct ?fr
     where {
        {%Person% snvoc:knows ?fr.} union
        {%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr. filter (?fr != %Person%)}
     }
    } .
    ?group snvoc:hasMember ?mem . ?mem snvoc:hasPerson ?fr .
    ?mem snvoc:joinDate ?date . filter (?date >= "%Date0%"^^xsd:date) .
    ?post snvoc:hasCreator ?fr . ?group snvoc:containerOf ?post
}
group by ?group
order by desc(2) ?group
limit 20
```

- Query 6

```
select ?tag count (*)
where {
    {select distinct ?fr
     where {
        {%Person% snvoc:knows ?fr.} union
        {%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr. filter (?fr != %Person%)}
     }
    } .
    ?post snvoc:hasCreator ?fr . ?post snvoc:hasTag %Tag% .
    ?post snvoc:hasTag ?tag . filter (?tag != %Tag%) .
}
group by ?tag
order by desc(2) ?tag
limit 10
```

- Query 7

```
select ?liker ?first ?last ?ldt
       (if ((exists {  %Person% snvoc:knows ?liker}), 0, 1) as ?is_new)
       ?post ?content (bif:datediff ("minute", ?dt, ?ldt) as ?lag)
where {
    ?lk snvoc:hasPost ?post . ?post snvoc:hasCreator %Person% .
    {{ ?post snvoc:content ?content } union {?post snvoc:imageFile ?content}} .
    ?liker snvoc:likes ?lk . ?liker  snvoc:firstName ?first . ?liker snvoc:lastName ?last .
    ?post snvoc:creationDate ?dt . ?lk snvoc:creationDate ?ldt .
}
order by desc (?ldt) ?liker
limit 20
```

- Query 8

```
select ?from ?first ?last ?dt ?rep ?content
where {
    {select ?rep ?dt
     where {
        ?post snvoc:hasCreator %Person% .
        ?rep snvoc:replyOf ?post . ?rep snvoc:creationDate ?dt .
     }
     order by desc (?dt)
     limit 20
    } .
    ?rep snvoc:hasCreator ?from . ?rep snvoc:content ?content .
    ?from snvoc:firstName ?first . ?from snvoc:lastName ?last .
}
order by desc(?dt) ?rep
```

- Query 9

```
select ?fr ?first ?last ?post ?content ?date
where {
    {select distinct ?fr
     where {
        {%Person% snvoc:knows ?fr.} union
        {%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr. filter (?fr != %Person%)}
     }
    }
    ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last .
    ?post snvoc:hasCreator ?fr. ?post snvoc:creationDate ?date.
    filter (?date < "%Date0%"^^xsd:date).
    {{?post snvoc:content ?content} union {?post snvoc:imageFile ?content}} .
```

```
}
order by desc (?date) ?post
limit 20
```

- Query 10

```
select ?first ?last
       (((select  count (*)
          where {
              ?post snvoc:hasCreator ?fof . ?post snvoc:hasTag ?tag .
              %Person% snvoc:hasInterest ?tag
          }
         ))
         -
         ((select  count (*)
          where {
              ?post snvoc:hasCreator ?fof . ?post snvoc:hasTag ?tag .
              filter (!exists {%Person% snvoc:hasInterest ?tag})
          }
         )) as ?score )
        ?fof  ?gender ?based
where {
    {select distinct ?fof
     where {
        %Person% snvoc:knows ?fr . ?fr snvoc:knows ?fof . filter (?fof != %Person%)
        minus { %Person% snvoc:knows ?fof } .
     }
    } .
    ?fof snvoc:firstName ?first . ?fof snvoc:lastName ?last . ?fof snvoc:gender ?gender .
    ?fof snvoc:birthday ?bday . ?fof snvoc:isLocatedIn ?based .
    filter (1 = if (bif:month (?bday) = %HS0%, if (bif:dayofmonth(?bday) > 21, 1, 0),
                if (bif:month (?bday) = %HS1%, if (bif:dayofmonth(?bday) < 22, 1, 0), 0)))
}
order by desc(3) ?fof
limit 10
```

- Query 11

```
select ?first ?last ?startdate ?orgname ?fr
where {
    ?w snvoc:hasOrganisation ?org . ?org foaf:name ?orgname . ?org snvoc:isLocatedIn %Country% .
    ?fr snvoc:workAt ?w . ?w snvoc:workFrom ?startdate . filter (?startdate < %Date0%) .
    {select distinct ?fr
     where {
        {%Person% snvoc:knows ?fr.} union
        {%Person% snvoc:knows ?fr2. ?fr2 snvoc:knows ?fr. filter (?fr != %Person%)}
     }
    } .
    ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last .
}
order by ?startdate ?fr ?orgname
limit 10
```

- Query 12

```
select ?exp ?first ?last ?tag count (*)
where {
    %Person% snvoc:knows ?exp . ?exp snvoc:firstName ?first . ?exp snvoc:lastName ?last .
    ?reply snvoc:hasCreator ?exp . ?reply snvoc:replyOf  ?org_post .
    filter (!exists {?org_post snvoc:replyOf ?xx}) .
    ?org_post snvoc:hasTag ?tag . ?tag a %TagType% .
}
group by ?exp ?first ?last ?tag
order by desc(5) ?exp ?tag
limit 20
```

A.2 DEX

- Implementation of query 5 in dex.

```
// We first obtain the person with the personId specified by parameter.
v.setLongVoid(personId);
long personOID = graph.findObject(personId, v);

// We obtain all friends and friends of friends of the person.
Objects friends = graph.neighbors(personOID, knows, EdgesDirection.Outgoing);
Objects allFriends = graph.neighbors(friends, knows, EdgesDirection.Outgoing);
allFriends.union(friends);
allFriends.remove(personOID);
friends.close();

// We obtain all the hasMember relations where allFriends are involved.
Objects members = graph.explode(allFriends, hasMember, EdgesDirection.Ingoing);

// We select those, from these hasMember relations, those whose join date is greater or equal than date.
// and obtain the groups.
v.setTimestampVoid(date);
Objects candidate = graph.select(joinDate, Condition.GreaterEqual, v, members);
Objects finalSelection = graph.tails(candidate);
candidate.close();
members.close();

// We obtain the posts created by allFriends.
Objects posts = graph.neighbors(allFriends, hasCreator, EdgesDirection.Ingoing);

// For each group
ObjectsIterator iterator = finalSelection.iterator();
while (iterator.hasNext()) {
  long oid = iterator.next();
  Container c = new Container();

  // We obtain all the posts and the moderator of that group.
  Objects postsGroup = graph.neighbors(oid, containerOf, EdgesDirection.Outgoing);
  Objects moderators = graph.neighbors(oid, hasModerator, EdgesDirection.Outgoing);
  long moderatorOid = moderators.any();
  moderators.close();

  // We obtain the posts created by the moderator.
  Objects postsModerator = graph.neighbors(moderatorOid, hasCreator, EdgesDirection.Ingoing);

  // We remove the posts from the moderator from the posts of the group.
  postsGroup.difference(postsModerator);
  postsModerator.close();

  // We calculate the size of the intersection between the posts of the friends, and the final post collection
  postsGroup.intersection(posts);
  long count = postsGroup.size();

  if (count > 0) {
    // We get the attributes needed for the projection.
     graph.getAttribute(oid, forumId, v);
     c.row[0] = db.getForumURI(v.getLong());
     c.compare2 = String.valueOf(v.getLong());
     c.row[1] = String.valueOf(count);
     c.compare = count;
     results.add(c);
  }

  postsGroup.close();
```

```
}

// We close the collections that remain opened.
iterator.close();
posts.close();
finalSelection.close();
allFriends.close();

Collections.sort(results);
```

A.3 Neo4J

- Query 1

```
MATCH (person:Person {firstName:{person_first_name}})
WITH person
ORDER BY person.lastName
LIMIT {limit}
MATCH (person)-[:IS_LOCATED_IN]->(personCity:Place:City)
MATCH (uniCity:City)<-[:IS_LOCATED_IN]-(uni:University)<-[studyAt:STUDY_AT]-(person)
WITH collect(DISTINCT (uni.name + ', ' + uniCity.name+ '(' + studyAt.classYear + ')'))
    AS unis, person, personCity
MATCH (companyCountry:Place:Country)<-[:IS_LOCATED_IN]-(company:Company)<-[worksAt:WORKS_AT]-(person)
WITH collect(DISTINCT (company.name + ', ' + companyCountry.name + '('+ worksAt.workFrom + ')'))
    AS companies, unis, person, personCity
RETURN person.firstName AS firstName, person.lastName AS lastName, person.birthday AS birthday,
    person.creationDate AS creation, person.gender AS gender, person.languages AS languages,
    person.browserUsed AS browser, person.locationIP AS ip, person.email AS emails,
    personCity.name AS personCity, unis, companies
```

- Query 2

```
MATCH (:Person {id:{person_id}})-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(post:Post)
WHERE post.creationDate<={max_date}
RETURN friend.id AS personId, friend.firstName AS personFirstName, friend.lastName AS personLastName,
    post.id AS postId, post.content AS postContent, post.creationDate AS postDate
ORDER BY postDate DESC
LIMIT 20
```

- Query 3

```
MATCH (person:Person)-[:KNOWS*1..2]-(friend:Person)
WHERE person.id={person_id}
MATCH (friend)<-[:HAS_CREATOR]-(postX:Post)-[:IS_LOCATED_IN]->(countryX:Country)
WHERE countryX.name={country_x} AND postX.creationDate>={min_date} AND postX.creationDate<={max_date}
WITH friend, count(DISTINCT postX) AS xCount
MATCH (friend)<-[:HAS_CREATOR]-(postY:Post)-[:IS_LOCATED_IN]->(countryY:Country)
WHERE countryY.name={country_y} AND postY.creationDate>={min_date} AND postY.creationDate<={max_date}
WITH friend.firstName + ' ' + friend.lastName  AS friendName , xCount, count(DISTINCT postY) AS yCount
RETURN friendName, xCount, yCount, xCount + yCount AS xyCount
ORDER BY xyCount DESC
```

- Query 4

```
MATCH (person:Person)-[:KNOWS]-(friend:Person)
WHERE person.id={person_id}
MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
WHERE post.creationDate>={min_date} AND post.creationDate<={max_date}
```

```
MATCH (post)-[HAS_TAG]->(tag:Tag)
WITH DISTINCT tag, collect(tag) AS tags
RETURN tag.name AS tagName, length(tags) AS tagCount
ORDER BY tagCount DESC
LIMIT 10
```

- Query 5

```
// Can be expressed as single query Neo4j 2.0 - need to update this

MATCH (person:Person)-[:KNOWS*1..2]-(friend:Person)
WHERE person.id={person_id}
MATCH (friend)<-[membership:HAS_MEMBER]-(forum:Forum)
WHERE membership.joinDate>{join_date}
MATCH (friend)<-[:HAS_CREATOR]-(comment:Comment)
WHERE (comment)-[:REPLY_OF*0..]->(:Comment)-[:REPLY_OF]->(:Post)<-[:CONTAINER_OF]-(forum)
RETURN forum.title AS forum, count(comment) AS commentCount
ORDER BY commentCount DESC

MATCH (person:Person)-[:KNOWS*1..2]-(friend:Person)
WHERE person.id={person_id}
MATCH (friend)<-[membership:HAS_MEMBER]-(forum:Forum)
WHERE membership.joinDate>{join_date}
MATCH (friend)<-[:HAS_CREATOR]-(post:Post)<-[:CONTAINER_OF]-(forum)
RETURN forum.title AS forum, count(post) AS postCount
ORDER BY postCount DESC
```

- Query 6

```
MATCH (person:Person {id:{person_id}})-[:KNOWS*1..2]-(:Person)<-[:HAS_CREATOR]-(post:Post),
    (post)-[:HAS_TAG]->(tag:Tag {name:{tag_name}})
WITH DISTINCT post
MATCH (post)-[:HAS_TAG]->(tag:Tag)
WHERE NOT(tag.name={tag_name})
RETURN tag.name AS tagName, count(tag) AS tagCount
ORDER BY tagCount DESC
LIMIT 10
```

- Query 7

```
MATCH (person:Person {id:{person_id}})-[:IS_LOCATED_IN]->(:City)-[:IS_LOCATED_IN]->(country:Country),
    (country)<-[:IS_LOCATED_IN]-(post:Post)-[:HAS_CREATOR]->(creator:Person),
    (post)-[:HAS_TAG]->(tag:Tag)
WHERE NOT((person)-[:KNOWS]-(creator)) AND
    post.creationDate>{min_date} AND post.creationDate<{max_date}
RETURN DISTINCT tag.name AS tagName, count(tag) AS tagCount
ORDER BY tagCount DESC
LIMIT 10
```