LDBC

Cooperative Project

FP7 – 317548

D3.3.2 Graph database infrastructure and language expressivity

Coordinator: [Norbert Martínez]

With contributions from: [Renzo Angles (VUA), Alex Averbuch (NEO), Orri Erling (OGL)]

1st Quality Reviewer: Peter Boncz (VUA)
2nd Quality Reviewer: Irini Fundulaki (FORTH)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M12
Actual delivery date:	M12
Version:	1.0
Total number of pages:	39
Keywords:	graph database, infrastructure, software environment, hardware environment, query language

Abstract

Analysis and exploration of large graphs is a very active area of research and innovation. The classical single computer in-memory model is not enough for the huge amount of relationships in some real use case scenarios like social networks. The requirements for graph database benchmarking come from different areas. In this deliverable we analyze these requirements with different approaches. First, there is a description of the multiple initiatives to store, manage and query those large graphs in different environments and platforms, such as shared-memory multiprocessors, distributed graphs, parallel computation models, etc. There is also a discussion of the performance factors and how workload types relate to them, focusing on the query execution and optimization challenges for graph-structured data. Finally, we analyze some of the most important graph query languages and propose an agnostic method to formalize graph queries in a graph database benchmark.

Executive Summary

The development of huge networks such as the Internet, geographical systems, protein interaction, transportation or social networks, has brought the need to manage information with inherent graph-like nature. In these scenarios, users are not only keen on retrieving plain tabular data from entities, but also relationships with other entities using explicit or implicit values and links to obtain more elaborated information. In addition, users are typically not interested in obtaining a list of results, but a set of entities that are interconnected satisfying a given constraint. Under these circumstances, the natural way to represent results is by means of graphs.

Deliverable D3.3.1 presented a representative list of use cases potentially useful for graph database benchmarking. They were not an exhaustive list of possible real scenarios where graph databases could be applied, but rather a selection of the most representative ones in terms of impact in the industrial community, the type of operations performed and the kind of graphs managed. Some of the presented use cases were, for example, Social network analysis (SNA), Information Technologies Analysis, EU Projects Analysis, or Geographical routing, among others. In these use cases scenarios, the ability of the new storage and communication systems to process and manage large amounts of real time linked data, in the order of trillions of relationships, is leading towards a extreme situation for the current graph analytical technologies.

Classical single-machine in-memory solutions and libraries for reasoning over graphs are often infeasible. Some of the challenges we identify are: (i) large graphs do not fit into a single physical memory address space; (ii) performance with very large graphs due to the random memory access behaviour on most algorithms; (iii) it is not easy to develop efficient implementations in external memory, parallel process and distributed computing; and (iv) on-line (ad-hoc) queries over large dynamic graphs with small latency is also a requirement. There are also specific requirements for some environments. For example, in distributed graphs the user must deal with cluster management and administration, fault tolerance, complexity in query processing due to the the amount of communication, debugging and optimization, etc.

There are multiple environments and computational models for graph analytics. For example, the *Symmetric Multiprocessor (SMP) Architecture*, where several processors operate in a shared memory environment and are packaged in a single machine. In this architecture, graph algorithms are split in many independent (parallel) calculations, but usually with random memory access patterns, and this is its main problem because multicore CPUs requires a cache-conscious query processing. SMP solutions try to solve this by enhancing memory locality and cache utilization through cache-conscious data structures; decoupling computation and inter-core communication; hiding memory latencies by prefetching data; and using native mechanisms such as atomic instructions and thread and memory affinity. Another example is *Distributed Graphs* in a distributed group of networked computers. Its main advantage is the increase of the parallel capabilities because the number of processors is not limited by the physical architecture of a shared-memory computer, but partitioned graphs are more difficult to manage and analyze. Also, there are performance penalties incurred due to the network communications because most of the graph algorithms cannot predict the number of steps and the scope or partition of the graph to be explored. A particular distributed environment is the *vertex-centrix model*, where each vertex is a single computation unit and the system provides mechanism to communicate and interact between vertices. This model is based on the Bulk Synchronous Parallel (BSP) model, where at each superstep all the vertices compute in parallel, and then they synchronize by exchanging messages between supersteps. This model favors the parallel computation at each superstep but pays a synchronization penalty between supersteps that can be reduced by using alternative asynchronous methods. Finally, MapReduce is also used for graph analytics. This model performs optimally only when a sequential algorithm that satisfy the restrictions imposed is clearly parallel and can be decomposed into a large number of independent computations. Instead, it fails when there are computational dependencies in the data or the algorithm requires iterative computation with transformed parameters, such as in edge-oriented tasks for path traversals.

As a common factor for all these environments, the two main elements of performance to consider are parallelism and locality. Parallelism can be at thread level or at instruction level, and locality can be spatial or temporal. The price of missing locality is increased latency, the time computation dependent on a data item is blocked. Different workloads (e.g. online web sites or analytics) have very different characteristics concerning parallelism and latency tolerance. Also, graph shaped data generally have less natural spatial locality than for

example relational data, and 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. Some of the classical techniques in DBMS to adapt algorithms to hardware capabilities are intra-query parallelism, vectoring of a large number of inputs in a single operator, compilation to native code, cache conscious algorithms, and memory affinity. These techniques are motivated by the ubiquity of multicore processors and by the great difference in memory access latency between cache and main memory. In the case of graph analytics there are new requirements that need novel approaches. For example, graph workloads are characterized by unpredictable random access and serving such workloads from disk hits extreme latency. For this reason, graph applications need to run from memory. If data size is thereby significantly reduced, memory based compression models are attractive. After exhausting the possibilities of compression, applications are required to go to scale-up or scale-out. Shared memory scale-up systems can accommodate up to a few terabytes of memory. Scale-out becomes unavoidable with large data, specially if one needs to deploy on common hardware, as on clouds. Finally, in SQL and SPARQL it is a common assumption that DBMS-application interactions are relatively few and most often take place over a client server connection. Thus latencies of tens of microseconds are expected and are compensated for by having a large amount of work done by the DBMS independently of the application. This is expected to drive progress in graph databases by highlighting the optimization opportunities in a query language and on the other hand may drive advances in SQL or SPARQL systems for similar logic injection into core query execution.

Declarative languages for graph databases are still an incipient area of research and innovation. For example, SPARQL is the standard query language used in RDF databases proposed by the W3C, and G-SPARQL an extension of SPARQL 1.0 to support special types of path queries. Also, Cypher is the graph query language provided by the Neo4j graph database. Although there are domain specific languages to describe graph analysis algorithm, their imperative nature reduces the optimization possibilities. Some of the capabilities that a declarative graph language should support are *pattern matching* to find subgraphs based on a graph pattern; *reachability queries* characterized by path or traversal problems, with the objective to test whether two given nodes are connected by a path; *aggregate queries* with operations, non related to the data model, that permit to summarize or operate on the query results; and *grouping* to return the result sequence grouped by values of different attributes or relationships. In general, and assuming that the main feature of a query language is the computation of graph pattern matching queries, both SPARQL and Cypher are able to express complex graph patterns, and they also thave the same expressive power or even more than the relational algebra. In any case, none of the current languages can be used for all graph frameworks or environments. Thus, for the specification of graph queries in the design of a benchmark, the query descriptions should be presented as data model agnostic as possible, and they should be described from the abstraction level of the application domain. Some of the elements to describe the query structure are the name, a summary textual description of the query; the detailed description of the query in textual plain English; the list of input parameters; the expected content and format of the query result; a textual functional description of the query, from the abstraction level of the database (not the application domain); and the relevance with a textual plain English of the reasoning for including the query in the workload.

Concluding, benchmarking graph databases must be aware of the diversity of environments and computational models. Choke points for the benchmarks will come from the analysis of the problems that are in common with other database technologies such as the relational model, and the new ones specific for the graph-shaped data. Finally, even with a good definition of the graph query choke points and the different workloads, it is necessary an agnostic representation to express graph queries in a portable form for any graph query environment.

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/wp3/deliverables/D3.3.2_Graph_database_infrastructure_and_language_expressivity/`		
EU Project Officer	Carola Carstens		

Deliverable	Number	D3.3.2	Title	Graph database infrastructure and language expressivity
Work Package	Number	WP3	Title	Graph Choke Point Analysis

Date of Delivery	Contractual	M12	Actual	M12
Status	version 1.0		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 (UPC)			
Responsible Author	Name	Norbert Martínez	E-mail	nmartine@ac.upc.edu
	Partner	UPC	Phone	+34934010967

Abstract (for dissemination)	Analysis and exploration of large graphs is a very active area of research and innovation. The classical single computer in-memory model is not enough for the huge amount of relationships in some real use case scenarios like social networks. The requirements for graph database benchmarking come from different areas. In this deliverable we analyze these requirements with different approaches. First, there is a description of the multiple initiatives to store, manage and query those large graphs in different environments and platforms, such as shared-memory multiprocessors, distributed graphs, parallel computation models, etc. There is also a discussion of the performance factors and how workload types relate to them, focusing on the query execution and optimization challenges for graph-structured data. Finally, we analyze some of the most important graph query languages and propose an agnostic method to formalize graph queries in a graph database benchmark.
Keywords	graph database, infrastructure, software environment, hardware environment, query language

Version Log			
Issue Date	Rev. No.	Author	Change
17/09/2013	0.1	Norbert Martinez	First draft
30/09/2013	1.0	Norbert Martinez	Final version with reviewer recommendations

Table of Contents

1 Introduction

The development of huge networks such as the Internet, geographical systems, protein interaction, transportation or social networks, has brought the need to manage information with inherent graph-like nature. In these scenarios, users are not only keen on retrieving plain tabular data from entities, but also relationships with other entities using explicit or implicit values and links to obtain more elaborated information. In addition, users are typically not interested in obtaining a list of results, but a set of entities that are interconnected satisfying a given constraint. Under these circumstances, the natural way to represent results is by means of graphs.

In the previous deliverable D3.3.1 we presented an introduction to graph concepts and graph databases, a detailed state of the art in graph query languages, and a description of several important use cases for graph storage and mining. Based on these previous analysis, in this deliverable we focus on the requirements for the construction of a graph database benchmark. Thus, in Section 2 we present an analysis of software and hardware environments for the graph use case scenarios introduced in D3.3.1. This analysis includes the comparison of data processing on different architectures for large graph data such as software solutions (e.g. map-reduce) and hardware platforms (e.g. cloud, cluster, multicore...). We present also some of the most important features in the existing query frameworks (e.g BSP such as Pregel or GraphChi).

Section 3 discuss some constituent factors of performance and comments how workload types relate to performance factors. Based on a previous work for the TPC-H benchmark, there is a discussion on how the query execution and optimization challenges can be extended to graph-structured data. Some of the covered areas are hardware DBMS architectures, scale out and data placement, and query specification. At the end some implications for graph benchmark design and workloads stress are presented.

Section 4 describes the basic characteristics and differences between some of the most important graph queries: SPARQL, G-SPARQL and Cypher. There is a detailed description of the syntax, semantics and expressivenes, including pattern matching, reachability and aggregates and grouping. Then Section 5 proposes a agnostic formal query specification for graph benchmarking.

Finally, Section 6 draws some conclusions summarized from the four previous sections.

2 Analysis of Environments for the Use Case Scenarios

Deliverable D3.3.1 presented a representative list of use cases potentially useful for graph database benchmarking. They were not an exhaustive list of possible real scenarios where graph databases could be applied, but rather a selection of the most representative ones in terms of impact in the industrial community, the type of operations performed and the kind of graphs managed. Some of the presented use cases were, for example:

- *Social network analysis (SNA)*, where nodes typically represent people and edges represent some form of social interaction between them, such as friendship, co-authorship, etc. SNA techniques have been effectively used in several areas of interest like social interaction and network evolution analysis, counter-terrorism and covert networks, or even viral marketing
- *Information Technologies Analysis*, based on the significant amount of internal and external data available in an organization. The added value information that can be obtained provides to the organizations with an understanding of the positioning of the world in relation to their knowledge and objectives.
- *EU Projects Analysis*, with a solution that integrates different public databases in a single graph database for the benefit of a better project proposal and analysis, linking EC official data with bibliographic data, and allowing the search for the best partners and the best bibliographic items for the State of the Art of projects.
- *Geographical* because graph databases are well suited for business applications involving geography, routing, and optimization, including: road, airline, rail, or shipping network.

In these use cases scenario, the ability of the new storage and communication systems to process and manage large amounts of real time linked data is leading towards a extreme situation for the current graph analytical technologies. For example, in the blogsphere the exact number of weblogs is unknown, but in 2011 it was estimated[1] that there was more than 172 million blogs with more than 1 million new posts being produced each day. Another example is Facebook[2], the most popular social network, that grown from 100M users in 2008 to 500M users in 2010. Now, in year 2013, Facebook has more than 1.1 billion of users with 655 million daily active users on average. This represents a very large database with 140 billion links and 100 petabytes of photos and videos. Twitter[3], another famous site for microblogging, reported in 2011 an average of more than 140 million tweets sent per day, with a record of 6939 tweets sent in one second just after midnight in Japan on New Year's Day. Finally, another example of large growing datasets is WhatsApp[4], which in 2012 reported that 18 billion messages (7 inbound, 11 outbound to a group) were sent on the last day of the year

In this chapter we analyze the different challenges for graph analysis in large datasets and we overview the state of the art in research and technology solutions to explore and mine these very large amounts of linked data.

[1] http://technorati.com/
[2] https://www.facebook.com
[3] https://twitter.com/
[4] www.whatsapp.com

2.1 Graph Analysis on Large Datasets

With very large graph-like datasets, the situation has changed significantly in the last years and the classical single-machine in-memory solutions and libraries for reasoning over graphs are often infeasible. Some of the challenges we identify are:

- *Capacity*: large graphs do not fit into a single physical memory address space. Social networks and protein interaction graphs are particularly difficult to handle and they cannot be easily decomposed into small parts. Even systems that manage a large number of small graphs, as used in bioinformatics, do not meet the requirements for querying large graphs.
- *Performance*: with very large graphs the performance is poor and usually dominated by memory latency due to the random memory access behaviour on most algorithms.
- *Ease of use*: it is not easy to develop efficient implementations. While researchers are focusing mainly in external memory, parallel process and distributed computing, the definition and implementation of efficient declarative programming languages is still in a preliminary stage
- *On-line (ad-hoc) queries*: querying large dynamic graphs with small latency is also a future requirement. Most of the research still focuses on predefined graph algorithms and analytics over static linked data.

At the same time that the data sizes are growing exponentially and the user requirements are more challeging, the hardware architectures, the communication systems and the computational resources are also evolving very fast. For example, the Cloud provides an undetermined number of dynamic distributed resources that allows for the process of very large volumes but new challenges appear: it is not easy to split the graph across cluster nodes, and finding a cut that minimizes the communication is still an open issue. Also, many real graphs have a substantial amount of inherent locality but this locality is limited due to the skewed vertex degree distribution which makes even more difficult an efficient graph partitioning. In particular, most graphs are scale-free (natural) graphs whose node degrees follow the power law distribution. For example, in DBpedia [4] over 90% nodes have less than 5 neighbors, while a few nodes have more than 100,000 neighbors.

One particular case in distributed systems is the new trend called *Big Data*. Big Data, as stated in [48], is characterized by the three V's: volume, velocity, variety. *Volume* means very large amounts of data, in the range of trillions of interrelated data units, but in most cases it is too big and comes in a *velocity* too fast for an optimal storage and processing. And *variety* is for different data formats that make more complex the integration and linking of multiple data sources into a single repository. If we consider Big Data in a distributed system, then the user must deal with:

- *Cluster management and administration*: instead of a single machine now we have multiple nodes interconnected through LAN or WAN networks.
- *Fault tolerance*: node failure is a common event in a topology with hundreds of servers running continously during days
- *Complexity in query processing*: complexity is now the amount of communication between the processing nodes and not the number of disk I/Os
- *Debugging and optimization*: writing and testing distributed algorithms with unpredictable performance is much more difficult
- *Online processing*: systems are designed more for offline batch processing rather than for interactive ad-hoc queries

Finally, some other new technologies should be taken into account, such as the *Graphics Processing Units* (GPUs), which can be used for parallel processing of matrix-based graph algorithms; or the new storage hardware technologies such as Flash and Non-Volatile Memories. that are more efficient when managing read-only graph data.

In the next sections we present some of the advances in different technologies and computational models such as *Symmetric Multiprocessor* (SMP) architectures, *Distributed Graphs*, the *Vertex-Centric* model based on the Bulk Synchronous Processing (BSP), and finally *MapReduce*.

2.2 Symmetric Multiprocessor (SMP) Architecture

A Symmetric Multiprocessor (SMP) is an architecture where several processors operate in a shared memory environment and are packaged in a single machine. Processors, also known as cores, are connected to shared memory by a high-speed bus with latencies of hundreds of cycles, and almost all modern CPUs have hardware support for synchronization operations between cores. Most high-performance computers are clusters of super-scalar (hyper-pipelined) SMPs.

The different cores can execute concurrent tasks at the same time. Also, some of the architectures provide an extra level of parallelism thanks to Simultaneous Multi Threading (SMT), allowing two threads to share processing resources in parallel on a single core. Thus, the main challenge for graph algorithms is how to exploit the parallel execution capabilities of SMP CPUs.

Graph algorithms, in general, can be split in many independent (parallel) calculations, but usually with random memory access patterns, and this is the main problem because multicore CPUs requires a cache-conscious query processing. A generic SMP processor has two cache levels: L1 is small, on chip, in the range of KB and with latencies of a few cycles; and L2, a little bit large in the range of KB to a few MB, with a latency of tenths of cycles. New architectures, also, have an extra L3 cache level shared across the cores on a socket. In this hierarchy of caches, random access patterns in graph exploration, with very little data reuse, lack of spatial and temporal locality and have long memory latencies and high synchronization costs [15].

Some of the techniques proposed to improve performance in SMP architectures are, among others: (i) enhance memory locality and cache utilization through cache-conscious data structures; (ii) decouple computation and inter-core communication; (iii) hide memory latencies by prefetching data; and (iv), use native mechanisms such as atomic instructions and thread and memory affinity.

In the last years there have been multiple initiatives to exploit SMPs parallelism in graph analysis with different approaches. For example, locality is improved with new approaches in graph representation, basically using the decomposition storage model presented in [21] and fully exploited in relational and XML databases such as MonetDB [17] or Vertica [9]. Some examples of cache-conscious data structures are DEX [38], where the graph is split in multiple key-value stores where the key is always the *oid* (object unique identifier) of a node or an edge, and the value is a scalar or a collection of oids that can be lightweight compressed; GraphChi [31], where the vertices of a graph are split into disjoint intervals, and each interval is associated to a shard that stores all the edges that have destination in the interval, and in the order of their source; SAP HANA [41], where a graph abstraction layer is implemented on top of the column store and provides efficient access to the vertices and edges of the graph, hiding the complexity of the graph representation and access to the application developer; and Trinity [45], which stores the graph in a distributed in-memory key-value store.

Another approach is to improve locality by reducing the number of accesses to disk and main memory, or the number and size of data transfers between caches. For example, compression is used to reduce the size of the graph in the Boost's compressed sparse row (CSR) storage format [46], which stores the graph on disk as adjacency sets of the out-edges of a vertex. And vectorization techniques, such as those implemented in MonetDB/X100 [18], improve the cache usage but are not still being used in graph analysis. Some disk-based systems use a similar approach, such as Graph-Chi [31], which is based on a parallel sliding windows method that requires only a very small number of non-sequential read accesses to the disk. This solution performs well also on SSDs because writes of updated vertices and edges are also scarce.

The parallel computation in Graph-Chi is based on the *Asynchronous Model*. The basic idea of this model is to provide an update function that uses the incident edges and the most recent values of vertices and edges. This function is executed for each of the vertices, iteratively, until a termination condition is satisfied. The ordering of updates usually follows a dynamic selective scheduling.

While the two previous techniques (storage and computation) are independent of the algorithms and the application developer, a third approach is to provide domain-specific languages (DSL) that are concious of the SMP requirements and capabilities. For example, Green-Marl [26] is a high level DSL to write graph analysis algorithms that exploit the data-level parallelism which is tipically abundant in analysis algorithms on large graphs. Green-Marl has a compiler and optimizer to convert high-level graph algorithms into C++. Its programming model is reminiscent to OpenMP, and some of its main features are: (i) architecture dependent optimizations; (ii) selection of parallel regions to determine which parallel iteration is going to be parallelized; (iii) deferred assignment by using temporary properties with a copy back of the final result; and (iv), save BFS children for reverse order traversals [36].

Finally, there are also programming frameworks that provide the facilities in shared-memory systems to easily design and implement efficient scalable parallel graph algorithms. For example, GraphLab [35] is a parallel framework for Machine Learning (ML) which exploits the sparse structure and common computational patterns of ML algorithms. A GraphLab program is composed of a data model with a data consistency model, update functions, sync mechanisms and scheduling primitives. The data model consists on a directed data graph that encodes the problem specific sparse computational structure and the directly modifiable program state, and a globally shared state in the form of a data table, a map between keys and blocks of data. The stateless user-defined update functions define the local computations operated on the data associated with small neighborhoods in the graph. Global aggregations are computed by invoking the sync mechanisms, or they can be computed in background when the algorithm is robust to approximate global statistics.

2.3 Distributed Graphs

A distributed system is a group of networked computers, also named as nodes. While in SMP all processors have access to a shared memory to exchange information between them, in a distributed system each node has its own private memory, and nodes communicate and coordinate their actions by passing messages. In the past, the topology of a distributed system was usually a star-shape organization with a central hub and a fixed number of satellites, each with the same hardware, resources and operating system capabilities. Now, the current trend is to have a *cloud* of a variable number of ubiquitous nodes interconnected through a network (LAN or WAN) with a dynamic topology. Each distributed node can have a different amount of resources and configuration, and the size and the topology of the cloud evolves as nodes enter and leave without restrictions.

The main advantage of a distributed system is the increase of the parallel capabilities because the number of processors is not limited by the physical architecture of a shared-memory computer. Distributed graph systems are those frameworks that store very large graphs in a distributed system and use all the parallel power to solve graph algorithms more efficiently. But distributed graphs are more difficult to manage and analyze, and the problems are more complex than in SMP. For example, locality is SMP was related to caching while in a distributed graph refers to how the graph is split between nodes. Graph partitioning is a very active area in research, but many of the frameworks still use a simple random hashing, where each vertex, its data and adjacencies are assigned to a partition depending on some hash function over the vertex id or content. Another common approach is to use the classical METIS algorithm [28] or distributed variants of it [29], but the main problem is still how to deal with natural graphs with power-law degree distributions that can lead to highly skewed partitions even with a random partition method.

Another important problem is the performance penalties incurred due to the network communications (e.g. message exchange and synchronization). Most of the graph algorithms cannot predict the number of steps and the scope or partition of the graph to be explored. At each step most graph operations do not have locality, and rely exclusively on random accesses. This means that a large amount of messages containing parts of the graph can be sent through the network at each step, and this will depend also on how the graph has been split. Additionally, the runtime of each phase is determined by the slowest machine, where the speed can depend on the hardware variability, the operating system, the number of concurrent services, multi-tenancy (virtualization), the network imbalance, the amount of graph data to explore, or the complexity of the tasks assigned to the node.

One of the most important distributed graph frameworks is Distributed GraphLab [34]. It extends the shared memory GraphLab abstraction presented in Section 2.2. The new proposal relaxes the scheduling requirements and introduces a new distributed data graph with data versioning and a fault-tolerance execution engine. It also incorporates a pipelined distributed locking to mitigate the effects of network latency. In more detail, the graph representation is based on a two-phased partitioning which can be efficiently load balanced on arbitrary cluster sizes. In the first step it creates a preliminary partition using domain specific knowledge or a distributed graph partitioning heuristic. Each partition, known as atom graph, contains enough information to be rebuilt or replicated with independence of other partitions. The initial number of partitions is larger than the number of nodes, and each one is stored as a separate file on a distributed storage system. In a second step, the connectivity structure and file locations of the atoms is stored in an atom index file as a meta-graph, and the atom graphs are loaded and replicated through the network nodes.

A different approach for a distributed graph system is Trinity.RDF [52], a distributed and memory-based graph engine for RDF data that is stored in its native graph form. Trinity.RDF is based on Trinity [45], a distributed in-memory key-value store, and it solves the SPARQL queries as subgraph matching queries with support of graph operations such as random walks or reachability. The RDF graph is randomly split in disjoint partitions by hashing on the vertices, and each RDF statement corresponds to an edge in the graph. To improve locality, each RDF entity is stored in a key-value pair with its out-adjacency and in-adjacency lists. These lists are split in such a way that adjacent nodes are in the same adjacency split. Thus, each machine can retrieve neighbors that reside on the same machine without incurring any network communication. With this graph representation, graph exploration is carried out on all distributed machines in parallel, and SPARQL queries are transformed to a subgraph matching problem. At a final step, the matchings for all individual triple patterns are centralized into a query proxy to produce the final results.

While in the previous approaches the queries were solved by using specific graph exploration mechanisms or standard RDF languages, Horton [44] provides a query language for reachability queries over directed and undirected attributed labelled graphs. This system has a distributed query execution engine with a query optimizer that allows interactive execution of queries on large distributed graphs in parallel. Graph partitions are stored in main memory to offer fast query response time. A reachability query is optimized and the query plan is translated into a finite state machine that is executed by each partition as a synchronous BFS. For each vertex it checks whether the nodes which are local to the partition satisfy the finite state machine. Then also checks if their outgoing edges also satisfy the state machine to decide whether to continue traversing along the path. When a vertex satisfies the final state then it is sent as a result to the client.

Finally, a different approach is followed in Grappa [39], a large-scale graph processing system with commodity processors. It is based on the hardware multi-threading systems programming model (e.g. Cray XMT): large shared global address space and per node private address spaces, explicit concurrency with a large number of lightweight threads, and full-bit synchronization. Grappa does not provide a framework for graph analysis and its main goal is to develop infrastructure to aid implementation of graph processing libraries that are able to manage large graphs on distributed machines.

2.4 The Vertex-centric Model

A particular distributed graph model is the *vertex-centric model* introduced in Pregel [37] as a method to solve graph algorithms in parallel using the bulk-synchronous parallel computation model (BSP) [49]. BSP is defined by three elements: (i) the components, that are able to execute independently a process and to send and receive messages; (ii) a router that delivers the messages between pairs of components in the most efficient way; and (iii) synchronization facilities at regular intervals of time. The computation in BSP consists in a sequence of *supersteps*. At each superstep each component can receive messages, run the process with its local configuration, and deliver new messages to other components. When the superstep finishes after some time units or due to a synchronization method when all components have finished their work, then a new superstep starts. The computation finishes after a stop condition such as a maximum number of supersteps or a global condition raised by the components. Basically, this model favors the parallel computation at each superstep but pays a synchronization penalty between supersteps.

Pregel adapts the BSP model to the resolution of graph algorithms in a vertex-centric approach where the components are the vertices. At each superstep, each vertex computes a user-defined `Compute` function. This function can process the incoming messages, modify the state of the vertex, and send new messages to its neighbors or to other vertices whose identifier is known. This message will be received at the next superstep. Also, at each superstep each vertex can *vote for halt* when no more tasks are expected to be done. Halted vertices will be inactive at the next superstep unless there are messages waiting to awake them. The process finishes after a superstep when all vertices are halted and the message queue is empty.

The input of a Pregel program is a directed graph. Each vertex is identified by a unique string identifier. This identifier is used to partition the graph, where each partition contains a set of vertices and their outgoing edges. Pregel also provides *combiners* that group several messages intented for a vertex in order to reduce the amount of communication, and *aggregators* as mechanisms for global communication, monitoring and data aggregation. At each superstep the vertices provide a value to an aggregator; then the system combines those values using a *reduction* operator; and the resulting value is made available to all vertices in the next superstep. Finally, Pregel also allows topology updates. It provides two mechanisms to achieve determinism and to avoid conflicting requests in the same superstep: partial ordering and user-defined handlers.

Pregel was designed for the Google cluster architecture. It provides a reduced C++ API for in-memory processing of graph partitions. In this model, the activity is focused in the local action of each vertex that improves parallelism. There is not any priority between vertices, and all the synchronization is during the communication between supersteps.

Pregel has become very popular due to its simplicity and focuses in a vertex-centric approach that is natural to a wide range of graph algorithms. But Pregel is a propietary technology from Google and it is not available to the community. Thus, several technologies have appeared in recent years to emulate or even improve the original vertex-centric model. This is, for example, the case of two Apache projects: HAMA [3] and Giraph [1]. Hama, written in Java, is a pure replica of the BSP computing framework implemented on top of HDFS (Hadoop Distributed File System), and includes a graph package for vertex-centric graph computations. Giraph is another Java implementation of Pregel but with several improvements such as master computation, sharded aggregators, edge-oriented input, and out-of-core computation. While Giraph was originally designed to run the whole computation in-memory, it has an LRU policy to swap partitions in an out-of-core scenario.

Another open-source system that emulates Pregel is GPS [43] (for Graph Processing System), a scalable and fault-tolerant execution framework of graph algorithms. GPS is implemented also Java and its compute nodes run HDFS (Hadoop Distributed File System). Basically, it extends Pregel with three features: (i) an extended API to make global computations more efficient and ease to express, where the global objects are used for coordination, data sharing, and statistics aggregation; (ii) a dynamic repartitioning scheme to reassign vertices to different workers during the computation, based on messaging patterns; and (iii) a partition technique across all computing nodes of the large adjacency lists of the vertices with a high degree. These two last extensions reduce the communication between nodes and improve the overall performance.

A different approach for vertex-centric computation is Signal/Collect [47], a scalable programming model for synchronous and asynchronous graph algorithms on typed graphs. In this model, the vertices send *signals* along the edges and then a *collect* function gathers the incoming signals at the vertices to perform some computation. The main difference with respect the classical BSP approach is that this model supports other synchronous and asynchronous execution models. For example, synchronous execution can be score-guided in order to enable the prioritizing of signal/collect operations. There are also two asynchronous executions: the first one gives no guarantees about the order of execution or the ratio of signal/collect operations; and the scheduled asynchronous operations are an extension with operation schedulers that optimize certain measures. Finally, this model also supports other features as conditional edges and computation phases, or aggregation by introducing aggregation vertices that collect the results from all the vertices it needs to aggregate.

2.5 MapReduce

Another distributed computing model is MapReduce (MR) [23, 32], a system that handles the communication, synchronization and failure recovery and hides most of the complexity coming from parallelism. The user writes the program logic in the form of two functions: *map* sequentially outputs the input data in the form of key-value pairs that are automatically reshuffled in groups of all values by key; and *reduce* performs the actual computation on the repartitioned data with the same key and outputs the final result.

MR is built on top of a Distributed File System (DFS), and a complex query requires several MR jobs, where each job represents one global communication round. In general MR performs optimally only when a sequential algorithm that satisfy the restrictions imposed is clearly parallel and can be decomposed into a large number of independent computations. Instead, MR fails when there are computational dependencies in the data or the algorithm requires iterative computation with transformed parameters. In the case of graph processing, MR is suitable for vertex-oriented tasks, while edge-oriented tasks can create a bottleneck in the system because it is difficult to determine the number of global communication rounds required to compute the query.

As in the case of Pregel, MR is also a propietary technology from Google. The equivalent technology from the Apache project is Hadoop [2], which is the open source implementation of MR. Hadoop provides the Distributed File System (HDFS) and PIG, a high level language for data analysis.

An example of a MR-based distributed graph processing framework is Pegasus [27], a Graph Mining library implemented on the top of the Hadoop platform. Pegasus is based on a single primitive called GIM-V (from Generalized Iterated Matrix-Vector multiplication), that has a good scale-up on the number of available machines with a linear running time on the number of edges. In general, many graph mining operations can be expressed as a repeated matrix-vector multiplication and can be written with a combination of the three customizable GIM-V basic operations over a matrix and a vector: *combine2* to combine an element from the matrix with an element from the vector; *combineAll* to combine all the combine2 results for a given vertex, and *assign* to update a value in the vector. The compution is iterated until an algorithm-specific convergence criterion is met.

Another example of MR for large graphs can be found in Surfer[19], a solution that combines the MR primitives with two propagation functions: *transfer* to export information from a vertex to its neighbors, and *combine* to aggregate the received information at a vertex. Combine can be easily parallelized across multiple machines, and the process is an iterative propagation that transfers the information of each vertex to its neighbor vertices iteratively, which is the basic pattern on traversing the graph in parallel.

Finally, a particular vertex-centric implementation on top of MapReduce is GraphX [50], which is based on the Spark [51] data-parallel framework. It extends the Resilient Distributed Datasets (RDD) to introduce the Resilient Distributed Graph (RDG), and provides a collection of computational primitives that are used to implement vertex-centric frameworks such as Pregel with new operations to view, filter and transform graphs. The partitioning of the graph is a tabular representation of a vertex-cut partitioning generated using several simple data-parallel heuristics for edge partitioning such as random vertex cuts based on edge hashing.

3 Choke Points for Hardware and Software Stress

TPC H Analyzed [16] offers an extensive analysis of the query execution and optimization challenges found in complex SQL queries. In the present section we shall discuss how this set of challenges is extended by working with graph-structured data.

We begin with a summary of the constituent factors of performance and further discuss how these principles have been exploited in databases in general. We then outline workload types and comment on how these related to performance factors. Throughout the discussion we highlight the differences of shared memory systems and clusters.

3.1 Elements of Performance

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.

Latency occurs at many different orders of magnitude.

- 0.5 : 1 ns - Access to L1 cache
- 100ns : Miss of lowest level of CPU cache
- - 1-6 us : Blocking for a mutex if needing to switch to kernel for task scheduling
- - 15-30 us : Message round trip between processes in shared memory[1]
- - 60 us : Message round trip between processes over InfiniBand TCP/IP[1]
- - 150 - 300 us : Message round trip over 1Gbit Ethernet TCPIP
- - 100 - 200 us : Read from SSD
- 8000 - 15000 us : Seek from disk

[1] MPI latencies are less but these depend on busy wait on behalf of the process waiting for the message. The time given is for blocking on a socket.

Additionally bandwidth is of importance. For large sequential transfers, we have the following orders of magnitude:

- 10+ GB/s : Memory
- 3-5 GB/s : named pipe between processes in shared memory
- 1 GB/s : QDR InfiniBand TCPIP[2]
- - 300MB/s SS reading
- 100-200 MB/s disk
- 80MB/s - 1Gbit Ethernet

The higher the latency, the more concurrently pending operations and the more mutually independent channels are needed in order to avoid the latency becoming a limiting factor to throughput.

Different workloads have very different characteristics as concerns parallelism and latency tolerance:

- *On-line web site*: A single page impression may cost one disk seek, e.g. up to 20 ms, the rest goes for network latencies and computation so that the total processing time stays under 200 ms.
- *Analytics*: An operation joining and then aggregating tens of millions of data items out of a database of a billion records may take 1 or 2 seconds, e.g. TPC H queries. The data will typically be in RAM.

Lookup workloads have very high natural parallelism, since these are driven by a large number of online users. Latency matters little as long as it is under 200 ms, as there is a minimum of 100 ms or so due to the Internet message round trip usually involved in using an online site. Analytics workloads do not have as much natural parallelism since these are driven by expert users asking complex questions involving large volumes of data. Thus exploitation of parallelism by the DBMS is key to high platform utilization.

3.2 Specifics of Graph-Shaped Data

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. A fact table may be kept in multiple orders for different purposes and some dimensions of the fact table will be more frequently used for selection than others. Ordering a fact table on date, if there is a date column, is for example common.

Graph vertices are assigned identifiers which determine their location in storage, whether disk or memory. Vertices of one type may be assigned consecutive identifiers (DEX [5, 38]) or URI's created by one thread may be given consecutive identifiers (Virtuoso [10, 24]) in the hope that being loaded together will result in being accessed together. Otherwise data gets stored in load order unless it is reclustered in a separate step based on graph structure or other criteria, e.g. geo location if applicable.

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. This overlapping may range from issuing multiple concurrent IO requests to secondary storage to issuing multiple CPU cache misses in parallel.

[2]MPI latencies are less but these depend on busy wait on behalf of the process waiting for the message. The time given is for blocking on a socket.

3.3 Hardware-conscious DBMS architectures

Database research is largely the science of performance. Therefore adaptation of algorithms to hardware capabilities has always been pursued. At present, the following techniques may be mentioned:

- *Intra-query parallelization* - Queries are broken into independently executable, loosely coupled units of work that are scheduled on a group of worker threads.
- *Vectoring* -Executing a single operator on a large number of input values in one invocation has the following advantages:
    1. interpretation overheads, e.g. cost of interfaces is amortized, the interface is crossed seldom.
    2. operators may gain instruction level parallelism from out of order execution, specially when these consist of tight loops. A specially noteworthy case is that of missing the CPU cache on multiple lines at the same time, thus the latency is seen only once for a number of misses. This is the case for example in hash based operators like hash join or group by where an out-of cache hash table is accessed simultaneously at multiple points.
    3. SIMD instructions may be used for arithmetic.
    4. Random index lookups, if enough are made at the same time, can be ordered and if the retrieved places exhibit locality, a n * log n set of index lookups approaches n + log n complexity. This may speed up index based access by up to 10x (Virtuoso).
- *Compilation* - Queries may be compiled to native machine code. This may be done with or without vectoring. If combined with vectoring, this removes the overhead of writing intermediate results of computations to memory, as these are carried in registers. Naturally interpretation overhead is entirely eliminated and boundaries of operators may be be broken down. (HYPER)
- *Cache conscious algorithms* - Buffers for intermediate results, hash tables etc may be aligned to cache lines and the chunk size may be set to correspond to CPU cache size.
- *Memory affinity* - Scale-up systems usually have multiple CPU sockets, each with local memory. Colocating processing with memory improves memory throughput. For this reason, it may be advantageous to partition data similarly to a scale out system also in shared memory.

These techniques are motivated by the ubiquity of multicore processors and by the great difference in memory access latency between cache and main memory. Specially for graph applications, we may mention a radical departure from today's accepted CPU design principles, namely Cray XMT [30], later Yarc Data [11] and its Threadstorm processor. This is a massive NUMA (non uniform memory architecture) system with 128 hardware threads on each core and no CPU cache. Memory access is interleaved so that consecutive memory words are on different CPU sockets, thus applications have no possibility of exploiting CPU to memory affinity.

Instead, applications are expected to create massive numbers of threads so that even though each thread is almost always waiting for memory the CPU's are kept busy by having many threads. This is an extreme case of the SPARC T2 and T3 concept. With XMT, very high aggregate memory throughput can be obtained as long as there are enough threads. However, a conventional DBMS optimized for cache would run very poorly on such a platform. On the other hand, for applications requiring scale-out size memory and consisting chiefly of unpredictable random access, XMT may offer substantially better performance than a cluster with InfiniBand and explicit code (e.g. MPI) for accessing remote data.

3.4 Scale Out, Latency and Data Placement

Since graph workloads are characterized by unpredictable random access, serving such workloads from disk hits extreme latency. While some sequential access dominated workloads may be run from secondary storage only having enough disks in parallel, random access dominated applications are excessively penalized. For this reason, graph applications need to run from memory. If data size is thereby significantly reduced, memory based compression models are attractive. For example, bitmap or delta compression for vertex lists may be applied.

After exhausting the possibilities of compression, applications are required to go to scale-up or scale-out. Shared memory scale-up systems can accommodate up to a few terabytes of memory, in high end cases 16 TB (Silicon Graphics Ultraviolet) or more e.g. Cray XMT. Such systems cost more than a commodity cluster with the same memory. Also scale-up systems tend to have less raw CPU power for the price.

Scale-out becomes unavoidable with large data, specially if one needs to deploy on common hardware, as on clouds. GDB's rarely support scale-out, only InfiniteGraph [7] is known to do so. However graph analytics frameworks generally do. Many RDF databases do support scale out, e.g. Virtuoso, BigData, 4Store.

Scale-out has been explored in many variants in the SQL and RDF spaces. With SQL, there are two main approaches, *cache fusion* (Oracle RAC) and *partitioning* (DB2, Vertica, most others). Further, diverse schemes of fault tolerance and redundancy either based on shared storage (Oracle RAC) or partitions in duplicate (Greenplum) have been deployed. On the RDF side, scale out is generally based on partitioning the data by range or hash of the subject or object or a triple, whichever is more conveniently placed. Thus different orderings of the triple or quad table have the same quad in different partitions.

Due to the prevalently API driven usage of GDB's, scale out is less attractive as it might involve a network round trip per API call, e.g. return outbound edges of a vertex. This is near-unfeasible even on a single server with shared memory communication, not to mention with a fast network, we are faced with latency in the tens of microseconds in only OS scheduling and task switching overheads.

On the other hand, when sending messages that represent hundreds of thousands of lookups in a single message, the latency stops being a factor and the platform is efficiently expoited, e.g. by vectoring and intra-query parallelism.

Scale-out support de facto requires some form of vectoring in order to amortize communication overheads and latency. If at least several hundred operations are carried in one message exchange the overhead becomes tolerable. The latency of one message exchange in shared memory corresponds to 50-100 random index lookups from an index of 1 billion entries (measure with Virtuoso). This observation outlines the de facto unfeasibility of single tuple operations over scale-out, even when no actual network is involved.

Graph analytics frameworks generally use a bulk synchronous processing (BSP) model. In such a model, computation is divided in supersteps, where each superstep updates the state of the graph and produces output that serves as input for the next superstep. Thus when messages between non-colocated vertices are required, these can be easily combined. Network latency is not generally an issue, only throughput. There is usually a synchronization barrier between supersteps, i.e. all communication must be concluded before start of computation. Some systems support more flexible scheduling.

3.5 API vs. Query Language

The situation in GDB query languages is in flux. Some systems offer declarative query, e.g. Neo4j Cypher [8] but generally all systems offer API based access.

In SQL and SPARQL it is a common assumption that DBMS-application interactions are relatively few and most often take place over a client server connection. Thus latencies of tens of microseconds are expected and are compensated for by having a large amount of work done by the DBMS independently of the application. In SQL databases, in-process API's sometimes exist but these are for run-time hosting and usually offer the same query language as the client server interface, only now without a client-server round trip.

With GDB's on the other hand it is customary to have a much tighter coupling between DBMS and application. Thus it is possible to embed application logic in any part of query processing, e.g. one could decide to traverse the lowest weight edges first when looking for a shortest path. Such logic is not easily expressible in SQL or SPARQL. With SQL or SPARQL, such things would usually be done on the application side, leading to loss of performance via increased numbers of client-server round trips.

On the other hand, an SQL or SPARQL system receiving a query with aggregation over many nested joins can execute such queries on multiple threads per query and can use vectoring (IBM DB2, VectorWise [53], Virtuoso, most column stores) or full materialization (MonetDB [17]). Furthermore, use of hash join for selective joins is common.

The equivalent with a GDB API would call for nested loops where the API would be crossed at least for each new vertex whose in or outbound edges are to be traversed. This loses potential performance gains from vectoring, i.e. it is cheaper to retrieve a batch of 1 million outbound edge lists than one outbound edge list one million times.

A benchmark needs to create incentives for streamlining API's while at the same time highlighting the advantages of tight embedding of application logic and query execution.

This is expected to drive progress in GDB's by highlighting the optimization opportunities in a query language and on the other hand may drive advances in SQL or SPARQL systems for similar logic injection into core query execution.

3.6 Implications for Benchmark Design

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.

As noted above, performance results from parallelism and locality. Workloads need to stress these two aspects, 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 oreferentially lend themselves to either style of processing.

- *Graph analytics* operations that typically would be implemented in a graph analytics framework. BSP-style processing can be implemented on top of a DBMS, as for instance has been done with Virtuoso. These workloads do have high intrinsic parallelism and locality. Even though the graph structure may exhibit arbitrary connectedness the fact of touching most vertices with no restrictions on order of access provides both parallelisn 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 incented to co-locate items that are frequently accessed together, e.g. a person and the person's posts. A relational schema may imply colocation by sharing a partitioning column between primary and foreign keys. But since GDB's and RDF systems typically do not have a notion of multipart primary key, this technique is not available. Hence other approaches will have to be explored. The matter of data location also impacts single server situations but is less acute there, due to lower latency.

4 Expressive Power of Query Languages

In this chapter we present a comparison of the syntax, semantics and capabilities (expressiveness) of the most important query languages available in the market. We consider three declarative query languages in our comparison: SPARQL, G-SPARQL, and Cypher. First, we briefly describe the syntax and semantics of the languages. Then, we compare the languages by presenting their capabilities to express several types of graph-oriented queries, all of them well-studied in the literature on graph databases.

Additionally, we present a query specification format which is proposed to describe the queries to be used in the design of a graph benchmark in LDBC. The specification is based on a textual description of the query and the definition of parameters, results, functionality, and relevance.

4.1 Graph query languages

In this section we describe the syntax and semantics of SPARQL 1.0, SPARQL 1.1, G-SPARQL and Cypher. SPARQL is the standard query language used in RDF databases proposed by the W3C. G-SPARQL is an extension of SPARQL 1.0 to support special types of path queries. Cypher is the graph query language provided by the Neo4j graph database.

Note that, although we recognize the existence and importance of domain specific languages to describe graph analysis algorithm, e.g. Gremlin[6] and Green-Marl[26], they are not included in this comparison. The main reason of our decision is the imperative nature of such languages, that makes them incomparable with declarative query languages.

SPARQL 1.0

SPARQL [40] is the standard query language for RDF databases (also named RDF triple stores). RDF data is based on three data domains: the domain of RDF resources, which contains data entities each one identified by a Uniform Resource Identifier (URI); the domain of RDF literals, which includes simple atomic values (e.g. strings, numbers, dates, etc.); and the domain of RDF black nodes, which contains anonymous resources (this domain is not considered in this document).

RDF defines a graph data model based on the notion of RDF triple. An *RDF triple* is an expression of the form *{ subject predicate object }* where the *subject* is a URI referencing a resource, the *predicate* is a URI referencing a property of the resource, and the *object* is either a URI or a Literal representing the property value. Assuming that subjects and objects can be represented as nodes and predicates as edges, a collection of RDF triples is called an *RDF graph*. A collection of RDF graphs is called an *RDF dataset*.

A SPARQL query allows complex graph pattern matching over multiple data sources, and the output can be a results multiset (i.e. allowing duplicates) or RDF graphs. A query is syntactically represented by a block consisting of zero or more prefix declarations, a query form (e.g. SELECT), zero or more dataset clauses (e.g. FROM) , a graph pattern expression, and possibly solution modifiers (e.g, ORDER BY). For example, the following expression defines a query over the RDF graph identified by URI `http://www.socialnetwork.org/`, and returns a multiset including the first name and age of persons whose age is greater than 18, and sorted by first name:

```
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N ?A
FROM <http://www.socialnetwork.org/sndata.rdf>
WHERE { ?X sn:type sn:Person . ?X sn:firstName ?N . ?X sn:age ?A . FILTER (?A > 18) }
ORDER BY ?N
```

Informally, the evaluation of a SPARQL query consists in the following procedure: to construct an RDF dataset based on the dataset clauses; to evaluate the graph pattern over the RDF dataset, which results in a multiset of solution mappings; to modify the solution mappings according to the solution sequence modifiers; and to prepare the query output according to the query form.

A `PREFIX` clause allows to assign a prefix (e.g. `sn`) to a URI (e.g. `http://www.socialnetwork.org/`). Prefixes are syntactic sugar to simplify the representation of URIs in SPARQL queries (e.g. `sn:firstName`). The set of dataset clauses allows to define the RDF dataset to be used in the query. In our example, the FROM dataset clause allows to define that the dataset of the query will include the RDF graph referenced by the URI `http://www.socialnetwork.org/sndata.rdf`.

The main element in a SPARQL query is the graph pattern expression defined in the WHERE clause. The most basic form of graph pattern is a *triple pattern*, which extends the definition of RDF triple by allowing variables (e.g `{?X sn:firstName ?N}`). Complex graph patterns are defined recursively as the combination of triple patterns with special operators. Assuming that P_1 and P_2 are graph patterns, and C is a filter condition (e.g. $?X > 18$), the expressions { P_1 . P_2 }, { P_1 UNION P_2 }, { P_1 OPTIONAL P_2 }, and { P_1 FILTER C } are complex graph patterns.

The evaluation of a SPARQL graph pattern is based on the notion of solution mappings. A *solution mapping* is a partial function μ from a set of variables to a set of RDF terms (i.e. URIs and literals). Hence, we use $\mu(?X) =$ "George" to denote that variable $?X$ is assigned with literal "George". Two solution mappings μ and μ' are compatible if and only if for every variable $?X$ shared by μ and μ', it applies that $\mu(?X) = \mu'(?X)$, i.e., the union of compatible mappings $\mu \cup \mu'$ is also a mapping.

The evaluation of a triple pattern T returns a set of solution mappings Ω such that, each solution mapping $\mu \in \Omega$ satisfies that the instantiation of T, by replacing variables according to μ, results in an RDF triple that occurs in the RDF dataset of the query. For example, given the triple pattern $T =$ {$?X$ `sn:firstName` $?N$}, and assuming that the RDF dataset contains the RDF triple [sn:Person1, sn:firstName, "Thomas"], a solution mapping μ_1 is part of the set of solutions for T if and only if $\mu_1(?X)$ = sn:Person1 and $\mu_1(?N)$ = "Thomas".

Assume that P_1 and P_2 are graph patterns, and Ω_1 and Ω_2 are their multisets of solutions mappings respectively. The evaluation of a complex graph pattern, denoted $eval(.)$, is defined as follows:

- $eval(\{P_1$. $P_2\})$ = { $\mu_1 \cup \mu_2$ | $\mu_1 \in \Omega_1$, $\mu_2 \in \Omega_2$, and μ_1 is compatible with μ_2 }
- $eval(\{P_1$ UNION $P_2\})$ = { μ | $\mu \in \Omega_1$ or $\mu \in \Omega_2$ }
- $eval(\{P_1$ OPTIONAL $P_2\})$ = $eval(\{P_1$. $P_2\}) \cup$ { $\mu_1 \in \Omega_1$ | μ_1 is not compatible with all $\mu_2 \in \Omega_2$ }
- $eval(\{P_1$ FILTER $C\})$ = { $\mu_1 \in \Omega_1$ | μ_1 satisfies the filter condition C }

Note that the evaluation of a complex graph pattern can result in a multiset of solutions mappings, i.e. duplicate solutions are allowed. The *solution modifiers* (ORDER BY, DISTINCT, OFFSET, LIMIT) can be used to restrict and format the original multiset of solutions. For example, the operator DISTINCT can be used to eliminate duplicates, i.e. for transforming the multiset of mappings into a set.

The *query form* allows to define the output of the query: a SELECT query form allows to project the variables of the graph pattern and returns a solutions sequence; a CONSTRUCT query form allows to construct an RDF graph with the results of the graph pattern matching; an ASK query form returns "false" where the result of evaluating the graph pattern is empty, and "true" otherwise.

SPARQL 1.1

The W3C specification of SPARQL 1.1 was released on March 2013. This version extends SPARQL 1.0 with the following features: explicit operators to express negation of graph patterns, arbitrary length path matching (i.e. reachability), aggregate operators (e.g. COUNT), subqueries, and query federation.

It has been shown [14] that the negation of graph patterns can be expressed in SPARQL 1.0 by a combination of the OPTIONAL and FILTER operators. In SPARQL 1.1, the negation can be explicitly expressed by using two types of expression, { P_1 MINUS P_2 } and { P_1 FILTER NOT EXISTS P_2 }, where P_1 and P_2 are graph patterns. In both cases, the evaluation returns a subset of the solution mappings of $eval(P_1)$ satisfying that they are incompatible with every solution mapping occurring in $eval(P_2)$.

SPARQL 1.1 introduces the notion of property paths as a feature to find a route between two nodes in the RDF graph. A *property path* is an expression of the form *{ subject regex object }* where *subject* is the source node of the path (URI or variable), *object* is the target node of the path (URI, literal or variable), and *regex* is a regular expression representing the path pattern. A expression (URI) is a basic regular expression where URI

references a property. Assuming that P and Q are regular expressions, the following operations are defined to produce recursively complex regular expressions:

- (P/Q) : which represents the concatenation of paths.
- $(P|Q)$: which represents the alternation of paths.
- $!(P)$: which represents the negation of a path.
- $(P)?$: which represents a path containing P, zero or one times.
- $(P)*$: which represents a path containing P, zero or more times.
- $(P)+$: which represents a path containing P, one or more times.

Property paths containing complex regular expression can be constructed by nesting the above basic regular expressions, e.g. $!(P/(Q|R))$.

The evaluation of a property path tries to find a connection, between the source node and the target node, as defined by the regular expression (i.e. following specific properties, a given number of times). For example, the expression `{ sn:Person1 sn:knows+ ?X }` returns the nodes (persons) which are reachable from the node `sn:Person1`, by following the property `sn:knows`, one or more times. The evaluation of a property path does not introduce duplicate solutions.

The following aggregate operators are supported in SPARQL 1.1.: COUNT, SUM, MIN, MAX, and AVG. Additionally, the operators GROUP BY and HAVING are allowed to apply restrictions over groups of solutions.

G-SPARQL

G-SPARQL [42] is a query language for querying attributed graphs, based on the syntax and semantics of SPARQL 1.0. An *attribute graph* is a graph where nodes and edges are allowed to have an arbitrary number of attributes. Hence, the graph data (values for attributes) are represented differently from the structural information of the graph (edges).

G-SPARQL extends SPARQL 1.0 with two main features: (1) graph patterns where value-based conditions can be applied on the attributes of nodes and edges; and (2) path pattern expressions allowing filtering conditions over the path pattern (e.g. value-based conditions over the attributes of vertices and/or edges in the path) and constraints on the path length.

The G-SPARQL syntax uses the symbol "@" to represent attributes of nodes and edges and differentiate them from the standard structural predicates (properties). Conditions over attributes can be defined by using two types of value-based predicates: *vertex predicates* which allow restrictions on the attributes of the graph nodes, e.g. `{?Person @firstName "George"}`; and *edge predicates* which allow conditions on the attributes of graph edges, e.g. `{?Person ?E(studyAt) ?Univ . ?E @classYear "2000"}`.

A path pattern in G-SPARQL is an expression *{ subject path object }* where *subject* is the source node of the path, *object* is the target node of the path, and $path$ is any of the following path expression: ??P , ?*P , ??P(*predicate*) , or ?*P(*predicate*). In a path expression: ??P is a path variable which indicates that the matching paths between the *subject* and the *object* can be of any arbitrary length; ?*P is a path variable that will be matched with the shortest path between *subject* and *object*; and *predicate* defines the relationship (edge) that the matching paths must satisfy.

G-SPARQL allows graph patterns of the form $(P_{path}$ FILTERPATH $C_{path})$ to apply a path condition C_{path} over a path pattern P_{path}. Assume that V_{path} is a path variable, N is a number, C_{eq} is a equality/inequality condition (e.g. $= 1$ or > 1), C_v is a valued-based condition (i.e. a value condition over an attribute). A *path condition* is one of the following expressions:

- `Length(`V_{path}`,`C_{eq}`)`: It allows to filter the paths bounded to path variable V_{path} by their length (number of edges) according to the equality/inequality condition C_{eq}, e.g. `Length(??X, <4)`.

- `AtLeastNode(`V_{path}`,`N`,`C_v`)`: It verifies that at least N number of nodes on each path bounded to V_{path} satisfy the value-based condition C_v, e.g. `(AtLeastNode(??X, 3, @gender "male"))`.
- `AtMostNode(`V_{path}`,`N`,`C_v`)`: It ensures that at most N number of nodes on each path bounded to V_{path} satisfies the value-based condition C_v.
- `AllNodes(`V_{path}`,`C_v`)`: It ensures that every node of each path bounded to V_{path} satisfies the value-based condition C_v.
- `AtLeastEdge(`V_{path}`,`N`,`C_v`)`: It verifies that at least N number of edges on each path bounded to V_{path} satisfies the value-based condition C_v.
- `AtMostEdge(`V_{path}`,`N`,`C_v`)`: It ensures that at most N number of edges on each path bounded to V_{path} satisfies the value-based condition C_v.
- `AllEdges(`V_{path}`,`C_v`)`: It ensures that every edge of each path bounded to V_{path} satisfies the value-based condition C_v.

Cypher

Neo4j is an open-source graph database that defines a query language called Cypher [8]. Cypher is a declarative graph query language, designed to be "humane" and intuitive. Its constructs are based on English prose and neat iconography (ASCII art), making it easier to understand. Cypher is inspired by a number of approaches and builds upon established practices for expressive querying.

Cypher is comprised of several clauses. The most basic query consist of a `START` clause followed by a `MATCH` and a `RETURN` clause. For example, the following query returns the last name of the friends of the persons whose first name is "George". "

```
START x=node:person(firstName="George")
MATCH (x)-[:knows]->(y)
RETURN y.lastName
```

The clause START specifies one or more starting points (nodes and relationships) in the graph. The MATCH clause contains the graph pattern of the query. The RETURN clause specifies which nodes, relationships, and properties in the matched data will be returned by the query.

The description of a graph pattern is made up of one or more paths, separated by commas. A path is a sequence of nodes and relationships that always start and end in nodes. A simple path is a expression `(x)->(y)` which defines a path starting from the node `(x)` to node `(y)` with an unconstrained ongoing relationship. In a path expression, nodes are drawn in parentheses whereas relationships are drawn in square brackets by using pairs of dashes and greater-than and less-than signs ($--> $ and $<--$). The $<$ and $>$ signs are optional, and indicate relationship direction. The name of a node/relationship can be prefixed by a colon to declare that the node/relationship should have a certain label/type. An optional relationship is represented by "[?]".

A path pattern can follow multiple graph relationships. These are called *variable length relationships*, and are marked as such using an asterisk ($*$). For example, the query `(a)-[:knows*]->(b)` expresses a path starting on the node `(a)`, following only outgoing relationships `knows`, until it reaches node `(b)`. The minimum and maximum number of steps (relationships) followed by a path can be defined as showed in the following expression: `(a)-[:knows*3..5]->(b)`. In order to access the collection of nodes and relationships of a path, a path identifier can be assigned to a path, e.g. `p = (x)->(y)`. The shortest path of a pattern expression P is obtained by using the expression `shortestPath(`P`)`

Cypher allows the expression of complex queries by using other clauses: WHERE provides a criteria for filtering patterns matching results (it is similar to the HAVING clause in SQL). FOREACH can be used to perform updating actions once per element in a list. UNION merges results from two or more queries. WITH divides a query into multiple, distinct parts. CREATE and CREATE UNIQUE can be used to create nodes and relationships. SET allows to set values of properties (attributes). DELETE allows to remove nodes, relationships and properties.

4.2 Expressive power of graph query languages

The expressive power of a query language determines the set of all queries expressible in that language [12]. Determining the expressive power of a language is relevant to understand its capabilities and complexity.

In order to determine the expressive power of a query language, usually one chooses a well-studied query language and compares both languages in their expressive power. A formal comparison between two languages is a complex process that implies the definition of transformations (for databases, queries and solutions) from one language to the other. In this section, we will compare informally SPARQL, G-SPARQL and Cypher by presenting several types of queries in natural language, and showing how they are expressed (if possible) in each query language.

Figure 4.1 shows an UML Class diagram that describes the structure of the graph database used as sample for the queries described in this section.

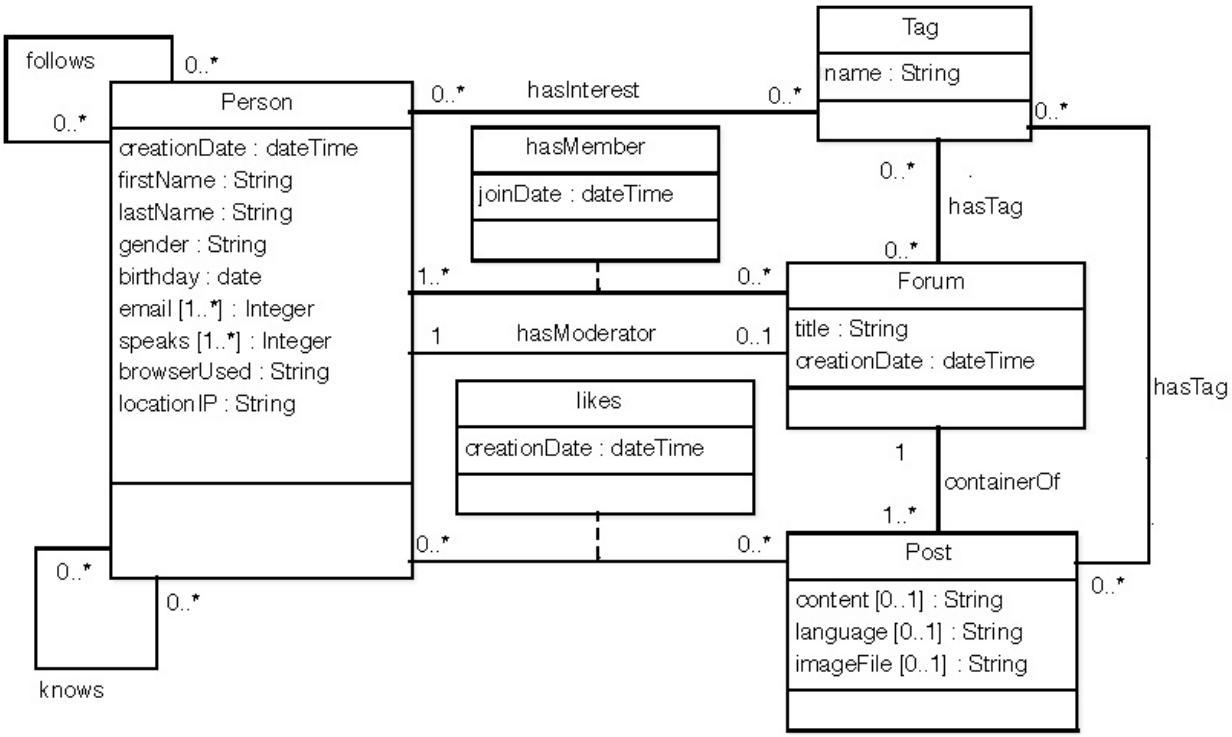

Figure 4.1: Schema of the sample graph database.

Pattern Matching Queries

A pattern matching query is based on the definition of a graph pattern and the objective is to find subgraphs (in the database graph) satisfying the graph pattern. We consider several types of pattern matching queries depending on the complexity of the graph pattern.

– Single node graph patterns This type of query looks for nodes having a given attribute or a condition over an attribute.

Example: return the persons whose attribute first name is “James”.

```
## SPARQL 1.0 and SPARQL 1.1
SELECT ?X
FROM <http://www.socialnetwork.org>
```

```
WHERE { ?X sn:firstName "James" }

## G-SPARQL
SELECT ?X
WHERE { ?X @firstName "James" }

## CYPHER
MATCH (person:Person)
WHERE person.firstName="James"
RETURN person
```

– Single graph patterns A single graph pattern consists of a single structure node-edge-node where variables are allowed in any part of the structure. A single graph pattern is oriented to evaluate adjacency between nodes.

Example: return the pairs of persons related by the "knows" relationship.

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X ?Y
WHERE { ?X sn:knows ?Y }

## G-SPARQL
SELECT ?X ?Y
WHERE { ?X knows ?Y }

## CYPHER
MATCH (person1:Person)-[:knows]->(person2:Person)
RETURN person1, person2
```

– Complex graph patterns

A complex graph pattern is a collection of single graph patterns connected by special operators, usually join, union, difference and negation. In the literature of graph query languages (see for example GraphLog [20]), a complex graph pattern is graphically represented as a graph containing multiple nodes, edges and variables, and special conditions can be defined over all of them (e.g value-based conditions over nodes, negation of edges, summarization, etc.). The evaluation of graph patterns is usually defined in terms of subgraph isomorphism [22, 25].

Example (Join of graph patterns): return the first name of persons having a friend named "Thomas" .

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName ?N .
        ?X sn:knows ?Y . ?Y sn:firstName "Thomas"  }

## G-SPARQL
SELECT ?N
WHERE { ?X @type "Person" . ?X @firstName ?N .
        ?X knows ?Y . ?Y @firstName "Thomas"  }

## CYPHER
MATCH (person:Person)-[:knows]->(thomas:Person)
WHERE thomas.firstName="Thomas"
RETURN person.firstName
```

Example (Union of graph patterns): return the persons interested in either "Queen" or "U2".

```
## SPARQL 1.0 and SPARQL 1.1
## This query introduces duplicates which are eliminated by the DISTINCT operator
PREFIX sn: <http://www.socialnetwork.org/>
SELECT DISTINCT ?X
WHERE { ?X sn:type sn:Person . ?X sn:hasInterest ?T . ?T sn:type sn:Tag .
        { { ?T sn:name "Queen"} UNION { ?T sn:name "U2" } } }

## SPARQL 1.0 and SPARQL 1.1
## This query avoids duplicates by using a FILTER condition
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X
WHERE { ?X sn:type sn:Person . ?X sn:hasInterest ?T . ?T sn:type sn:Tag .
        ?T sn:name ?N . FILTER ( ?N = "Queen" || ?N = "U2" ) }

## G-SPARQL
SELECT ?X
WHERE { ?X @type "Person" . ?X hasInterest ?T . ?T @type "Tag" .
        ?T @name ?N . FILTER ( ?N = "Queen" || ?N = "U2" ) }

## CYPHER
MATCH (person:Person)-[:hasInterest]->(tag:Tag)
WHERE tag.name="Queen" OR tag.name="U2"
RETURN DISTINCT person
```

Example (Difference/negation of graph patterns): return the first name of persons which do not like any post.

```
## SPARQL 1.0
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE {
 ?X sn:type sn:Person .
 { ?X sn:firstName ?N . OPTIONAL { ?X sn:likes ?P } } FILTER (!bound(?P))  }

## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { { ?X sn:type sn:Person . ?X sn:firstName ?N } MINUS { ?X sn:likes ?P } }

## G-SPARQL
SELECT ?N
WHERE {
 ?X @type "Person" .
 { ?X @firstName ?N . OPTIONAL { ?X likes ?P } } FILTER (!bound(?P))  }

## CYPHER
MATCH (person:Person)
WHERE NOT(person-[:likes]->(:Post))
RETURN person.firstName
```

Filter conditions in graph patterns

A graph pattern can be extended to allow filter boolean restrictions over node and edge labels.
Example: find the persons whose age is between 18 and 30.

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X
WHERE { ?X sn:type sn:Person . ?X sn:age ?A . FILTER (?A > 18 && ?A < 30) }

## G-SPARQL
SELECT ?X
WHERE { ?X @type "Person" . ?X @age ?A . FILTER (?A > 18 && ?A < 30) }

## CYPHER
MATCH (person:Person)
WHERE person.age>18 and person.age<30
RETURN person
```

Fixed-length path queries

A *fixed-length path query* is a special type of graph pattern which represents a traversal from a source node to a target node, by including a fixed number of nodes and edges.
Example: Find the names of people at distance 2 from "James" by following "knows" links.

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName "James" .
        ?X sn:knows ?Z . ?Z sn:knows ?Y . ?Y sn:firstName ?N .
        FILTER (!(?Y = ?X || ?Y = ?Z)) }

## G-SPARQL
SELECT ?N
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X knows ?Z . ?Z knows ?Y . ?Y @firstName ?N .
        FILTER (!(?Y = ?X || ?Y = ?Z)) }

## CYPHER
MATCH (james:Person)-[:knows]->(dist1person:Person)-[:knows]->(dist2person:Person)
WHERE james.name="James" AND NOT(dist2person=james) AND NOT(dist2person=dist1person)
RETURN DISTINCT dist2person
```

4.2.1 Reachability queries

Reachability queries are characterized by path or traversal problems, and the objective is to test whether two given nodes are connected by a path. Reachability queries are usually expressed by using regular path queries.

Regular path queries

A regular path query is usually represented as a single graph pattern (N_s E N_t) where N_s is the source node (value or variable), N_t is the target node (value or variable), and E is a regular expression representing the path pattern (see property paths in the description of SPARQL 1.1). A reachability query may involve the generation

of a boolean result, a set of nodes, a single possible solution path, or a set of possible paths. Regular path queries are not supported in SPARQL 1.0.

Example: find the first name of people that can be reached from "James" by relation "knows".

```
## SPARQL 1.1
 PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName "James" .
        ?X sn:knows* ?Y .
        ?Y sn:firstName ?N }

## G-SPARQL
SELECT ?N
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X knows* ?Y .
        ?Y @firstName ?N }

## CYPHER
MATCH (james:Person)-[:knows*]->(reachablePerson:Person)
WHERE james.firstName="James"
RETURN DISTINCT reachablePerson
```

Consider reachability queries where the computed paths are required to be returned as a sequence of nodes and edges. Only G-SPARQL and Cypher are able to answer this type of queries. SPARQL 1.1 is restricted to return the source node and the target node of the path.

Example: find the people that can be reached from "James" by following the relation "knows", and return the corresponding path.

```
## G-SPARQL
SELECT ?N ??P
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X ?PP(knows*) ?Y .
        ?Y @firstName ?N }

## CYPHER
MATCH path = (james:Person)-[:knows*]->(reachablePerson:Person)
WHERE james.firstName="James"
RETURN DISTINCT reachablePerson, path
```

Regular path queries with path-length restrictions

Some languages allow to restrict the length of the paths returned by a regular path query.

G-SPARQL allows expressions of the form `Length(??P, <3)` which allows to filter the path variable `??P` with the length condition `<3`. In Cypher, path-length restrictions can be defined in the recursive relation, for example `[:KNOWS*1..3]`. SPARQL 1.1 is not able to express path-length restrictions.

Example: return the paths of length 5-10, along the "knows" relation, that connect 'James' and 'Axel'.

```
## G-SPARQL
SELECT ??P
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X ??P(knows*) ?Y . ?Y @firstName "Axel" .
        FILTERPATH(Length(??P, >=5)) . FILTERPATH(Length(??P, <=10)) }
```

```
## CYPHER
MATCH path = (james:Person)-[:KNOWS*5..10]->(axel:Person)
WHERE james.firstName="James" AND axel.firstName="Axel"
RETURN path
```

Regular path queries with value-based restrictions

This type of queries implies the introduction of value-based restriction over the paths returned by a regular path query, i.e. value conditions over the attributes of nodes and edges belonging to the resulting paths.

SPARQL 1.1 does not support this kind of restrictions. G-SPARQL is characterized by allowing valued-based restrictions over specific nodes and edges (i.e. AtLeastNode, AtMostNode, AllNodes, AtLeastEdge, AtMostEdge, AllEdges). In the case of Cypher, this type of restrictions can be defined in the `WHERE` clause.

Example: find the first name of people that can be reached from "James" by following relations "knows" created during the year 2012 (assume that each relation "knows" contains an attribute "year" of creation).

```
## G-SPARQL
SELECT ?N
WHERE { ?X sn:firstName "James" .
        ?X ??P(sn:knows+) ?Y .
        ?Y sn:firstName ?N .
        FILTERPATH(AllEdges(??P, @year "2012")) }

## CYPHER
MATCH (james:Person)-[r:KNOWS]->(other:Person)
WHERE james.firstName="James" AND r.year=2012
RETURN other.firstName
```

Regular path queries with structural restrictions

This type of queries implies the introduction of structural restrictions over the paths returned by a regular path query, i.e. conditions over the edges of the nodes in the resulting paths.

Only Cypher is able to provide this kind of restrictions. They can be defined in the clause `WHERE` by using the collection function `nodes(path)` which returns the nodes occurring in the resulting paths associate to the given `path`.

Example: find the first name of people that can be reached from "James" by following relations "knows" and satisfying that each people in the sequence also knows "James").

```
## CYPHER
MATCH path = (james:Person)-[:KNOWS*]->(other:Person)
WHERE filter(node IN nodes(path) WHERE (node:Person)-[:KNOWS]->(james:Person))
RETURN other.firstName
```

Shortest path queries

A shortest path query means to compute the quickest/shortest route between two nodes in the graph. Most languages provide ad-hoc functions to calculate shortest paths queries. In some cases the shortest path can be calculated by combining a reachability query with aggregate operators (e.g. COUNT + MIN), although it requires that the reachability query results in a set of paths.

SPARQL 1.1. is not able to calculate shortest path queries. G-SPARQL and Cypher provide special predicates to return the shortest path.

Example: Return the shortest path between "James" and "Axel" by following the relation "knows".

```
## G-SPARQL
SELECT ?*P
WHERE { ?X @firstName "James" .
        ?X ?*P(knows+) ?Y .
        ?Y @firstName "Axel"}

## CYPHER
MATCH path = shortestPath((james:Person)-[:knows]-(axel:Person))
WHERE james.firstName="James" AND axel.firstName="Axel"
RETURN path
```

4.2.2 Aggregate queries and grouping

Aggregate queries are based on special operators, non related to the data model, that permit to summarize or operate on the query results. Common aggregate operators include: COUNT, SUM, AVG, MIN, and MAX. Aggregate operators are very useful to calculate special information about nodes and edges in the graph, for example the degree of a node (i.e. by counting the neighbors of the node) or the length of the shortest path between two nodes.

SPARQL 1.0 and G-SPARQL do not support aggregate operators. SPARQL 1.1 and Cypher defines all the common aggregate operators. Additionally, Cypher includes special aggregate operators for paths, for example `length(path)` allows to obtain the length of the given `path`.

Example: Return the number of friends of "James"

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT (COUNT(?Y) AS ?Friends)
WHERE { ?X sn:firstName "James" . ?X sn:knows ?Y }

## CYPHER
MATCH (james:Person)-[:FRIENDS]->(friend:Person)
WHERE james.firstName='James'
RETURN count(friend)
```

Example: Return the length of the shortest path between "James" and "Axel".

```
## CYPHER
MATCH path = shortestPath((james:Person)-[:knows]-(axel:Person))
WHERE james.firstName="James" AND axel.firstName="Axel"
RETURN length(path)
```

Grouping.

The result sequence of an aggregate query can be grouped by values of different attributes or relationships. This is the function of the GROUP BY operator in SQL.

Operators for grouping are defined in SPARQL 1.1 and Cypher. G-SPARQL does not support grouping.

Example: Return the number of friends of each person

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X, (COUNT(?Y) AS ?Friends)
WHERE { ?X sn:type sn:Person . ?X sn:knows ?Y }
GROUP BY X?
```

```
## CYPHER
MATCH (person:Person)-[:FRIENDS]->(friend:Person)
RETURN person, count(friend)
```

Group conditions.

Some query languages allows to filter the groups according to a given condition. This is the function of the HAVING operator in SQL.

Example: for each person having 100 friends or more, returns their first name and friends' number.

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N, (COUNT(?F) AS ?FriendsNumber)
WHERE { ?X sn:type sn:Person . ?X sn:firstName ?N . ?X sn:knows ?F }
GROUP BY ?X
HAVING (COUNT(?F) >= 100)

## CYPHER
MATCH (person:Person)-[:knows]->(friend:Person)
WITH person, count(friend) AS friendsNumber
WHERE friendsNumber>=100
RETURN person.firstName, friendsNumber
```

4.2.3 Restrictions over result sequences

The result sequence of a query can be restricted by using the following operators:

- DISTINCT: to return only distinct (different) values.
- ORDER BY: to sort (ascending or descending) the result sequence by a node, edge or property.
- OFFSET: to define where the solutions start from in the sequence of solutions (i.e. a given position in the sequence).
- LIMIT: to restrict the number of solutions to a given number.

Most of the above operators are supported by SPARQL 1.1 and Cypher.
Example: Return the youngest top-5 distinct friends of "James".

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT DISTINCT ?F ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName "James" .
        ?X sn:knows ?F . ?F sn:birthday ?B }
ORDER BY ASC(?B)
LIMIT 5

## CYPHER
MATCH (james:Person)-[:knows]->(friend:Person)
WHERE james.name="James
RETURN friend
ORDER BY friend.birthday ASC
LIMIT 5
```

4.2.4 Comments about the expressive power of SPARQL, G-SPARQL and Cypher

The main feature of a query language is the computation of graph pattern matching queries. SPARQL is able to express complex graph patterns by means of a collection of triple patterns whose solutions can be combined and restricted by using several operators (i.e. AND, UNION, OPTIONAL, and FILTER). Similarly, Cypher expresses graph patterns in its MATCH clause, which allows for the expression of arbitrarily sized graph patterns (not restricted to triples as in SPARQL). In conjunction with its built-in functions (aggregation, collection, scalar, etc.) and predicates in its WHERE clause, these Cypher subgraphs can be constrained, combined, and transformed.

It has been shown [14] that SPARQL 1.0 has the same expressive power of relational algebra under set semantics[1], i.e. when duplicates are not considered in the computation of queries. Considering that G-SPARQL is an extension of SPARQL 1.0, we can say that G-SPARQL is at least as expressive as SPARQL 1.0 and relational algebra (and probably more, considering its support for path queries). From the introductory description of Cypher, and the examples presented above, we can suppose that Cypher is able to express the same queries as SPARQL 1.0 and relational algebra (however a formal proof is necessary).

It has been shown that SQL is more expressive than relational algebra for two main reasons [33]: SQL operates under bag semantics (i.e. it considers duplicates); and SQL supports aggregate functions and grouping. Like SPARQL, Cypher also supports numerous aggregate and grouping functions. Moreover, SPARQL and Cypher returns multisets (bags), and via its `DISTINCT` keyword also supports sets.

In spite of its useful facilities, the initial versions of SQL are not able to compute reachability and other recursive queries [13]. It was until 1999, where the SQL3 standard introduced recursive queries and with this the support for computing the transitive closure in a graph. As described in the above section, transitive closure queries are usually expressed in graph query languages by using simple regular path queries. SPARQL 1.0 does not support simple regular path queries, but this feature was included in SPARQL 1.1 with the name of property paths. Property paths are very useful to express reachability queries with structural conditions, however they present a drawback: variables are allowed at the ends of the path but not in the path itself. G-SPARQL [42] extends the expressive power of SPARQL 1.1 by allowing regular path queries with value-based restrictions on the nodes and edges in the path.

As with G-SPARQL, Cypher has rich support for the expression of reachability queries, including regular path queries restricted by attribute value, relationship types, node types, (minimum and maximum) path length, and more. In addition, Cypher includes a library of built-in functions for working with the returned paths, including but not limited to: filtering paths based on predicates (e.g. length or content), and extracting attributes from elements within a path. Finally, the `RETURN` (projection) clause in Cypher allows for entire matching paths to be returned. Hence, Cypher presents more expressive power than G-SPARQL.

It has been shown [33] that the addition of aggregate functions to relational algebra and SQL strictly increases its expressive power (e.g. aggregates allow to compare cardinalities of relations). This feature increases the expressive power of SPARQL 1.1 and Cypher.

[1] It is also well know that Relational Algebra has the same expressive power as Relational Calculus (First Order logic without functions) and non-recursive safe Datalog with negation [12]

5 Expressing Queries in Benchmarks

When describing queries for benchmarks it is important that those descriptions are agnostic of data model, vendor, implementation, etc. To achieve this requires that query descriptions are at a sufficiently high level of abstraction. Given that all LDBC benchmarks are developed within the context of a use case (i.e., a concrete application, in a specific domain), queries will be described at the level of the application domain. The specific method for describing queries is covered in the following section.

5.1 Format for query specification

In this section we present a proposal of format for specification of graph queries to be used in the design of a benchmark. To keep query descriptions as data model agnostic as possible, they will be described from the abstraction level of the application domain.

Query description structure

A query description will take the following structure:

a) Query name: Summary textual description of the query.

b) Description: Detailed description of the query in textual format (plain English).

c) Parameters: List of input parameters.

d) Result: Description explaining expected content and format of the query result (see the pre-defined format presented below).

e) Functionality: Textual functional description of the query, from the abstraction level of the database (not the application domain).

f) Relevance: it is a textual description (plain English) of the reasoning for including this query in the workload. It should include a discussion about the technical challenges (Choke Points) targeted by the query.

Result description syntax

The result of a query should be described by using the following format for representing entities, relationships, attributes, sets and collections (ordered and unordered):

- Entity
  - Rule: One word (or multiple words appended together). Uppercase first character. Each appended word has uppercase first character.
  - Example: `FamousPerson`
  - Example description: Entity of type “FamousPerson”
- Relationship
  - Rule: One word (or multiple words appended together). Lower case first character. Each appended word has uppercase first character. Surrounded by "arrow" to communicate direction.
  - Example: `-worksAt->`
  - Example description: Directed relationship of type “worksAt”.
- Attribute

  - Rule: One word (or multiple words appended together). Lower case first character. Each appended word has uppercase first character. May apply to either Entity or Relationship. Dereferenced by a "." prefix.
  - Example: `.firstName`
  - Example description: Attribute with name "firstName".

- Unordered/Ordered Sets
  - Rule: An unordered set is represented by "{ }", and when ordered by "{{ }}". A set does not contain duplicates.
  - Example: `{FamousPerson.firstName}`
  - Example description: Unordered set of all unique "firstName" values belonging to "FamousPerson" entities.

- Unordered/Ordered Collections
  - Rule: An unordered collection/bag is represented by "( )", and when ordered by "(( ))". A collection may have duplicates.
  - Example: `(FamousPerson.firstName)`
  - Example description: List of all "firstName" values belonging to "FamousPerson" entities.

Query description example

```
a) Query name: Simple person search
b) Description:
   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.
c) Parameters:
   - firstName: the first name of a person (e.g, James).
d) Result (for each result return)
   - Person.firstName
   - Person.lastName
   - {Person.email}
   - {Person.language}
   - Person-isLocatedIn->Location.name
   - (Person-studyAt->University.name,
      Person-studyAt->.classYear,
      Person-studyAt->University-isLocatedIn->Country.name)
e) Functionality
   - Simple lookup query
f) Relevance
   - Choke Points: Complex aggregate performance (e.g. concatenation)
   - The optimizer is expected not to get stuck in comparing different permutations
     of single valued attributes, all will be fetched, order does not matter.
   - The optimizer is 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.
   - The literals should be translated from internal representation after the top k
     order by.
   - Interesting mostly for throughput.
```

6 Conclusions

Graph data is growing much more faster than the current technology is able to manage. With huge storage systems and fast communication infrastructures, social networks or scientific applications are creating very large graphs with an enormous number of interrelated entities. In this scenario, the classical approach of single-computer in-memory solutions is not enough, and we need more parallel power to run complex graph algorithms on top of partitioned graphs. Also, while performance has been the main focus of research in the last years, usability and query specification are still an open issue.

Benchmarking graph databases must be aware of this situation. First, the multiple environments or platforms used for graph storage and querying should be covered by the benchmarks. For example, it is not the same a single SMP computer with many shared-memory cores that the same number of CPUs in a distributed architecture, where the latencies in communication can be more important than the locality problems due to memory and cache access. Software solutions are also quite different, from the synchronous processing based in BSP or the strict sequential computation in MapReduce, to new asynchronous proposals in the vertex-centric model.

Thus, from the existing technologies and computation models we can infer different choke points to be benchmarked. Also, the existing knowledge coming from well known benchmarks such as TPC-H can be used to identify what problems are in common with other database technologies such as the relational model, and which ones need to be adapted to the new graph-shaped data. Again, it is not easy because there are hardware factors such as latencies or data placement highly dependent from software factors such as graph partitioning or random accesses in path traversal queries, which are the basis of most graph analysis algorithms.

Finally, even with a good definition of the graph query choke points and the different workloads, it is very difficult to express queries in a portable form for any graph query environment. Graph query languages are still in development, and in most cases are strictly dependent of a particular solution or platform, if not too much theoretical to be used in practice. In a first step, a graph benchmark specification probably will rely only in textual definitions of the query behavior, parameters and results, leaving the responsibility to graph database implementators and researchers of the porting of the queries to a particular platform.

REFERENCES

[1] Apache Giraph. http://giraph.apache.org/,https://github.com/apache/giraph.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache Hama. http://hama.apache.org/.

[4] DBpedia. http://dbpedia.org/.

[5] DEX High-performance Graph Database. http://www.sparsity-technologies.com/dex.

[6] Gremlin Graph Query Language. https://github.com/tinkerpop/gremlin/wiki.

[7] InfiniteGraph Distributed Graph Database. http://www.objectivity.com/infinitegraph.

[8] Neo4j Cypher Documentation. http://docs.neo4j.org/refcard/2.0/.

[9] Vertica. http://http://www.vertica.com.

[10] Virtuoso Universal Server. http://virtuoso.openlinksw.com/.

[11] YarcData Urika. http://www.yarcdata.com/.

[12] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[13] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 110–119. ACM Press, 1979.

[14] Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, number 5318 in LNCS, pages 114–129, 2008.

[15] David A Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 547–556. IEEE, 2005.

[16] 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.

[17] PA Boncz. *Monet: a next-generation database kernel for query-intensive applications*. PhD thesis, Universiteit van Amsterdam, 2002.

[18] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[19] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1123–1126. ACM, 2010.

[20] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a Visual Formalism for Real Life Recursion. In *Proceedings of the 9th Symposium on Principles of Database Systems (PODS)*, pages 404–416. ACM Press, April 1990.

[21] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[22] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A Graphical Query Language Supporting Recursion. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data*, pages 323–330. ACM Press, May 1987.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[25] Claudio Gutierrez, Carlos Hurtado, and Alberto O.Mendelzon. Foundations of Semantic Web Databases. In *Proceedings of the 23rd Symposium on Principles of Database Systems (PODS)*, pages 95–106, June 2004.

[26] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[27] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[28] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[29] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[30] Petr Konecny. Introducing the cray xmt. In *Proc. Cray User Group meeting (CUG 2007)*, 2007.

[31] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[32] Ralf Lämmel. Google's mapreduce programming model - revisited. *Science of computer programming*, 70(1):1–30, 2008.

[33] Leonid Libkin. Expressive power of sql. *Theoretical Computer Science*, 296(3):379–404, 2003.

[34] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[35] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[36] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[38] Norbert Martínez-Bazan, M Ángel Águila-Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, pages 110–119. ACM, 2012.

[39] Jacob Nelson, Brandon Myers, Andrew H Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, Dan Grossman, Simon Kahan, and Mark Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, pages 10–10. USENIX Association, 2011.

[40] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/2008/REC-115-sparql-query-20080115/, January 15 2008.

[41] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the sap hana database. In *BTW*, pages 403–420, 2013.

[42] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, 2012.

[43] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. 2013.

[44] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. Horton: Online query execution engine for large distributed graphs. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1289–1292. IEEE, 2012.

[45] Bin Shao, Haixun Wang, and Yatao Li. The trinity graph engine. *Microsoft Research*, 2012.

[46] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[47] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: Graph algorithms for the (semantic) web. In *The Semantic Web–ISWC 2010*, pages 764–780. Springer, 2010.

[48] Dan Suciu. Big data begets big database theory. In *Big Data*, pages 1–5. Springer, 2013.

[49] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[50] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, Ion Stoica, and EECS AMPLab. Graphx: A resilient distributed graph system on spark. 2013.

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[52] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 265–276. VLDB Endowment, 2013.

[53] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A vectorized analytical dbms. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1349–1350. IEEE, 2012.