Large Scale Querying and Processing for Property Graphs PhD Symposium*

Mohamed Ragab
Data Systems Group, University of Tartu
Tartu, Estonia
mohamed.ragab@ut.ee

ABSTRACT

Recently, large scale graph data management, querying and processing have experienced a renaissance in several timely application domains (e.g., social networks, bibliographical networks and knowledge graphs). However, these applications still introduce new challenges with large-scale graph processing. Therefore, recently, we have witnessed a remarkable growth in the prevalence of work on graph processing in both academia and industry. Querying and processing large graphs is an interesting and challenging task. Recently, several centralized/distributed large-scale graph processing frameworks have been developed. However, they mainly focus on batch graph analytics. On the other hand, the state-of-the-art graph databases can't sustain for distributed efficient querying for large graphs with complex queries. In particular, online large scale graph querying engines are still limited. In this paper, we present a research plan shipped with the state-of-the-art techniques for large-scale property graph querying and processing. We present our goals and initial results for querying and processing large property graphs based on the emerging and promising `Apache Spark` framework, a defacto standard platform for big data processing. In principle, the design of this research plan is revolving around two main goals. The first goal focuses on designing an adequate and efficient graph-based storage backend that can be integrated with the Apache Spark framework. The second goal focuses on developing various *Graph-aware* optimization techniques (e.g., graph indexing, graph materialized views), and extending the default relational Spark `Catalyst` optimizer with *Graph-aware* cost-based optimizations. Achieving these contributions can significantly enhance the performance of executing graph queries on top of `Apache Spark`.

1 INTRODUCTION

Graphs are everywhere. They are intuitive and rich data models that can represent strong connectivity within the data. Due to their rich expressivity, graphs are widely used in several application domains including the Internet of Things (IoT), social networks, knowledge graphs, transportation networks, Semantic Web, and Linked Open Data (LOD) among many others [17]. In principle, graph processing is not a new problem. However, recently, it gained an increasing attention and momentum, more than before, in several timely applications [22]. This is due to the ongoing huge explosion in graph data alongside with a great availability of computational power to process this data. Nowadays, several enterprises have or planned to use graph technologies for their data storage and processing applications. Moreover, Graph databases are currently widely used in the industry to manage

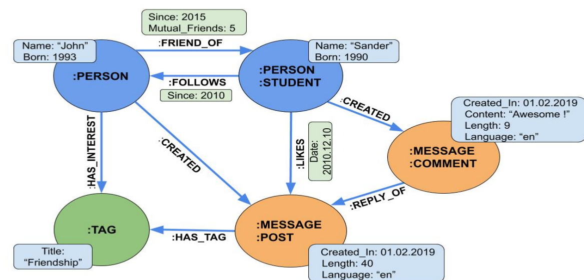

Figure 1: A simple example of a Property Graph

graph data following the core principles of relational database systems [10]. Popular Graph databases include `Neo4j`[1], `Titan`[2], `ArangoDB`[3] and `HyperGraphDB`[4] among many others.

In general, graphs can be represented in different data models [1]. In practice, the two most commonly-used graph data models are: Edge-Directed/Labelled graph (e.g. Resource Description Framework (RDF[5])) for representing data in triples (*Subject*, *Predicate*, and *Object*), and the *Property Graph* (PG) data model [9]. The PG model extends edge-directed/labelled graphs by adding (multiple) *labels* for the nodes and *types* for the edges, as well as adding (multiple) key-value *proprieties* for both nodes and edges of the graph. In this paper, we focus on the PG model, as it is currently the most widely used and supported graph data model in industry as well as in the academia. In particular, most of the current top and widely used graph databases use the property graph data model [8]. This great success and wide spread of PG model is due to its great balance between conceptual and intuitive simplicity, in addition to its rich expressiveness [20]. Figure 1 illustrates an example of a simple property graph.

Recently, several graph query languages have been proposed to support querying different kinds of graph data models [1]. For example, the *W3C* community has developed and standardized `SPARQL`, a query language for querying the RDF-typed graphs [16]. `Gremlin` [19] has been proposed as a functional programming graph query language that supports the property graph model, and optimized for supporting graph navigational/traversal queries. `Oracle` has designed `PGQL` [21], an SQL-like graph query language which also supports querying the property graph data model. Facebook also presented `GraphQl` [13], a *REST-API* like graph query language for accessing the web data as a graph of objects. `Neo4j` designed `Cypher` [9] as its main query language which targets querying the property graph data model in a natural and intuitive way. In practice, `Cypher` is currently the

*The supervisor of this work is Sherif Sakr

[1] https://neo4j.com/
[2] https://github.com/thinkaurelius/titan
[3] https://www.arangodb.com/
[4] http://www.hypergraphdb.org/
[5] https://www.w3.org/RDF/

© Copyright 2020 for this paper held by its author(s). Published in the proceedings of DOLAP 2020 (March 30, 2020, Copenhagen, Denmark, co-located with EDBT/ICDT 2020) on CEUR-WS.org. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2572/short3.pdf

most popular graph query language, and has been supported by several other graph-based projects and graph databases including SAP HANA[6], RedisGraph[7], Agens Graph[8], MemGraph[9], and Morpheus[10] (Cypher for Apache Spark) [7].

Problem Statement: With the continuous increase in graph data, processing large graphs introduces several challenges and detrimental issues to the performance of graph-based applications [12]. In particular, one of the common challenges of large-scale graph processing is the efficient evaluation of graph queries. In particular, the evaluation of a graph query mainly depends on the graph scope (i.e. the number of nodes and edges it touches) [20]. Therefore, real-world complex graph queries may unexpectedly take a long time to be answered [18]. In practice, most of current graph databases architecture are typically designed to work on a single-machine (non-clustered). Therefore, graph querying solutions can only handle the Online Transactional Processing (OLTP-style) query workload, which defines relatively simple computational retrievals on a limited subset of the graph data. For instance, *Neo4j* is optimized for subgraph traversals and for medium-sized OLTP query workloads. Whereas, for complex Online Analytical Processing (OLAP-style) query workload (where the query needs to touch huge parts of the graph, and complex joins and aggregations are required), graph databases are not the best solution.

In this paper, we provide an overview of the current state-of-the-art efforts in solving the large scale graph querying along side with their limitations (Section 2). We present our planned contributions based on one of the emerging distributed processing platforms for querying large graph data, Morpheus[11] (Section 3). We present our initial results in Section 4, before we conclude the paper in Section 5.

2 STATE OF THE ART

Distributed processing frameworks can be utilized to solve the graph scalability issues with query evaluation. Apache Spark represents the defacto standard for *distributed* big data processing [2]. Unlike MapReduce model, Spark uses the main memory for parallel computations over large datasets. Thus, it can be up to 100 times faster than Hadoop [24]. Spark maintains this level of efficiency due to its core data abstraction which is known as *Resilient Distributed Datasets*(RDDs). An *RDD* is an immutable, distributed and fault tolerant collection of data elements which can be partitioned across the memory of nodes in the cluster. Another efficient data abstraction of Spark is the Spark *DataFrames*. DataFrames are organized according to a specific schema into named and data-typed columns like a table in the relational databases. Spark proposes various higher level libraries on top of *RDDs* and *DataFrames* abstractions, GraphX [11] and SparkSQL [3] for processing structured and semi-structured large data.

Spark-SQL is a high-level library for processing structured as well as semi-structured large datasets. It enables querying these datasets stored in DataFrames abstraction using SQL. Spark-SQL acts as a distributed SQL query engine over large structured datasets. In addition, SparkSQL offers a *Catalyst* optimizer in its core for improving query executions [4]. *Catalyst* optimizer is mainly a rule-based optimizer, adding optimization rules based on functional programming constructs in *Scala* language. It can also apply variety of relational cost-based optimizations for improving the quality of multiple alternative query execution plans.

GraphX extends the low level RDD abstraction, and introduces a new abstraction called *Resilient Distributed Graphs* (*RDG*). In a graph, *RDG* relates records with vertices and edges and produces an expressive computational primitives' collection. GraphX chains the benefits of graph-parallel and data-parallel systems. However, GraphX is not currently actively maintained. Besides, GraphX is based on the low level *RDGs*. Thus, it cannot exploit the Spark 2's *Catalyst* query optimizer that supports only Spark DataFrames API. Moreover, GraphX is only available to Scala users.

GraphFrames is a graph package built on DataFrames [5]. GraphFrames benefits from the scalability and high performance of DataFrames. They provide a uniform API for graph processing available from *Scala*, *Java*, and *Python*. GraphFrames API implements DataFrame-based graph algorithms, and also incorporates simple graph pattern matching with fixed length patterns (called *'motifs'*). Although GraphFrames are based on Spark DataFrames API, they have a semantically-weak graph data model (i.e. based on un-typed edges and vertices). Moreover, The *motif* pattern matching facility is very limited in comparison to other well-established graph query languages like Cypher. Besides, other important features which are present in Spark GraphX such as partitioning are missing in the GraphFrames package.

In practice, by default, Spark does not support processing and querying of property graph data model, despite is widespread use. To this extent, the Morpheus project has come to the scene. In particular, the Morpheus project has been designed to enable the evaluation of Cypher over large property graphs using DataFrames on top of Apache Spark framework. In practice, this framework enables combining the scalability of the Spark framework with the features and capabilities of Neo4j by enabling the Cypher language to be integrated into the Spark analytics pipeline. Interestingly, graph processing and querying can be then easily interwoven with other Spark processing analytics libraries such as *Spark GraphX*, *Spark ML* or *Spark-SQL*. Moreover, this enables easy merging of graphs from Morpheus into Neo4j. Besides more advanced capabilities of Morpheus such as the ability to handle multiple graphs (i.e. graph *Composability*) from different data sources even if they are not graph sources (i.e relational data sources), it has the ability to create *graph views* on the data as well.

Figure 2 illustrates the architecture of Morpheus framework. In Morpheus, Cypher queries are translated into Abstract Syntax Tree (*AST*). Then, Morpheus core system translates this AST into DataFrame operations with schema and Data-Type handling. It is worth noting that DataFrames in Spark use schema, while Neo4j or generally property graphs optionally use a schema (i.e. schema free data model). Therefore, Morpheus provides a Graph Data Definition Language *GDDL* for handling schema mapping. Particularly, *GDDL* expresses property graph *types* and maps between those *types* and the relational data sources. Moreover, the Morpheus core system manages importing graph data that can reside in different Spark storage backends such as HDFS (i.e. in different file formats), Hive, relational databases using *JDBC*, or Neo4j (i.e. Morpheus Property graph data sources *PGDs*), and exporting these property graphs directly back to those Spark storage backends. This interestingly means that graph data can

[6]https://s3.amazonaws.com/artifacts.opencypher.org/website/ocim1/slides/Graph+Pattern+Matching+in+SAP+HANA.pdf
[7]https://oss.redislabs.com/redisgraph/
[8]https://bitnine.net/agensgraph/
[9]https://memgraph.com/
[10]https://github.com/opencypher/morpheus
[11]https://github.com/opencypher/morpheus

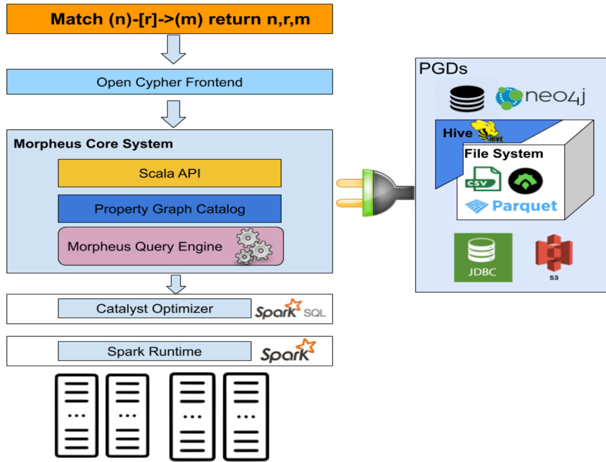

Figure 2: Morpheus Architecture

be read from these native Spark data sources without altering nor copying the original data sources to Morpheus. Particularly, it is like you plug-in these storage backends to the Morpheus framework as shown in Figure 2. The native Spark *Catalyst* optimizer is used in Morpheus pipeline for making various core query optimizations for the generated relational plan of operations. Last but not least, Morpheus runs these optimized query plans on the Spark cluster using distributed Spark runtime environment.

3 RESEARCH PLAN

In general, `Morpheus` has been designed to enable executing Cypher queries on top of Spark. However, on the backend, the property graphs are represented and maintained using Spark relational DataFrames. Therefore, Cypher graph-based queries are internally translated into relational operations over these DataFrames. Therefore, Spark still performs operations on the property graph as tabular data and views with specified schema. Thus, adding a *graph-aware* optimization layer for Spark can significantly enhance the performance of graph query execution on the property graph data. In this research plan, we are focusing on two main aspects for enhancing querying and processing large property graphs in the context of the `Morpheus` project. The first aspect is to design an efficient Spark-based storage backend for persisting property graphs. The other aspect is to provide graph-aware optimizations for query processing inside Morpheus such as Graph indexing, Graph Materialized Views, and last but not least graph cost-based optimizations on top of the default Spark Catalyst optimizer. In order to achieve these aspects in our research plan, we focus on answering the following *Research Questions (RQs)*:

RQ1: Graph Persistence(Which storage backend achieves better performance ?): Large graphs require well-suited persistence solutions that are efficient for query evaluations and processing [20]. As mentioned earlier, graph data in Morpheus can settle on multiple different data sources such as HDFS with its different file formats (e.g. Avro, Parquet, and CSV), Neo4j, Hive, or other kinds of relational DBs (Figure 3). Therefore, first of all, we need to investigate which Spark storage backend for the large property graph data is the best performing one in the context of Morpheus. Deciding on the best performing storage backend with large property graphs plays a major role in enhancing the performance of Morpheus, and further gives us useful

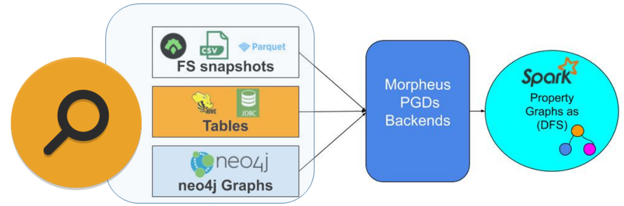

Figure 3: Comparison of Spark PG storage backends

insights for the subsequent optimizations (i.e. graph *Indexing* and *Partitioning*). Further, figuring out the top performing backends helps optimizing the query plan afterwards. For instance, if the best performing storage backend is a *Columnar-Oriented* backend (e.g. *ORC*), it is better for making more *pushing projections down* in the query plan. Whereas, if it is a *Row-Oriented* backend, it is better to make more *pushing selections down* in the plan.

RQ2: Graph Indexes (How can we use Graph Indexing for Better Performance?): The default method for processing graph queries is to perform a *subgraph matching* search against the graph dataset [14]. Several graph indexing techniques have been proposed in the literature. In practice, building a graph index is a multi-faceted process. That is, it depends on using the graph structural information for enumerating and extracting the most frequent features (i.e. graph sub-structures), and then building a data structure of these features. These data structures are such as *Hash Tables* , *Lattices*, *Tries* or *Trees* [14]. The indexed features can be in the form of simple graph patterns/paths, trees, graphs, or a mix of graphs and trees. Further, selecting these features to be indexed can be done exhaustively via enumerating all such features across the whole graph data set [15], or via mining the graph data set for frequent patterns or features (i.e. *Discriminative Features*) [23, 26]. This mining is recommended in building a graph index as the size of the created graph index should be reasonable. It is also worth noting that, most of the existing graph indexing algorithms are only able to handle undirected graphs with labelled vertices [14].

Currently, Morpheus doesn't use any indexing mechanism for property graphs while executing graph queries. To this end, we aim to build an efficient indexing scheme of the property graphs in an offline mode (taking into consideration its schema, as well as its storage backend). Then, this index will be used for reducing the search space for the complex graph pattern matching task. Thus, consulting this index for executing the graph query workload is better than exhaustive *vertex-to-vertex* correspondence checking from the query to the graph which involves a lot of expensive join operations in the relational representation. In our plan, we don't consider the overhead of updating the built index, as Morpheus is currently supporting only read operations, and thus no insertions and deletions happen to the already generated property graph.

RQ3: Graph Materialized Views (How can we use Graph Views for better performance?): Morpheus and most of graph databases tend to compute each query from scratch without being aware of the previous query workloads [25]. Particularly, if we repeatedly execute the same query using Morpheus, the execution plan always stays the same and yields no changes in the execution plan nor in time improvement. Moreover, Spark-SQL registered DataFrames are (by default) non-materialized views [3]. Spark-SQL can materialize/cache DataFrames in memory, but this cannot well capture the graph structural information. To this direction, we aim to provide a solution for this limitation,

leveraging the potentials of *graph materialized views* and the previous graph query workload. In Particular, we aim to use the information from the previous query workload to list and materialize the most frequent substructures and proprieties that will be stored (preferably in memory) for accelerating the incoming graph queries.

It is worthy to note that, graph materialization has its side effects regarding the memory space that you need to sacrifice for keeping those views. This also comes with another challenge concerning the selection of best proper views (i.e. graph substructures and properties of interest) to materialize and keep in memory [25]. This means that Materialization in our case, will take into consideration the graph structure. Thus materialization will be only for specific 'frequent/hot' sub-structures rather than materializing the entire query results.

RQ4: Graph Cost-Based Optimization (How can we use graph CBO for better performance?): In general, Spark SQL uses the *Catalyst* optimizer to optimize all the queries written both in explicit SQL or in a *DataFrame Domain Specific Language* (DSL). Basically, Catalyst is a Spark library built as a relational-based optimization engine. Each rule in the rule-based part of Catalyst focuses on a specific optimization. Catalyst can also apply various relational cost-based optimizations for improving the quality of multiple alternative query physical execution plans. Although there are several efforts for optimizing the cost-based techniques in Spark-SQL such the work proposed recently in [6] optimizing (Generalized Projection/Selection/Join) queries, these optimizations are not graph-aware cost optimizations. To this extent, providing *Graph-aware Cost-Based-Optimizations* (GCBOs) for selecting the best execution plan of the graph query (using best guess approach that takes into account the important *graph structural information/statistics* about the graph dataset instead of basic relational statistics) will have better optimization and performance for addressing such graph queries in Spark.

To tackle this challenge, we aim to provide a graph-aware query planner which will be implemented as a layer on top of the default Spark Catalyst for providing a *GCBO* query plan, taking into account the statistics of the property graph that resides in Morpheus storage backend. Particularly, the new graph planner/optimizer can select the best join of tables order based on *selectivity* and *cardinality estimations* of graph patterns in the graph query for *filter* and *join* operators. Therefore, at the query time, the new GCBO can suggest a more optimized query plan for the Catalyst to follow.

4 PRELIMINARY RESULTS

In this section, we describe our initial experimental results for answering `RQ1`. In particular, we have designed a set of *micro* and *macro* benchmarking experiments for evaluating the performance of different Spark storage backends supported by the `Morpheus framework`. These storage backends are: Neo4j, Hive, and HDFS with its different file formats (CSV,Parquet, and ORC). Notably, we don't copy data from these Spark storage backends, we only evaluate Morpheus performance with the data already resides in them. We have used the Cypher *LDBC Social Network Benchmark (SNB) BI* benchmark query workload[12]. Our selected queries are read only queries (i.e. no updates are supported by Morpheus).

Hardware and Software Configurations: Our experiments have been performed on a Desktop PC running a Cloudera Virtual Machine (VM) v.5.13 with Centos v7.3 Linux system, running on Intel(R) Core(TM) i5-8250U 1.60 GHzX64-based CPU and 24 GB DDR3 of physical memory. We also used a 64GB virtual hard drive for our VM. We used Spark V2.3 parcel on Cloudera VM to fully support Spark-SQL capabilities. We used the already installed Hive service on Cloudera VM (version:hive-1.1.0+cdh5.16.1+1431), and neo4j V3.5.8.

Benchmark Datasets: Using the *LDBC SNB data generator*[13], we generated a graph data set (in *CSV* format) of *Scale Factor* (SF=1). We used this data to create a property graph in Neo4j using `Neo4j import` tool[14]. The generated property graph has more than 3M nodes, and more than 17M relationships. We also created a graph of tables and views of the same schema inside Hive. Further, we used `Morpheus` Framework to read this property graph either from Hive or Neo4j to store the same graph into HDFS into Morpheus supported file formats (CSV, ORC, Parquet).

For both experiments (Micro and Macro Benchmarking), we run the experiments for all queries five times (excluding the first *cold-start* run time, to avoid the warm-up bias, and computed an average of the other four run times). Notably, we take the *(*ln) function of average run times in the Macro-Benchmark experiment[15].

Morpheus Macro-Benchmark: For the Macro Benchmarking experiment, we selected 21 BI queries (i.e. which are valid to run in the current Morpheus Cypher constructs)[16]. The results of Figure 4 show that `Hive` has the lowest performance in general for running most of the queries even those that are not complex with 70% of low performance than others. HDFS backends in general outperform Neo4j and Hive with 100% of better performance. In particular, Parquet format in HDFS has the best performance. It outperforms ORC and CSV format in most cases of running the queries with 42%. While both CSV and ORC achieve only 28.5% of higher performance.

Morpheus Micro-Benchmark: In our Micro-benchmark experiment, we run 18 Atomic/micro level BI queries[17]. The results of Figure 5 show that Neo4j has the lowest performance in general for running the first 12 queries with 66% of low performance than others. Hive starts to perform worse than Neo4j (and all other systems) only when the number of joins increase and sorting being applied on queries from *Q13* to *Q18*. While, HDFS backends in general outperform Neo4j and Hive with 94.4% of better performance. In particular, Parquet format in HDFS has the best performance, it outperforms ORC and CSV in most queries with 55%. While CSV and ORC only outperform with 22.2% and 16.6%, respectively.

5 CONCLUSIONS AND FUTURE WORK

We are living in an era of continuous huge growth of more and more connected data. Querying and processing large graphs is an interesting and challenging task. The Morpheus framework aims at integrating Cypher query language to work as a graph query language on top of Spark. Morpheus translates Cypher queries into relational Dataframes operations that can fit in the Spark-SQL environment. Morpheus depends mainly on default Spark Catalyst optimizer for optimizing those relational operators. No graph indexing nor graph materialized views are maintained in Morpheus or Spark SQL framework for optimizing property

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

[13]https://github.com/ldbc/ldbc_snb_datagen

[14]https://neo4j.com/docs/operations-manual/current/tools/import/

[15]The code and results of our initial experiments is available on https://github.com/DataSystemsGroupUT/MorephusStorageBenchmarking

[16]http://bit.ly/2W5b01N Macro LDBC SNB BI Queries

[17]http://bit.ly/2Pa4TrF Micro/Atomic level queries

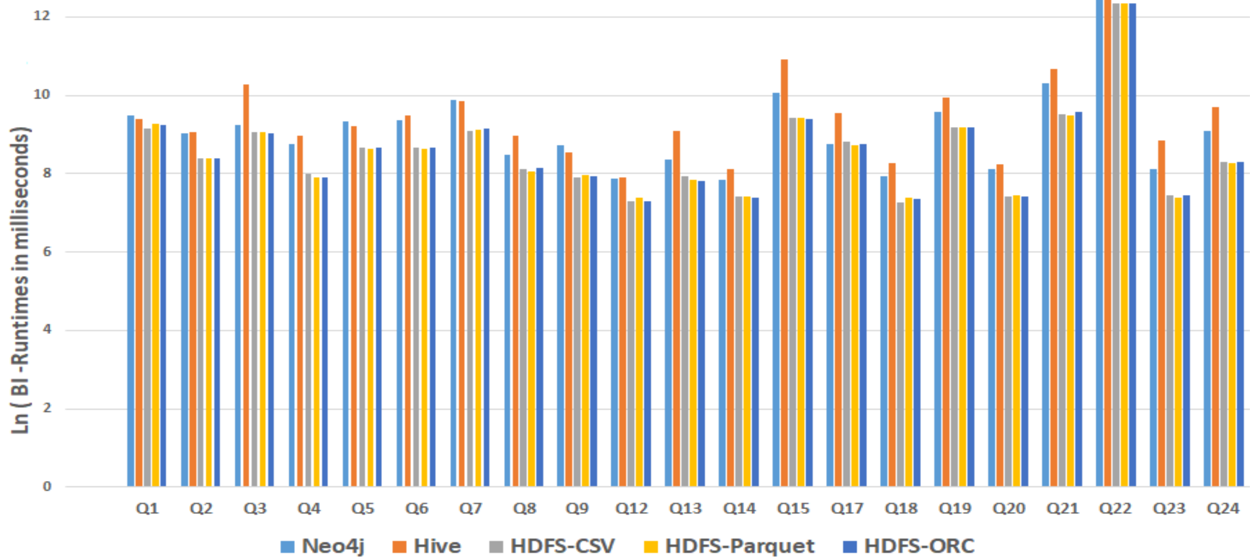

Figure 4: SNB-BI query results

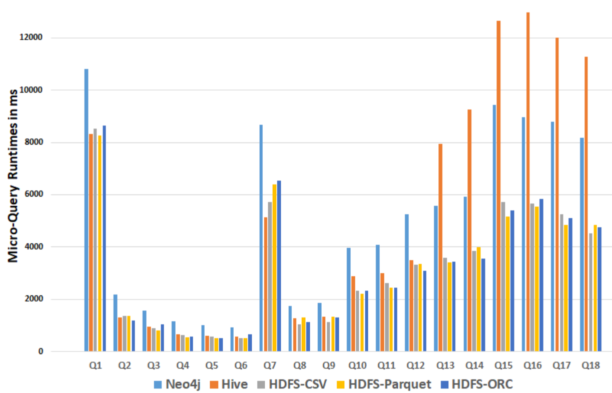

Figure 5: Atomic level query results

graph data querying and processing. In this PhD work, we focus on tackling these challenges by designing an efficient storage backend for persisting property graphs for `Morpheus`. In addition, we aim at providing *graph-aware* techniques (e.g., indexes, materialized views) for Spark to optimize the graph queries, in addition to other graph-aware CBO for Spark Catalyst Optiomizer. We believe that achieving these contributions as our future research plan can significantly enhance the performance of executing graph queries using the `Morpheus` framework.

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 68.

[2] Michael Armbrust et al. 2015. Scaling spark in the real world: performance and usability. *PVLDB* 8, 12 (2015).

[3] Michael Armbrust et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*.

[4] Michael Armbrust, Doug Bateman, Reynold Xin, and Matei Zaharia. 2016. Introduction to spark 2.0 for database researchers. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2193–2194.

[5] Ramazan Ali Bahrami, Jayati Gulati, and Muhammad Abulaish. 2017. Efficient processing of SPARQL queries over graphframes. In *Proceedings of the International Conference on Web Intelligence*. ACM, 678–685.

[6] Lorenzo Baldacci and Matteo Golfarelli. 2018. A Cost Model for SPARK SQL. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2018), 819–832.

[7] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema validation and evolution for graph databases. *arXiv preprint arXiv:1902.06427* (2019).

[8] Hirokazu Chiba, Ryota Yamanaka, and Shota Matsumoto. 2019. Property Graph Exchange Format. *arXiv preprint arXiv:1907.03936* (2019).

[9] Nadime Francis et al. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD*.

[10] Abel Gómez, Amine Benelallam, and Massimo Tisi. 2015. Decentralized model persistence for distributed computing.

[11] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[12] Chuang-Yi Gui et al. 2019. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34, 2 (2019).

[13] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, 1155–1164.

[14] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2015. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1566–1577.

[15] Karsten Klein, Nils Kriege, and Petra Mutzel. 2011. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*.

[16] Egor V Kostylev et al. 2015. SPARQL with property paths. In *ISWC*.

[17] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph summarization methods and applications: A survey. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 62.

[18] Bingqing Lyu, Lu Qin, Xuemin Lin, Lijun Chang, and Jeffrey Xu Yu. 2016. Scalable supergraph search in large graph databases. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 157–168.

[19] Marko A Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *PDPL*.

[20] Gábor Szárnyas. 2019. Query, analysis, and benchmarking techniques for evolving property graphs of software systems. (2019).

[21] Oskar van Rest et al. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*.

[22] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. 2017. Big graph analytics platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195.

[23] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 335–346.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. [n.d.]. Spark: Cluster computing with working sets. ([n. d.]).

[25] Yan Zhang. 2017. *Efficient Structure-aware OLAP Query Processing over Large Property Graphs*. Master's thesis. University of Waterloo.

[26] Peixiang Zhao, Jeffrey Xu Yu, and Philip S Yu. 2007. Graph indexing: tree+ delta<= graph. In *PVLDB*.