Matthias Hauck, Marcus Paradies, Holger Fröning, Wolfgang Lehner, Hannes Rauhe

Highspeed Graph Processing Exploiting Main-Memory Column Stores

Erstveröffentlichung in / First published in:

Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien, 24.-25.08.2015. Springer, S. 503-514. ISBN 978-3-319-27308-2.

DOI: http://dx.doi.org/10.1007/978-3-319-27308-2_41

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

Highspeed Graph Processing Exploiting Main-Memory Column Stores

Matthias Hauck[1(✉)], Marcus Paradies[2], Holger Fröning[1], Wolfgang Lehner[2], and Hannes Rauhe[3]

[1] Computer Engineering Group, Ruprecht-Karls University of Heidelberg, Heidelberg, Germany
{matthias.hauck,holger.froening}@ziti.uni-heidelberg.de
[2] Database Systems Group, Tu Dresden, Dresden, Germany
m.paradies@sap.com, wolfgang.lehner@tu-dresden.de
[3] SAP SE, Weinheim, Germany
hannes.rauhe@sap.com

Abstract. A popular belief in the graph database community is that relational database management systems are generally ill-suited for efficient graph processing. This might apply for analytic graph queries performing iterative computations on the graph, but does not necessarily hold true for short-running, OLTP-style graph queries. In this paper we argue that, instead of extending a graph database management system with traditional relational operators—predicate evaluation, sorting, grouping, and aggregations among others—one should consider adding a graph abstraction and graph-specific operations, such as graph traversals and pattern matching, to relational database management systems. We use an exemplary query from the interactive query workload of the LDBC social network benchmark and run it against our enhanced in-memory, columnar relational database system to support our claims. Our performance measurements indicate that a columnar RDBMS—extended by graph-specific operators and data structures—can serve as a foundation for high-speed graph processing on big memory machines with non-uniform memory access and a large number of available cores.

1 Introduction

The proliferation of graph-shaped data in the enterprise domain and the ever-growing need to process billion-scale graphs efficiently are the key drivers for the evolution of a plethora of graph processing systems targeting large cluster installations [8,11,13]. Analytic graph queries on static graphs of an immense scale can be executed by these systems in an acceptable execution time. While there has been a large body of work focusing on distributed, shared-nothing graph processing frameworks and graph algorithms, transactional graph workloads comprising short-running, concurrent queries from a large number of client applications have been largely ignored by the research community so far. Such *interactive* queries usually access a small fraction of the entire graph and perform selective filter operations on vertex and edge attributes based on some

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

predicates, run simple graph traversals to explore the neighborhood of a vertex or a group of vertices, and aggregate attribute values.

The execution of interactive graph queries is well-supported in graph database management systems (GDBMSs), such as Neo4j [2], Sparksee [14] or InfiniteGraph [1]. GDBMSs rely on the property graph model, where a graph consists of a set of vertices and a set of edges with an arbitrary number of attributes assigned to them [17]. Additionally, they provide a clear graph abstraction, intuitive declarative or imperative graph query interfaces, and specialized storage and execution capabilities to process graph queries efficiently. Besides functional advantages over RDBMSs, GDBMSs claim to offer superior performance for graph-specific operations—graph traversals are one prominent example—by storing the graph topology in an index-free vertex adjacency structure. While they offer good performance for topology-centric queries, GDBMSs usually suffer from poor performance for attribute-centric queries that perform predicate evaluation on or aggregating of attribute values. The interactive query workload of the LDBC benchmark suite—the de-facto standard benchmark for graph processing—indicates that interactive graph queries demand a seamless integration of query processing on the graph topology and access to vertex/edge attributes in the same database system [7].

Based on an in-depth study of the LDBC interactive query workload and the *choke points* defined by the benchmark committee[1], we revisit the question whether RDBMSs are generally ill-suited for graph processing compared to native GDBMSs. Instead, we argue that large fractions of the graph queries from the interactive workload of the LDBC can benefit from an optimized handling of vertex/edge attributes and the efficient predicate evaluation of point and range queries on it. Based on these observations we identify graph-specific operations that have to be added to a RDBMS to complement the already available processing capabilities for aggregations, sorting, and predicate evaluation. We implemented one exemplary query from the LDBC interactive workload on top of a columnar RDBMS prototype, which we enhanced by graph operators and data structures. We tailor our graph operators and data structures to run on server machines with large amounts of available memory, a large number of cores, and non-uniform memory access (NUMA). Our contributions can be summarized as follows:

- Based on a detailed study of the interactive query workload of the LDBC benchmark suite, we derive functional and non-functional requirements for a columnar RDBMS with integrated graph processing capabilities and specify a set of operations that are required to process the exemplary query.
- We propose a novel architecture that extends a columnar RDBMS by adding a native graph abstraction and a graph-specific secondary index structure.
- We perform an experimental evaluation based on an exemplary query from the interactive query workload of the LDBC social network benchmark and demonstrate that our hybrid approach scales with increasing dataset sizes and also provides a NUMA-friendly graph abstraction that can leverage all available computing resources efficiently.

[1] http://ldbcouncil.org/sites/default/files/LDBC_D2.2.1.pdf.

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

2 Related Work

Although distributed, shared-nothing graph processing —inspired by Pregel [13]— recently received a considerable amount of attention in the industry and research community, there is also an increased interest to store and process large graphs on a single machine (either on disk/flash or in memory). In fact, there is evidence that even notebooks are able to outperform distributed graph processing systems in certain scenarios [10]. Graph databases, such as, Neo4j [2], Titan [4], InfiniteGraph [1], Sparksee [14], and OrientDB [3], are gaining popularity for enterprises and provide native graph storage, querying and transaction support. All of them are native graph systems with own implementations to guarantee transactionality, logging, and recovery and therefore cannot be used as a dedicated engine inside a RDBMS.

For graph analytics, where no transactions and no update support is required, there is a wide variety of single-node graph processing systems targeting multi-core server machines with a large number of cores and a vast amount of memory available. System G is a read-only graph processing system leveraging a key-value store as persistence, a set of compressed sparse row (CSR) like data structures, and is optimized towards CPU cache reuse and parallelization [21]. In comparison to our approach, they do not use a column store as primary persistence to efficiently process also vertex/edge attributes.

PGX.ISO is an in-memory graph processing system for querying graph data using graph patterns [16] and analytic queries using Green-Marl [9]. The focus of PGX.ISO is on read-only workloads aiming for minimizing query response time. We aim at providing an integrated solution as part of a RDBMS and design our system to also cope with transactions and concurrent write operations. Moreover, our approach can perform graph querying on the latest version of the data, without the need to replicate it into a dedicated graph system.

Recent graph processing systems, such as, Llama, target not only static graphs, but also allow modifications to the vertex/edge properties and the graph topology [12]. Llama implements a mutable CSR data structure and stores multiple snapshots of the graph, one for each update to the graph. Since the entire system is built on top of a CSR with own containers for the vertex/edge properties, it is not possible to integrate Llama into a RDBMS seamlessly.

Recent advances in modern hardware, including the availability of large amounts of main memory with non-uniform access (NUMA), multi-core, and SIMD instructions triggered interesting discussions in these emerging hardware technologies [5,6,22]. EmptyHeaded is a graph pattern matching engine based on a configurable CSR data structure and compiles queries into boolean algebra expressions leveraging SIMD parallelization [5]. The focus, however, is on read-only workloads with topology-centric queries and cannot be used easily to support dynamic graphs. Cui et al. propose a set of techniques to improve the performance and scalability of breadth-first search on large NUMA machines by minimizing cross-socket communication [6]. Since we are aiming at providing general graph processing capabilities instead of specific algorithms, their results can be incorporated into our system. Zhang et al. investigate the effect of remote memory accesses for graph

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

analytics and propose POLYMER, a NUMA-aware graph analytics system [22] with a focus on read-only workloads and a small number of attributes on vertices and edges. We believe that some of the proposed techniques concerning access to the graph topology can be also applied in our system.

Increasingly, even RDBMS provide graph support to some extent on relational tables [18–20]. Welc et al. show that a RDBMS can serve selected graph use cases (e.g., shortest path) by heavy indexing of the graph topology using B-trees and outperform native graph databases. Instead of exposing graph processing directly to SQL, Sun et al. translate GREMLIN, a traversal-oriented graph query language, into SQL statements. The deepest level of integration into a RDBMS has been proposed by providing a native graph query language, a native execution engine, and graph-aware traversal operators inside the database kernel [18].

The LDBC benchmark suite is a community effort to define a standard benchmark for graph query processing and consists of three different workloads, an *analytic*, a *business intelligence*, and an *interactive* workload [7]. While the analytic workload focuses on long-running, offline queries potentially accessing the entire graph and computing some global graph measure, the interactive queries perform transactional, short-running requests accessing only a small fraction of the graph with the focus on the selection of vertices/edges according to some predicate filter, simple aggregations, and graph traversals.

To summarize, none of the discussed graph system focuses on transactional graph workloads with interleaved read and write operations—except for native graph databases. On the other extreme, RDBMS provide transactional guarantees, but lack a high-speed graph processing layer in the form of tailored data structures for storing the graph topology. Our long-term goal is to close this gap by proposing a combination of the best of both worlds, transactional guarantees of a RDBMS combined with high-speed graph processing capabilities of a native graph processing system.

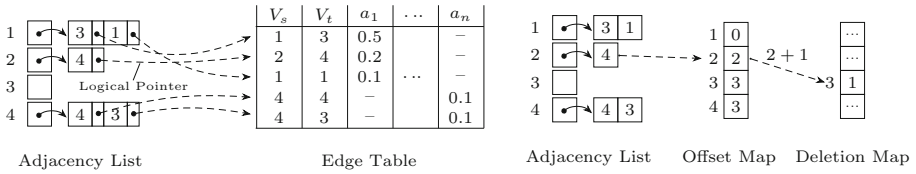
3 System Architecture

In this section we discuss the requirements and the design of GRAPHITE, a columnar RDBMS prototype with a native graph abstraction, graph operators, and graph-specific secondary index structures [15]. We designed our system with the following design goals in mind:

Native Graph Abstraction. Relational operators are not well-suited to process native graph operations, such as traversals and graph pattern matching, efficiently. Hence, GRAPHITE uses a native graph abstraction to implement graph-specific operations and redirects set-based operations, such as predicate evaluation or aggregations, to the relational operators of the RDBMS.

Query Performance. The query performance of interactive graph queries in GRAPHITE should be close or even exceed the performance of native graph databases. This requires exposing a low-level graph interface that sits on top of the graph data structures and can be used in combination with relational operators.

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41



(a) Adjacency list and mapping to edge table. (b) Access to deletion map.

Fig. 1. Usage of the adjacency list in conjunction with the relational edge table and in the presence of edge deletions.

Space Efficiency. Saving memory bandwidth is one of the keys of achieving query performance scalability on large server machines. We support lightweight compression techniques, like run-length encoding, dictionary encoding, and sparse encoding on the columns, and store only internal, dense numerical identifiers in the secondary index structures to keep the memory footprint low.

Integratability into a RDBMS. GRAPHITE is designed to be integrated as a component into a columnar RDBMS, where the primary copy of the data is kept in relational tables, accelerated by additional secondary graph index structures to store the graph topology.

Transactionality. We target interactive graph queries interleaved with concurrent updates on the graph data. Therefore a separate graph engine that operates on a snapshot of the data is not feasible. Our system inherits the transaction concept of the RDBMS and enables transactional query processing directly on the graph data. Due to space constraints, we do not discuss this aspect in detail in the course of this paper.

3.1 Columnar Graph Storage

GRAPHITE stores a graph in a columnar storage representation consisting of two column groups, one for vertices and one for edges. We describe a vertex by a unique identifier and an edge by a tuple of source and target vertex and an implicit edge direction. Both vertices and edges can have an optional type and an arbitrary set of attributes. Each column group is divided into a static, highly-compressed read partition and a dynamic, append-only write partition. To lower the overall memory consumption we can apply light-weight compression techniques on the columns, like dictionary encoding, run-length encoding, and sparse encoding.

3.2 Secondary Graph Index Structure

The in-memory representation of the graph topology is not well-suited for fast and fine-granular topological graph operations, such as the retrieval of outgoing edges for a given vertex, as it has a time complexity of $\mathcal{O}(|E|)$.

To efficiently support fine-granular topological operations, we provide a secondary index structure to store the graph topology—while the attributes remain

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

in relational tables. We chose an adjacency list representation, since it supports fast graph operations and can handle updates of the graph topology gracefully. Our adjacency list consists of the core data structure to store the graph topology and several auxiliary data structures to support updates, deletions, and combined processing with vertex/edge attributes stored in the relational tables. Figure 1a illustrates the adjacency list and the interplay with the edge table through logical indexes that point to the corresponding entry in the table. We use a similar mechanism to address records in the vertex table. To support bi-directional topological operations, the adjacency list can be stored for both traversal directions.

We support deletion operations in GRAPHITE through an offset map and a deletion map as depicted in Fig. 1b. To avoid the cost of copying and reorganizing parts of the adjacency list when an edge deletion occurs, we use a bitmap to invalidate the corresponding entry and periodically reorganize the adjacency list data structure in a batch processing step. We address each edge in the deletion map using the relative position of the edge in the source vertex list and an offset for the source from the offset map. For example, the position of edge $e = (2, 4)$ can be computed from the summation of the offset found at position 2 in the offset map and the relative position ($p = 1$) of vertex 4 in the neighborhood list of vertex 2.

Additionally, the deletion map can be used for a reoccurring pattern in graph queries: a part of the graph is selected using predicates and subsequent operations are only executed on this subgraph. A subgraph is a lightweight materialized view requiring one bit per edge and is always connected to the complete adjacency list. It has its own deletion map, in which only vertices and edges are valid that fulfill the predicate criteria. Operations on the subgraph are performed similar to operations on the adjacency lists, except for the use of a dedicated validity map.

4 Implementation Details

We implemented our prototype in C++ and used for parallelization the INTEL TBB library[2]. Our implementation is tailored towards utilizing the available hardware resources as much as possible by using cache-friendly, concurrent data structures and parallelizable algorithms as basic building blocks, such as, duplicate detection and the retrieval of adjacent vertices. Although our focus is on improving the response time for graph operations, a careful implementation is also required for utility functions, such as predicate parsing and evaluation. For short-running queries, the parsing of a predicate can even outweigh the actual predicate evaluation and therefore should be implemented with care. In the following we provide implementation details on two important aspects of our system—the Graph API and the implementation of interactive graph queries.

4.1 Basic Graph Operations & Building Blocks

The Graph API provides a unified access to the graph topology stored in an adjacency list and vertex/edge attributes stored in the corresponding relational tables.

[2] https://www.threadingbuildingblocks.org.

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

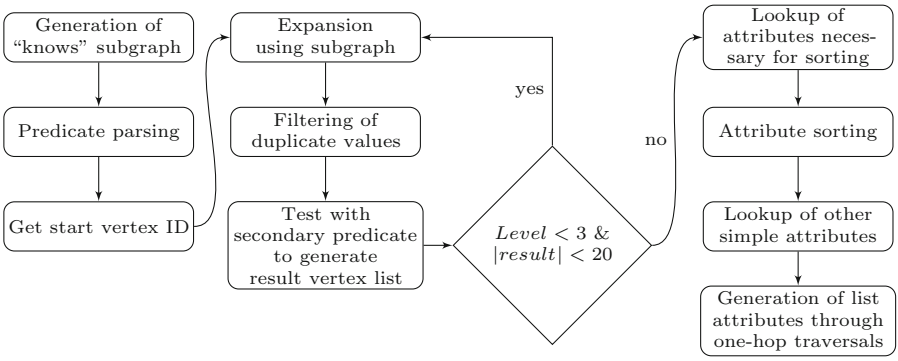


Fig. 2. Flow diagram of our implementation of Q1 from the interactive query workload of the LDBC.

We implement the Graph API such that topological operations, such as, the edge expansion for a given set of vertices, can be seamlessly combined with relational operations, specifically the retrieval of vertex/edge attributes and the evaluation of predicates on sets of vertices/edges. We represent a vertex/an edge by a unique identifier—32 bit or 64 bit—and sets of vertices/edges as dense or sparse data structures—bitsets or dynamic arrays—depending on the cardinality of the set.

We identified several building blocks that can be found in most interactive graph queries from the LDBC benchmark, such as, a conditional edge traversal with a given vertex/edge predicate and duplicate filtering on multiple vertex/edge multisets. For topological operations, we parallelize the calls to the adjacency list in the backend and use multiset semantics. For often used vertex/edge predicate combinations, we provide a lightweight subgraph concept that stores a materialized view qualified by a set of vertex/edge predicates.

Efficient duplicate filtering is crucial for achieving superior query performance, especially for traversals over multiple steps. We implemented two differences approaches for duplicate filtering: (i) a hash-based and (ii) a sort/merge-based approach. The hash-set approach is fully parallelized and uses a concurrent hash-set to probe for vertex/edge identifiers and to insert them concurrently into a new set of discovered vertices/edges. The sort/merge approach first sorts the input in parallel, followed by a merge operation with the previous discovered vertices.

4.2 Query Implementation

Our system is tailored towards the efficient execution of interactive graph queries that access only a small portion of the complete graph. We use the interactive query workload of the LDBC to verify and evaluate our system in terms of response time, scalability to larger dataset sizes, and functional completeness [7]. In the following we provide a description of the implementation of one exemplary query from the interactive workload. We chose query Q1 as representative query since it covers a large subset of the required functionality to process the entire interactive workload.

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

Query Q1 performs a conditional, multi-level traversal and requires efficient and interleaved access to the graph topology and the attributes. More specifically, Q1 returns up to 20 friends with a specific first name of a given person (via the 3-hop neighborhood). In addition to the set of persons, we also return summaries about their workplaces and places of study, and sort the result ascending by the friendship distance followed by the last name and the identifier.

Figure 2 depicts a flow diagram of our implementation of query Q1 and the used building blocks of the Graph API. We realized the edge expansion in the multi-hop traversal using a subgraph, so that only edges of the friendship relation are expanded. For the subsequent filtering of duplicate vertex IDs we implemented hash-based and sort/merge-based duplicate elimination routines. The retrieval of all vertices with the given name is integrated into the multi-hop traversal, allowing for an early abort of the traversal, when the anticipated number of vertices satisfying the predicate is reached. We perform the sorting of the result at the earliest possible point during the execution to minimize the number of copy operations of materialized attributes. Finally, we fetch the attribute values from the relational tables through single-hop traversals and materialize the result.

4.3 Memory Consumption

The memory consumption M of the adjacency list depends linearly on the number of edges $|E|$ and the number of vertices $|V|$, where the coefficients depend on the implementation. We implement the core adjacency list as two STL vectors of vectors using 64 bit IDs, one for the neighborhood and the other for the corresponding logical pointer to the edge attribute table.

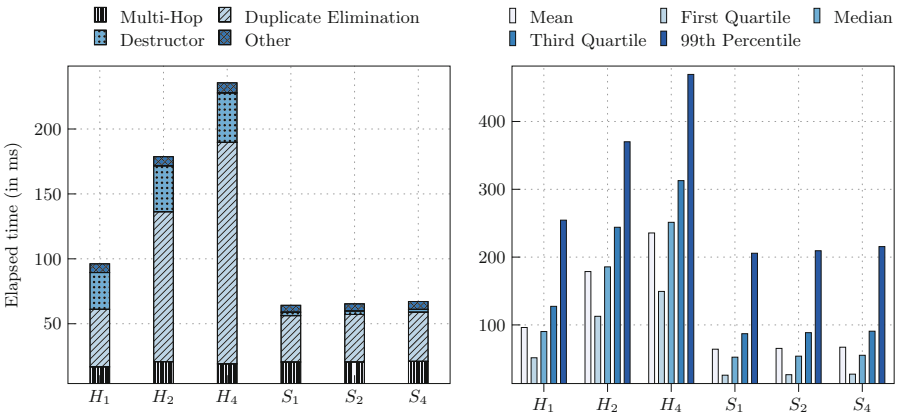
The total memory consumption of a graph in Graphite does not only depend on the adjacency list, but also on the edge and vertex attribute tables. For the LDBC data set of SF1, the adjacency list consumes $M(V, E) = 557\,\text{MB}$(single direction), while the vertex and edge tables consume 1355 MB using a dictionary encoding and 32 bit IDs. These numbers are without overheads.

In general, we find that for larger scale factors of the LDBC data set, the memory footprint of the vertex and edge attributes stored in the relational tables dominates the overall memory consumption, while the space overhead of the secondary adjacency list remains small. Since Graphite stores attributes in a columnar storage layout, lightweight compression techniques can be applied to reduce the memory footprint of the attributes.

5 Evaluation

In this section we evaluate our implementation with focus on how it performs on large NUMA systems. We use a four-socket Linux based system with Intel Xeon X7560 (Nehalem) CPUs. Our system is equipped with 4×8 cores @2.27 GH_z with Hyperthreading enabled, 24 MB last level cache at each socket, and 512 GB DDR3 RAM. We compile Graphite with Intel TBB 4.3 update 3 and GCC 4.8 with

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41



(a) Run time shares of the mean run time.

(b) Run time distribution.

Fig. 3. Elapsed time for Q1 (10k queries) using 8 cores on different number of sockets at SF100. We evaluate two different versions for duplicate elimination—hash-based H_x and sort-based S_x—where $x \in \{1, 2, 4\}$ refers to the number of utilized sockets.

the optimization flags `-O3` and `march=native`, enabling the compiler to use the complete instruction set of the CPU.

We used the data generator of the LDBC benchmark to generate scale factors 1 to 100 and reassembled the output into a single vertex and a single edge table, respectively. We randomly generated representatives of query Q1 by using the generated parameters from the data generation process. In our experiments, we report the total elapsed time of the executed queries, including setup and destruction time of auxiliary data structures.

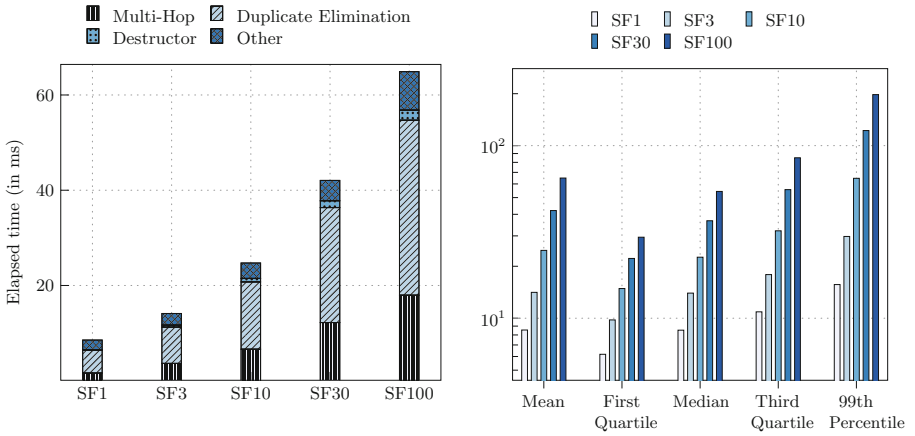
The only exception is the generation of subgraphs, which we exclude from the total elapsed time. Since we currently do not enforce any constraints on the graph—for example that two vertices of type *knows* can be only connected via an edge of type *knows*—we cannot easily partition the graph into isolated subgraphs. To simulate a materialized view enforcing such a constraint on vertex and edge types, we use the concept of a materialized subgraph in our experiments.

5.1 NUMA Effects

In our first experiment we analyze the NUMA behavior of GRAPHITE. We use a constant number of eight cores evenly distributed across different number of sockets and use `numactl` to pin threads to sockets. In Fig. 3a we report the mean run time shares using the two different duplicate filter methods. While the sort-based duplication elimination routine is not affected by the distribution across different sockets, the hash-based routine is sensitive to the thread placement policy.

The reason for this sensitivity is our global concurrent hash-set implementation that relies on atomic insertion operations. Every time a vertex ID is inserted into the concurrent hash-set, a random memory access occurs. If this memory access

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41



(a) Run time shares of the mean run time.

(b) Run time distribution.

Fig. 4. Run time of Q1 (10k queries) using a sorted list duplicate filter at different scale factors (SF) using 32 cores.

is redirected to different socket, the cache coherency protocol needs to be invoked. The expansion operations on the adjacency list (multi-hop) behave similarly to the sort-based duplicate elimination and are unaffected by NUMA. In Fig. 3b we depict the run time distribution of the elapsed time of Q1 for different input parameters. For both routines of duplicate elimination, the run time distribution scales to various input parameters and intermediate result sizes.

5.2 Performance

We evaluate the scalability of our approach using the sort-based duplicate filter for varying scale factors[3]—we use SF1 to SF100—and present our results in Fig. 4a.

The total elapsed time increases slower than the growth of the total data set size—for SF100 the mean run time is only 64.9 ms. For all evaluated scale factors, the largest portion of the total elapsed time is spent in the multi-hop expansion and the subsequent vertex duplicate elimination step. With increasing scale factors, we experienced an increased elapsed time for both steps.

This effect is caused by growing intermediate vertex sets after the expansion and consequently more work to be done during the duplicate elimination. Since the scale factor of the data set does not directly reflect the scaling of the graph topology—SF1 contains about 11,000 vertices of type person and about 400,000 connections between them—and the number of traversal levels is limited to three, the overall run time increase is not proportional to the total data set size.

[3] The numbers presented in Fig. 3a for eight core at SF100 are representative for other scale factors and core numbers: the actual run time of the hash-based approach takes longer and is more expensive to destroy.

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

We experienced a similar behavior when taking a closer look at the run time distribution for different input configurations of Q1 and report on the experimental results in Fig. 4b. Similar to the NUMA experimental results, we verify that the overall run time of Q1 is not directly related to the total data set size, but correlated with the size of the queried subgraph of the graph topology. Further, we experienced that the total query execution depends on the chosen input configuration and the start vertex for the traversal, resulting in significant variations for the size of the 3-hop neighborhood.

The remaining parts of the query evaluation contribute only a minor fraction to the total query execution time. For example, we apply the secondary predicate on the set of vertex IDs after the processing of the duplicate filter that already reduced the number of vertices.

The subgraph generation is not part of the querying process, but is triggered at the beginning of the query session. The elapsed time of the subgraph generation for vertex type *person* and edge type *knows* accounts with about 40.8 ms for SF1 and grows linearly to 4876.9 ms for SF100.

6 Conclusion

We presented GRAPHITE, a columnar RDBMS architecture and implementation—extended by a native graph abstraction and a graph-optimized secondary data structure—that allows seamlessly combining graph with relational operations in the same database engine. Based on a detailed study of the interactive query workload of the LDBC benchmark, we derived requirements that have to be fulfilled to support interactive queries directly on graph data stored in a RDBMS. To improve the response time of topological queries, we introduced an adjacency list as a secondary index structure that is tightly coupled with the corresponding vertex/edge attribute tables. Our prototypical implementation of Q1 from the interactive query workload of the LDBC shows a NUMA-friendly behavior, the run time scales with the dataset size of the query and shows competitive query performance compared with native GDBMS in other publications [7]. This work is currently in progress, so as next steps we plan to extend our experimental evaluation to other queries from the LDBC benchmark and to evaluate GRAPHITE in the presence of concurrent, transactional write operations.

References

1. InfiniteGraph project website. www.objectivity.com/infinitegraph
2. Neo4j project website. http://neo4j.com
3. OrientDB project website. http://www.orientdb.org/
4. Titan project website. http://thinkaurelius.github.io/titan
5. Aberger, C.R., Nötzli, A., Olukotun, K., Ré, C.: EmptyHeaded: boolean algebra based graph processing, CoRR abs/1503.02368 (2015)
6. Cui, Z., Chen, L., Chen, M., Bao, Y., Huang, Y., Lv, H.: Evaluation and optimization of breadth-first search on NUMA cluster. In: Proceedings of CLUSTER 2012, pp. 438–448 (2012). http://dx.doi.org/10.1109/CLUSTER.2012.29

Final edited form was published in "Euro-Par 2015: Parallel Processing Workshops Euro-Par, International Workshops. Wien 2015", S. 503-514, ISBN 978-3-319-27308-2
http://dx.doi.org/10.1007/978-3-319-27308-2_41

7. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.A.: The LDBC social network benchmark: interactive workload. In: Proceedings of SIGMOD 2015, pp. 619–630 (2015)
8. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: graph processing in a distributed dataflow framework. In: Proceedings of OSDI 2014, pp. 599–613 (2014)
9. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In: Proceedings of ASPLOS 2012, pp. 349–362 (2012)
10. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI 2012, pp. 31–46. USENIX Association, Berkeley (2012)
11. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
12. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: LLAMA: efficient graph analytics using large multiversioned arrays. In: Proceedings of ICDE 2015 (2015)
13. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of SIGMOD 2010, pp. 135–146 (2010)
14. Martínez-Bazan, N., Águila Lorente, M.A., Muntés-Mulero, V., Dominguez-Sal, D., Gómez-Villamor, S., Larriba-Pey, J.L.: Efficient graph management based on bitmap indices. In: Proceedings of IDEAS 2012, pp. 110–119 (2012)
15. Paradies, M., Lehner, W., Bornhövd, C.: GRAPHITE: an extensible graph traversal framework for relational database management systems. In: Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM 2015, pp. 29:1–29:12 (2015)
16. Raman, R., van Rest, O., Hong, S., Wu, Z., Chafi, H., Banerjee, J.: PGX.ISO: parallel and efficient in-memory engine for subgraph isomorphism. In: Proceedings of GRADES 2014, pp. 5:1–5:6 (2014)
17. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bull. Am. Soc. Inf. Sci. Technol. **36**(6), 35–41 (2010)
18. Rudolf, M., Paradies, M., Bornhövd, C., Lehner, W.: The graph story of the SAP HANA database. In: Proceedings of BTW 2013, pp. 403–420 (2013)
19. Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., Xie, G.: SQLGraph: an efficient relational-based property graph store. In: Proceedings of SIGMOD 2015 (2015)
20. Welc, A., Raman, R., Wu, Z., Hong, S., Chafi, H., Banerjee, J.: Graph analysis: do we have to reinvent the wheel? In: Proceedings of GRADES 2013, pp. 7:1–7:6 (2013)
21. Xia, Y., Tanase, I.G., Nai, L., Tan, W., Liu, Y., Crawford, J., Lin, C.: Explore efficient data organization for large scale graph analytics and storage. In: Proceedings of BigData 2014, pp. 942–951 (2014)
22. Zhang, K., Chen, R., Chen, H.: NUMA-aware graph-structured analytics. In: Proceedings of SIGPLAN 2015, pp. 183–193 (2015)