Towards a Novel Concept for High Performance Graph Processing

Matthias Hauck
Ruprecht-Karls Universität Heidelberg/SAP SE
matthias.hauck@sap.com

Abstract

With the increasing importance of graph-structured data and the rising number of applications relying on a graph data model, there is a growing need to store, manipulate, and analyze graph data efficiently. To satisfy this need, a plethora of solutions, tailored for specific use cases and environments, has been proposed. This includes graph database management systems that provide a high-level abstraction, query optimization, transactions, and support for dynamic graphs, and graph processing engines that provide contrary a low-level abstraction and superior performance for graph algorithms. As of now, combining the advantages of both classes requires expensive data transfer and poses data synchronization challenges.

We propose a graph engine concept that combines the advantages of a high-level graph interface to formulate and optimize graph algorithms with the performance advantages of a high-performance graph processing engine. Our concept fosters an adaptive execution strategy allowing to correct inadequate optimization decisions made at compile time and handles dynamic graphs, multiple vertex and edge attributes, and concurrently running graph operations gracefully.

1 Observations & Motivations

Different fields have applications using graph-shaped data with divergent properties. Depending on the application, the graphs have different magnitudes of numbers of vertices and edges, diameters, vertex in/out degree distributions (homogeneous or heterogeneous), and numbers of vertex and edge attributes. Social networks typically expose a small diameter and a power law vertex degree distribution. In contrast, road networks typically have a large diameter and low vertex in/out degrees.

The applications expose also different data modification patterns: these pattern range from no modifications, over infrequent batch-wise modifications, to frequent modifications. Furthermore, these modifications can happen concurrently with other operations.

On graphs a broad range of algorithms could be applied, with different access patterns and intermediate working set sizes: For example, a textbook PageRank algorithm processes all vertices uniform in every iteration. Breadth-first search has typically a varying workload that depends on the number of newly discovered input vertices from the previous iteration (frontier queue) and the number of edges of these vertices that it processes in each iteration.

For several algorithms optimizations have been proposed, which address different aspects: asynchronous execution could enhance algorithmic properties like convergence speed [6], direction optimization of traversals [1] leverages graph properties, and push/pull communication optimizations could improve data access or information propagation [7]. The right selection of these optimizations, in combination with good choices in other areas like resource management, data structures, and parallelization, is crucial for achieving good performance.

We see two general approaches to process graph data in a single-node system: graph processing engines (GPEs) and graph database management systems (GDBMSs). GPEs like Galois [4] and Ligra [5] are designed for high-speed graph processing on immutable graphs. To maximize the performance, they provide fast primitives for graph processing and features for algorithmic and hardware adaptivity. Graph algorithms are usually implemented against an API offered by the GPE. An exception is Green-Marl [3], a high-level, domain-specific language (DSL) for graph analysis, which is translated into an optimized, low-level, platform-specific implementation using a source-to-source compiler.

GDBMSs like Neo4j[1] and Sparksee[2] provide a native, mutable graph store, support for attributes, querying, transaction support, logging, and recovery. Most GDBMSs use the property-graph model, which relies on a multi-relational graph, with attributes linked to vertices and edges as key/value pairs. Algorithms for graph analysis in GDBMSs are implemented against an API, which does not offer opportunities for optimization between API calls. Declarative DSLs like Neo4j's Cypher exist for graph pattern matching that separate the query description from the actual implementation.

To the best of our knowledge, there is no graph data management system available that provides a high-level abstraction, query optimization, and support for dynamic graphs as in a GDBMS and the performance and hardware adaptivity of a tuned GPE.

2 Graph Engine Concept

We envision a GPE that provides the performance of a standalone GPE, while being integrated into a DBMS and thereby leveraging all the available features, such as graph data manipulation, query optimization, graph statistics, and transaction support. Specifically, we derive the following conceptual considerations:

Generality: We see a strong demand for a general-purpose GPE, which provides an interface for abstract graph algorithm descriptions, i.e., in form of an abstract syntax tree (AST) from a DSL. Instead of being specifically tuned, the GPE is able to process the abstract algorithm description, map the algorithm to an optimized operator plan, and finally to execute the plan.

Adaptivity: We anticipate that off-line decisions including algorithm selection and runtime configuration are not appropriate for all cases. Estimations of the cost of a graph algorithm implementation can be inadequate, because of the data-dependent behavior at execution time. It seems promising

[1] `http://neo4j.com`
[2] `http://www.sparsity-technologies.com`

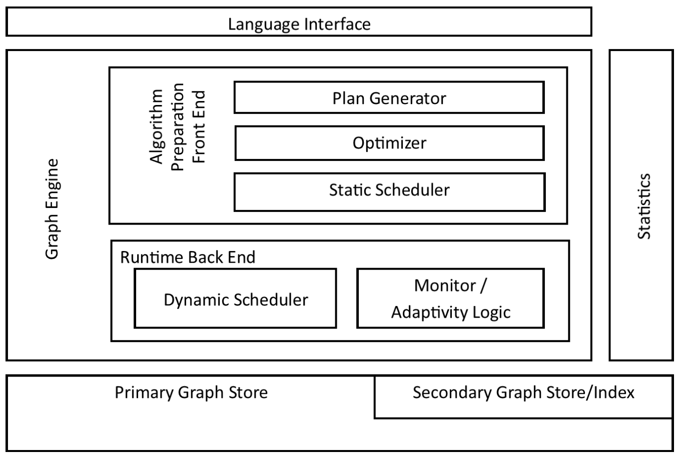

Figure 1: GPE architecture and intermediate environment

to introduce a concept that supports on-line monitoring and on-line adaption, while still being able to perform optimizations on the graph algorithms at compile time.

Integration: Instead of using a separate GDBMS for processing the transactional graph data and a GPE for graph analysis tasks, we foster a bridging solution that allows graph computations without data movement between different systems. The GPE supports features of GDBMSs like transaction support and relies on the property-graph model to represent graph data.

3 Proposed Design

We outline the general architecture and the basic building blocks of our GPE in Figure 1. The environment of the GPE is a non-distributed, relational, in-memory DBMS context. Two columnar tables store the graph topology and vertex and edge attributes, one for vertices and one for edges, respectively. To accelerate topology-centric operations, we store the graph topology in a secondary index store, i.e., an adjacency list [2].

Additionally, the GPE can use statistics based on the graph data and previous GPE executions. Above the GPE is a language interface, i.e., from a DSL that provides the GPE a high-level algorithm description in form of an AST for execution. We divide the GPE into two parts: A algorithm preparation front end and a runtime back end.

3.1 Algorithm Preparation Front End

The front end receives an AST from the language interface and performs a preprocessing on the algorithm description to create the final execution plan. We divide the front end into several parts: a plan generator, an optimizer, and a static scheduler.

The plan generator constructs an initial operator plan from the AST. Therefore it maps the AST to a logical operator plan, performs logical optimizations on this plan, and a decomposition into independent parts. The optimizer uses the logical operator plan and maps it to the physical operators. For this mapping, the optimizer uses a cost estimation, which is based on an analysis of the algorithm description and available statistics. Finally, the static schedule plans a mapping from operators to resources and creates the final execution plan. It optimizes the (intermediate) data placement and the execution ordering and placement of the operators. Here the static scheduler has to trade off data movement against useful parallelism in and between operators.

3.2 Runtime Back End

The runtime back end is responsible for the dynamic execution of the execution plan and its operators. Internally, the operators perform subtasks that can be dynamically scheduled and executed in parallel. These subtasks do the access operations on the graph topology or the evaluation of predicates on vertex and edge attributes.

The runtime provides several features to support these operators. A priority queue can help an operator to execute subtasks speculatively. This speculative execution allows an asynchronous execution and an increased parallelism.

Additionally, the runtime monitors the execution of the graph algorithm. While monitoring, it gathers statistical data about the graph and the plan execution. These data are used by the adaptivity logic that is able to alter the execution plan and its operators. Potential altering starts with minor changes like the degree of parallelism and ends with major changes like the use of an alternative execution strategy.

4 Summary

Graph processing workloads have a broad range of requirements and properties. Different applications have diverse types of graph data, algorithms, and data manipulation patterns. In addition, they require different features like the support for attributes and transaction support. Especially performance optimizations require therefore several considerations, because they depend not only on the algorithm, but also on the graph data and the hardware.

To offer both functionality and performance for a broad range of graph processing workloads, we propose a GPE inside a DBMS that provides a high-level abstraction, capabilities for dynamic graphs, and performance optimization. The GPE provides support for static and dynamic optimizations through the front end for static optimizations and the back end for optimizations at execution time.

References

[1] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. *Scientific Programming 21*, 3-4 (2013), pp. 137–148.

[2] Hauck, M., Paradies, M., Fröning, H., Lehner, W., and Rauhe, H. Highspeed Graph Processing Exploiting Main-Memory Column Stores. In *Euro-Par 2015: Parallel Processing Workshops* (2015), Springer, pp. 503–514.

[3] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proc. ASPLOS'12* (2012), pp. 349–362.

[4] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 211–222.

[5] Shun, J., and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, pp. 135–146.

[6] Wang, G., Xie, W., Demers, A. J., and Gehrke, J. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR* (2013).

[7] Whang, J. J., Lenharth, A., Dhillon, I. S., and Pingali, K. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 438–450.