TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science
Database Systems Group

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
DOKTORINGENIEUR (DR.–ING.)

Graph Processing in Main-Memory Column Stores

- Kurzfassung -

vorgelegt an der Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Marcus Paradies
geboren am 10. Februar 1987 in Ilmenau

Gutachter: **Prof. Dr.–Ing. Wolfgang Lehner**
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden, Deutschland

Prof. Dr. Thomas Neumann
Technische Universität München
Institut für Informatik – Lehrstuhl III Datenbanksysteme
85748 Garching, Deutschland

Fachreferent: **Prof. Dr. Christof Fetzer**
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Systems Engineering
01062 Dresden, Deutschland

Eingereicht im Dezember 2016

1 INTRODUCTION

Evermore, novel and traditional business applications leverage the advantages of a graph data model, such as the offered schema flexibility and an explicit representation of relationships between entities. The potential to derive new business insights from graph-shaped data through graph analytics is increasingly attracting companies from a variety of industries, ranging from web companies to traditional enterprises. As a consequence, more and more companies are confronted with the challenge of storing, manipulating, and querying potentially terabytes of graph-structured data for enterprise-critical applications. Although graph structure is already latent in most database schemata and inherently represented by foreign key relationships, managing native graph data is moving into the focus as it allows rapid application development due to the absence of an upfront defined database schema.

Existing solutions performing graph operations on business-critical data either use a combination of SQL and application logic or employ a graph management system (GMS), such as NEO4J or SPARKSEE, or distributed graph systems, such as GRAPHLAB or APACHE GIRAPH. For the first approach, relying only on SQL typically results in poor execution performance caused by the functional mismatch between a graph algebra and the relational algebra. The alternative is to process the data in a native GMS to overcome the unsuitability of the relational algebra to express complex graph queries in an RDBMS. Since the majority of these enterprise-critical applications exclusively run on relational DBMSs, employing a specialized system for storing and processing graph data, however, is typically not sensible. Besides the maintenance overhead for keeping the systems in sync, combining graph and relational operations is hard to realize as it requires expensive data transfer across system boundaries. In the following we further elaborate on the characteristics of modern data management system landscapes as can be found in most large companies nowadays.

1.1 Heterogeneous Data Management System Landscapes

In the era of *Big Data*, companies face tremendous challenges when processing data of different shape, size, and velocity. These challenges are the key drivers that led to the separation of DBMSs into isolated *data silos* and the proliferation of diverse DBMS landscapes. The heterogeneity of the DBMS landscape and its separation into data silos, however, poses the challenges of orchestrating query processing and assuring data consistency across system boundaries.

Traditionally, row-oriented, disk-based RDBMSs were designed for transactional business processing with frequent updates to the database and short-running point queries. They cannot, however, serve non-traditional workloads, such as graph processing, stream processing, and statistical operations [1]. The recent NoSQL movement is one indicator of this paradigm shift from traditional, row-oriented RDBMSs tailored to business transaction processing towards highly specialized systems for novel business applications. Incoming transactional data is usually processed by a few operational systems that rely on mature RDBMS technology to store, manipulate, and query the data. System stability, security, and reliability are of utmost importance for these systems since a database corruption or a security leak can have a tremendous negative business impact. Specialized data management systems are usually not designed to cope with these non-functional requirements, which is why they are mainly used to run on replicated data only [1].

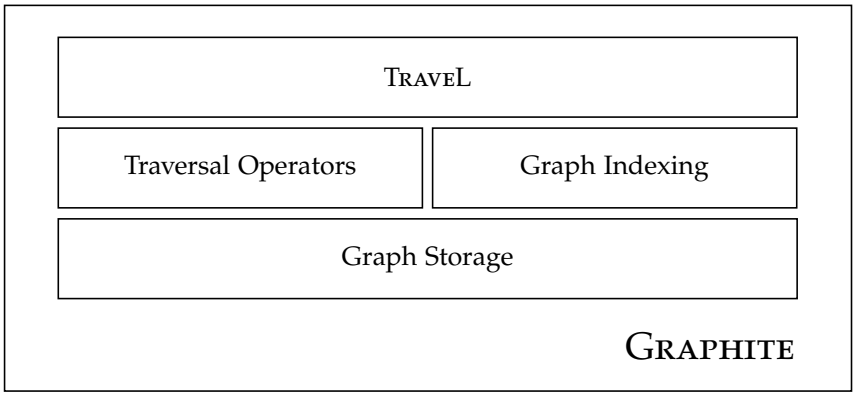


Figure 1: System architecture and major components of Graphite.

1.2 Cross-Data-Model Query Processing

A graph does not consist of a topology only but also has a rich set of attributes attached to vertices and edges. To issue queries that access data from different data models and types seamlessly, the data should be ideally stored in a single DBMS.

If one would want to issue a query than spans different data models against a diverse system landscape, one would need a large number of specialized systems and an additional orchestration layer to merge intermediate results. To cope with these issues, there are ongoing efforts to consolidate the DBMS landscape where possible and to allow online querying even on the most recent data snapshot that is not necessarily relational. One of the most promising possible solutions is the development of a *data platform* [2], which is a multi-engine DBMS accommodating native support for a large variety of data models and query processing capabilities.

1.3 Contributions

In the course of this thesis, we describe the system architecture and the core components of Graphite as part of an operational RDBMS. Graphite is a performance-oriented graph data management system allowing to seamlessly combine processing of graph data with relational data in the same system. We develop Graphite as an extension of an RDBMS that is competitive in terms of execution performance with native graph processing systems while retaining strong guarantees required by enterprise-critical applications, including transaction support, backup & recovery, and security management. Figure 1 illustrates our major contributions in the context of Graphite. We propose a columnar storage representation for graph data to leverage the already existing and mature data management and query processing infrastructure of relational database management systems. At the core of Graphite we propose an execution engine solely based on set operations and graph traversals. Our design is driven by the observation that different graph topologies expose different algorithmic requirements to the design of a graph traversal operator. We derive two graph traversal implementations targeting the most common graph topologies and demonstrate how graph-specific statistics can be leveraged to select the optimal physical traversal operator. To accelerate graph traversals, we propose two graph-specific, updateable secondary index structures to improve the performance of vertex neighborhood expansion. Finally, we introduce a domain-specific graph query language called TraveL, which offers an intuitive programming model to extend graph traversals with custom application logic at runtime.

2 GRAPH STORAGE

In this section we discuss the physical storage representation of graph data in GRAPHITE and a data reorganization technique to improve the overall memory consumption. Parts of the material have been developed together with Michael Rudolf, Radwan Deeb, and Wolfgang Lehner and have been partly published in [3].

2.1 Physical Graph Representation

GRAPHITE supports the property graph model, which has emerged as the de-facto standard data model for general purpose graph processing in enterprise environments [4]. The property graph model describes a multi-relational, directed graph by a set of vertices and a set of edges. Both, vertices and edges, can have an arbitrary number of attributes assigned as key/value pairs.

We store a graph in two physical column groups, one for the vertices and one for the edges, respectively. A column group is a vertically partitioned physical universal table, where a new attribute can be added by appending a new column to the column group. Figure 2 depicts an example representation of a small data graph. We map each vertex and edge to a single entry in the column group and each attribute to a separate column. Each vertex has a unique identifier as the only mandatory attribute. An edge is drawn from the columns V_s and V_t that represent the *source vertex* and the *target vertex* of an edge, respectively.

id	type	name	title	...
1	User	John	–	...
2	Product	–	Shining	
3	Product	–	It	...
4	Category	Horror	–	
5	Category	Literature	–	...

(a) Vertex column group.

V_s	V_t	type	rating	...
2	3	similar	–	...
2	4	belongs	–	
3	4	belongs	–	...
1	3	rated	5.0	
1	2	rated	4.0	...
4	5	category	–	

(b) Edge column group.

Figure 2: Mapping of a property graph to column groups.

READ- AND WRITE-OPTIMIZED STORAGE We divide the graph storage into a read-optimized and a write-optimized data container. Such a separation is commonly used in columnar RDBMS to allow fast read operations on the compressed read-optimized store while retaining a high data ingestion rate on the write-optimized store [5–7]. Therefore, we store a dynamic graph in GRAPHITE in two read-optimized column groups for vertices and edges, respectively, and two write-optimized column groups for vertices and edges, respectively.

We employ two levels of data compression, the first level is dictionary encoding, the second level uses lightweight compression techniques to compact reoccurring values. We apply dictionary encoding on each column and map each distinct value in the column to a fixed-length numerical value code. Consequently, each column is a composite structure of a dictionary providing mappings between values and their corresponding value codes and a *data vector* only containing the value codes instead of the actual values.

A dictionary creates a dense domain by guaranteeing that all value codes are drawn from $[1, |D|]$, where $|D|$ refers to the number of distinct values in the column. On the second compression level, we apply lightweight compression on the data vector, for example to compact reoccurring values through run-length encoding. The read-optimized column group is immutable, i.e., all data insertions and updates are

redirected to the write-optimized column group. We handle deletions through a validity vector, which indicates whether a record is visible and accessible in the specific transaction context. To guarantee fast query processing, we periodically merge the write-optimized column group into the read-optimized column group.

2.2 Graph Data Compression Techniques

Our proposed columnar graph storage has two major shortcomings: (1) through the explicit representation of `NULL` values in sparsely populated columns, the memory consumption is higher than for an equivalent normalized database schema and (2) the materialization of a complete row is more expensive since more columns have to be accessed although most of them only contain `NULL` values.

We introduce Tetris, a row reordering technique to improve the overall data compression ratio for wide and sparse vertically partitioned tables. Since `NULL` values are the most frequent value in such scenarios, Tetris compresses the table by producing long runs of `NULL` values within a column and by subsequently applying run-length compression on each column. In contrast to traditional row reordering approaches based on lexicographical sorting and heuristics for candidate column selections, Tetris relies on clustering techniques applied on a row level.

Naturally, Tetris groups records with a similar set of exposed attributes close to each other in the table, resulting in a logical partitioning of records semantically belonging to the same type. This observation can be used to implement advanced scan routines that restrict the scan range to certain groups of records in the table.

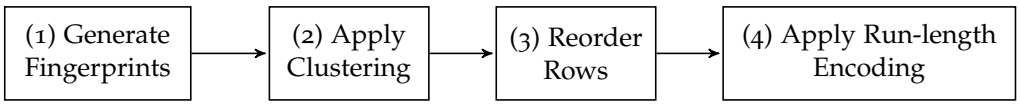


Figure 3: Tetris workflow.

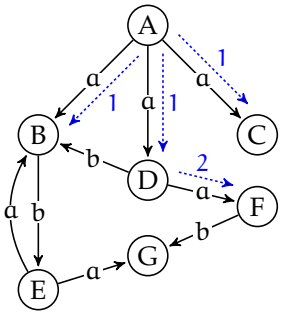
Figure 3 depicts the overall workflow of Tetris. First, we generate for all records in the table a representative fingerprint, which is used in a subsequent step to compute a normalized distance measure and to apply the k-means clustering algorithm by assigning each row to a cluster. As a result, all entities belonging to the same cluster expose a similar set of attributes and are more likely to represent entities of the same semantic type. In contrast to the basic k-means algorithm, Tetris does not require an upfront defined number of target clusters k, but instead adjusts the number of clusters automatically during the clustering phase using Bayesian statistics. In the reordering phase, we sort the table by cluster identifier and apply further sort order optimizations. Finally, we apply run-length compression on the sorted columns.

3 GRAPH TRAVERSAL OPERATORS

In this section we introduce the notion of a graph traversal operator and propose two traversal strategies on the columnar graph storage, a *level-synchronous* and *fragmented-incremental* graph traversal. Parts of the material have been developed together with Wolfgang Lehner and have been published in [8].

A graph traversal operator receives a *traversal configuration* and returns a set of discovered vertices. Figure 4 depicts a set of traversal queries and their corresponding results on the given example graph.

GRAPH TRAVERSAL OPERATOR IMPLEMENTATIONS We subdivide the processing of a traversal operation into three processing phases—a *preparation phase*, a *traversal phase*,



Traversal configuration	Result
$(\{A\}, \text{“type} = a\text{”}, 0, 1, \rightarrow)$	$\{A, B, C, D\}$
$(\{A\}, \text{“type} = a\text{”}, 1, 1, \rightarrow)$	$\{B, C, D\}$
$(\{A\}, \text{“type} = a\text{”}, 2, 2, \rightarrow)$	$\{F\}$
$(\{A\}, \text{“type} = a\text{”}, 1, \infty, \rightarrow)$	$\{B, C, D, F\}$
$(\{E\}, \text{“type} = b\text{”}, 2, 2, \leftarrow)$	$\{D\}$
$(\{A\}, \text{“type} = a \text{ OR type} = b\text{”}, 2, 2, \rightarrow)$	$\{E, F\}$

Figure 4: Example traversal queries and their corresponding results.

and a *decoding phase*. Traversal algorithms appear in many variations favoring different graph topologies and types of traversal queries. While a dense graph with a high average degree and a skewed degree distribution benefits from a skew-resilient, level-synchronous traversal algorithm, a sparse graph with a low average degree takes advantage from a more fine-granular traversal strategy. We select the optimal traversal operator implementation based on collected graph statistics and the properties of the traversal query. We propose two functionally equivalent traversal strategies, namely a *level-synchronous* (LS) traversal and a *fragmented-incremental* (FI) traversal, which target different graph properties and traversal queries.

	Graph Topology			Traversal Query	
	Degree Distribution	Average Degree	Diameter	Depth	Predicate
LS-traversal	*power-law*	*large*	*small*	*small*	*unselective*
FI-traversal	*uniform*	*small*	*large*	*large*	*selective*

Table 1: Overview of traversal strategies and their targeted graph topologies and query characteristics.

Table 1 summarizes the characteristics of both traversal strategies. The level-synchronous traversal works particularly well on graphs with a small graph diameter, a power-law degree distribution, and for short-running traversal queries with a large average outdegree and a small traversal depth. In contrast, the fragmented-incremental traversal favors a large graph diameter, a very sparse graph with a small average vertex degree, and long-running traversal queries with a large traversal depth.

3.1 Level-Synchronous Traversal

The LS-traversal operates in a level-synchronous manner and discovers vertices in a strict breadth-first ordering. It operates on an edge list represented by the columns V_s and V_t, which store source and target vertices of edges, respectively. For each traversal iteration, the LS-traversal scans the complete edge list to retrieve neighboring vertices and returns a set of vertices adjacent to the vertices of the input set. We use a set-based formalization of the graph traversal and implement the LS-traversal based on set operations.

We employ data parallelization in the LS-traversal on the *vertex level* and on the *adjacency level* by splitting the source vertex (V_s) and target vertex (V_t) columns into n equal-sized logical edge partitions, respectively. Each traversal iteration is composed of a full column scan, followed by a positional value fetch operation to retrieve adjacent vertices for a given set of vertices. The final *result collection* phase performs cycle handling and merges intermediate results. The traversal algorithm either terminates if the recursion boundary is reached or no more vertices have been discovered and

forwards its output to the materialization phase, or continues with the next traversal iteration.

If the performed number of traversal iterations or the diameter of the graph is small and all available hardware resources can be utilized, a scan-based graph traversal can provide a reasonable execution performance. In this case we can diminish the computational overhead imposed by the LS-traversal for reading edges multiple times through parallelized scan operations on the edge list. If, however, a single traversal query cannot leverage all available parallelization capabilities of the DBMS—caused by a high query workload with possibly hundreds of traversal queries running in parallel—, the LS-traversal suffers from the *work inefficiency* of the algorithm.

3.2 Fragmented-Incremental Traversal

In this section we propose an alternative traversal strategy, which reduces the number of accessed edges compared to the LS-traversal significantly. We build on the general observation that the size of the frontier set in each traversal iteration is not uniformly distributed across all traversal iterations, but instead grows until the traversal reaches a certain traversal depth—the traversal iteration where most of the vertices are discovered—and then shrinks again afterwards [9]. While for scale-free graphs with a skewed degree distribution and a small graph diameter the increase of the size of the frontier set is steeper, for very sparse graphs, such as road networks, the increase of the size of the frontier set is smoother. The FI-traversal avoids to scan the entire edge column group for each traversal iteration by consulting during each traversal iteration a light-weight secondary index structure—the transition graph index. The transition graph index provides detailed information about column fragments and potential transitions between them during the traversal. Thereby, it employs a *fragment-at-a-time* processing model and processes the traversal asynchronously.

GENERAL IDEA The FI-traversal divides the edge list represented in columns V_s and V_t into non-over-lapping, disjoint *column fragments* and executes the traversal *fragment-wise* instead of *column-wise*. A column fragment contains a subset of the edges of the graph and can be seen as a logical partition of the edge list. The FI-traversal accesses only those column fragments that are relevant for the traversal and skips all other column fragments. Based on the frontier set, the FI-traversal determines the next column fragments to read. In contrast to the LS-traversal, which operates *level-synchronously*, the FI-traversal runs *level-asynchronously* and reads in each traversal iteration only a small portion of the graph instead of the complete edge list. More specifically, the LS-traversal collects frontiers from a single traversal iteration during the scan of the complete edge list; the FI-traversal collects frontiers from multiple traversal iterations during a single read of a column fragment. To determine the set of candidate fragments, the FI-traversal leverages a secondary data structure, the *transition graph*, which stores transitions between column fragments.

Figure 5 depicts two example edge tables *with* and *without* edge clustering enabled and their corresponding transition graphs. A transition between two column fragments indicates the existence of (at least) one path of length two with one edge $e_1 \coloneqq \langle u, v\rangle \in E_{F_1}$ and one edge $e_2 \coloneqq \langle v, w\rangle \in E_{F_2}$. For example, in Figure 5 (c) there is a transition between F_2 and F_3 since there is a path $13 \rightsquigarrow 12 \rightsquigarrow 15$ with $\langle 13, 12\rangle \in E_{F_2}$ and $\langle 12, 15\rangle \in E_{F_3}$.

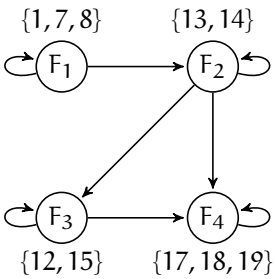
Additionally, we store a *column fragment synopsis* attached to each column fragment in the transition graph. A column fragment synopsis $S_{F_i} \coloneqq \{\, u \mid \langle u, v\rangle \in E_{F_i} \,\}$ is a concise representation of the distinct source vertices in the edge set E_{F_i}. For example, the column fragment synopsis of the column fragment F_2 in the transition graph depicted in Figure 5 (c) is the set $\{13, 14\}$.

	V_s	V_t
F_1	7	8
	1	7
	8	1
	8	13
	8	14
F_2	13	12
	14	19
	14	15
	14	13
F_3	12	15
	15	19
	15	17
	15	18
F_4	19	18
	18	17
	17	16

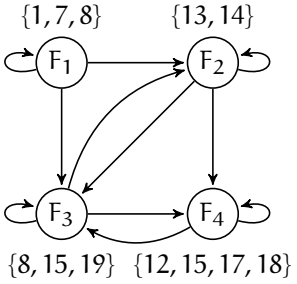
(a) Edge table *(clustered)*

	V_s	V_t
F_1	1	7
	8	1
	7	8
	8	13
F_2	13	12
	14	19
	14	15
	14	13
F_3	8	14
	15	19
	19	18
	15	18
F_4	12	15
	15	17
	18	17
	17	16

(b) Edge table *(unclustered)*



(c) Transition graph *(clustered)*



(d) Transition graph *(unclustered)*

Figure 5: Edge tables and corresponding transition graphs with column fragment size 4.

In summary, the FI-traversal outperforms the LS-traversal for graphs with a low density and short traversal queries by up to two orders of magnitude. In contrast, the LS-traversal performs significantly better than the FI-traversal, if the graph is dense or the query traverses a large fraction of the whole graph.

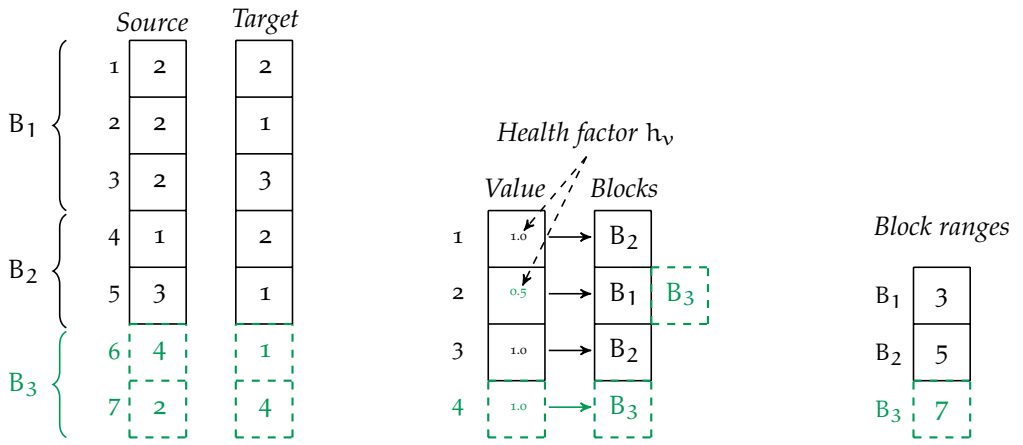
4 SECONDARY INDEX STRUCTURES FOR GRAPHS

In GRAPHITE, the primary graph storage is organized in column groups—so we consider *adjacency lists* as secondary index structures. For other native graph managements systems (GMS), however, an adjacency list might be the primary storage of the graph topology. Only a few graph indices can handle evolving graphs, which are becoming increasingly the predominant graph workload pattern. The unpredictable update performance and the large memory footprint make specialized graph index structures unattractive for a general-purpose DBMS.

Specifically, we define the major design goals of a general-purpose graph index to be *maintainability*, *applicability*, and *scalability*. Based on the design goals we devise two graph index structures, namely a *block-based* and an *adjacency-based* graph index structure. Both index structures can be used interchangeably, but expose different advantages and disadvantages in terms of construction time, index maintenance, lookup time, and memory footprint. Parts of the material have been developed together with Sebastian Rode, Matthias Hauck, and Wolfgang Lehner and have been published in [10] and [11], respectively.

4.1 Block-Based Topology Index

The block-based topology index extends the idea of an immutable CSR data structure with the ability to efficiently perform edge insertions at runtime. Instead of devising



(a) Edge column group representation.

(b) Block-based topology index.

Figure 6: Updating the block-based topology index with minimal block size = 2 (modifications colored in green).

an immutable CSR data structure, we propose a lightweight, mutable, secondary index, which operates solely on the edge column group representation. To construct a block-based topology index, we divide the clustered source vertex column into non-overlapping, contiguous blocks of potentially varying size. Conceptually, we represent a block as a tuple $\langle id, start, end \rangle$, where *id* corresponds to a unique identifier, *start* corresponds to the start position and *end* to the end position of the block in the edge column group, respectively. In a subsequent step, we store for each distinct source vertex a set of blocks.

CONSTRUCTION AND MAINTENANCE Before we construct the block-based topology index, we cluster the edge column group by source vertex. We provide an adaptive mechanism that allows handling low outdegree and high outdegree vertices—as they appear in scale-free graphs—equally well. If the outdegree of a vertex is larger than the minimal block size b_{min}, we extend the block accordingly to store all outgoing edges of a vertex in a single block. If the outdegree of a vertex is smaller than the minimal block size, we fill the block with other vertices until the minimal block size is reached.

We handle edge deletions by marking invalid edges in a lightweight invalidation data structure, which is part of the visibility and access control component of GRAPHITE. To insert an edge (cf. Figure 6), we add it to the end of the edge column group. By appending an edge to the edge column group, however, we likely break the source vertex clustering criterion. Although the block-based topology index does not rely on a strict edge clustering, it shows the best performance for an optimal clustering, as we can map each vertex in the column to exactly one block. If the column is not optimally clustered, each vertex (and its outgoing edges) can appear in multiple blocks and therefore points to a set of blocks.

We use a measure—the *health factor*—to quantify the overall quality of the index structure with respect to query performance. The health factor h_v reaches its maximum ($h_v = 1.0$), when all adjacent vertices of a vertex can be pulled from a single block. If the health factor h of the index is below a threshold τ, we consider the index as not beneficial anymore to considerably speed up neighborhood queries.

EXAMPLE INDEX REPRESENTATION Figure 6 illustrates the insertion of two edges $\langle 4, 1 \rangle$ and $\langle 2, 4 \rangle$ at the end of the edge column group and the corresponding updates to the index. The first insertion triggers the creation of a new block B_3, which is increased until the *minimal block size* is reached. While blocks residing in the static

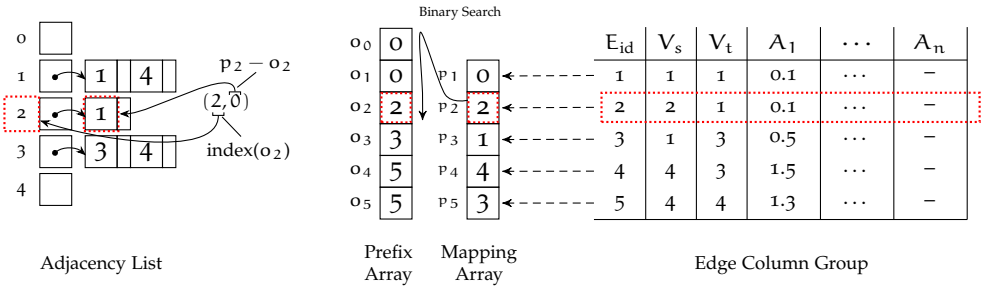


Figure 7: Mapping of rows in the edge column group to entries in the adjacency list.

fraction of the edge column group can vary in size, all blocks in the dynamic part have a fixed, but configurable size. After inserting the edges into the edge group, we update the mapping of vertices to blocks. The second case increases for a single vertex the number of blocks to read. If the source vertex column is perfectly clustered or the outgoing edges for each vertex can be fetched from a single block, we refer to the index structure as being in a good *health state*. An index lookup achieves the best possible query performance as for each vertex only a single block needs to be scanned. Frequent edge insertions, however, *pollute* the index and degrade the index health. At some point, the index health degradation is so severe that even a full-column scan outperforms an index lookup.

4.2 Adjacency-Based Topology Index

The adjacency-based topology index is a secondary index structure, which allows answering neighborhood queries directly and without the need to consult the primary copy of the data. This is in contrast to the block-based topology index, where neighborhood queries cannot be answered by index lookups only. In addition to the general requirements for the block-based index, we pose the following supplementary requirements: *support for bi-directional graph traversals*, *references to the attribute column groups*, *index mutability*, and *delta merge stability*.

The adjacency-based topology index holds internal mapping structures to allow accessing attribute values of vertices and edges from the adjacency list and to access the graph topology based on a predicate evaluation on the vertex/edge column groups. We build the core adjacency list from a projection of the edge column group to the edge ID, the source vertex, and the target vertex column. The algorithm is based on three passes: a *statistics gathering pass*, a *parallel sorting pass*, and an *parallel insertion pass*.

To combine graph with relational processing, such as filtering and aggregation on the vertex and edge column groups, we add mappings between the adjacency list and the corresponding vertex and edge column groups. GRAPHITE keeps bidirectional, light-weight, and updateable mapping tables between the adjacency list and the corresponding column groups.

Figure 7 depicts an exemplary mapping between the adjacency list and the corresponding edge column group. For example, to access the adjacency list entry for the edge with ID 2, we first retrieve the corresponding mapping array entry at position 2 (in this case the mapping value is 2). Next, we use the prefix array and perform a search to retrieve the largest element, which is lower or equal to the mapping value (in this case o_2). The index for the outer array (the source vertex) is the position o_2 in the prefix array. The index for the inner array—the index within the adjacency—can be computed as the subtraction of the entry in the prefix array from the mapping value. For the edge with ID 2, we retrieve the tuple $\langle 2, 0 \rangle$.

5 TRAVEL — A DSL FOR GRAPH ANALYSIS

To implement a domain-specific graph algorithm in the context of a complex graph application, simple graph traversals are typically not expressive enough nor do they allow customization to the user's needs. To cope with these issues, graph database vendors provide—in addition to their declarative graph query languages—procedural interfaces to write user-defined graph algorithms [12, 13]. Such imperative interfaces offer a powerful abstraction to write user-defined, domain-specific graph algorithms, but they also have major drawbacks. To the worse, writing graph algorithms in a general-purpose language prevents exploiting data- and domain-dependent optimizations at runtime and certain query optimization and rewriting techniques, such as selection push-down and leveraging intra-query parallelism cannot be applied.

We propose TraveL, a domain-specific query language for writing complex graph algorithms. In contrast to a low-level programming interface, TraveL provides high-level, graph-specific language constructs to formulate graph algorithms in a user-friendly and intuitive way while retaining an equivalent execution performance to manually optimized implementations written in C++. TraveL is statically typed and exhibits a graph abstraction and natively supports fundamental graph data types, such as vertices, edges, and paths, and operations thereon. It facilitates an imperative programming model with control flow elements and data querying and manipulation operations. The core programming concept of TraveL are *traversal hooks*, which allow extending optimized, built-in graph traversal operators by user-defined program logic. TraveL can be extended by exposing high-performance, built-in graph algorithms directly within a procedure script.

5.1 Model of Computation

A graph traversal discovers new vertices and traverses over edges in a deterministic and well-defined manner. We use an event-oriented programming model and the notion of *traversal events* to allow end users to extend the ordinary graph traversal semantics with custom logic. By ordinary traversal semantics, we refer to the traversal order of BFT and DFT, i.e., to discover vertices level-by-level (BFT) or to discover vertices recursively (DFT).

Such traversal events include the discovery of new vertices and the traversal over edges. Although it would be possible to define other, more specialized traversal events, i.e., the repeated visit of a vertex/an edge, we argue that a restricted number of event types is sufficient to compose more complex traversal events.

Each traversal event triggers the execution of a user-defined action—called *traversal hook*— that is defined for this event type. A traversal hook can produce and access volatile and persistent state, which is shared between invocations. Additionally, a traversal hook can change the semantics of the underlying graph traversal and steer the traversal during runtime.

The traversal operator calls the traversal hook for each triggered traversal event; the traversal hook can steer the traversal operation by either restricting/terminating or extending the traversal. The traversal hook stores intermediate results in the traversal state, which is shared across all traversal hook invocations. Data stored in the traversal state is immediately visible in the logically subsequent traversal hook invocation. Multiple traversal hooks can be associated with a graph traversal operator, where each traversal hook is assigned to a traversal event type. For example, the user can specify two traversal hooks reacting to the discovery of new vertices, where each traversal hook by itself might perform a different action. Traversal hooks can either share the traversal state or have exclusive state that is only visible within the specific traversal hook. In each call, the traversal hook can read its traversal state and access the graph through a common graph programming interface.

Listing 1: Summarized BOM explosion.

```
-- Preamble
CREATE TEMPORARY VERTEX ATTRIBUTE<INT> sum = 0;

-- hook definition
HOOK EDGE "H" (CONTEXT $e) {
  $h = HEAD($e);
  $t = TAIL($e);
  UPDATE $t { SET sum = $t@sum + ($e@quantity * $h@sum); }
}

-- Traversal definition
TRAVEL "BOM" (VERTEX $root) GRAPH "G" {
  UPDATE $root { SET sum = 1; }
  TRAVERSE BFS $root-[*]->(*) HOOK "H";
}
```

EXAMPLE Query support for bill-of-materials (BOM) applications is a common requirement in business environments. A BOM hierarchy is represented as an acyclic, single-rooted graph and describes the relationships between product parts. The graph contains an edge attribute `quantity`, which describes how many instances of the *subpart* are required to manufacture the *part*. To illustrate the use of traversal hooks, we revisit a fundamental operation on BOM hierarchies—summarized BOM explosion. It provides an answer to the question "What is the total quantity of each part required to build part `P1`?".

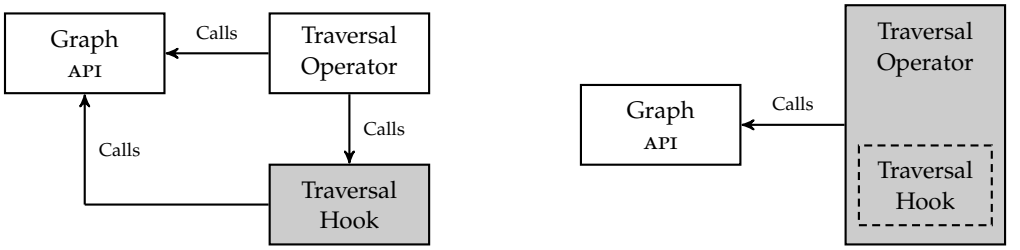
Listing 1 depicts a summarized BOM explosion expressed in TraveL. A TraveL script consists of an optional preamble, followed by a set of traversal hook definitions, and a main clause. We use a temporary vertex attribute `qnty` to collect intermediate results and initialize all values to zero, except for the root vertex `P1`. For each traversal hook invocation, we extract the head and the tail vertex from the context edge `$e` and store them in two temporary variables. We update the temporary vertex attribute `qnty` with the sum of the `qnty` of the tail vertex `$t` and the multiplication of the edge weight `quantity` and the vertex attribute `qnty` of the head vertex `$h`.

5.2 TraveL Compiler

We utilize the LLVM compiler framework to generate LLVM-IR code from a TraveL script and use the LLVM-IR code to orchestrate the execution logic, and to delegate more complex functions, such as operators and resource management tasks, to the graph backend. We identify two potential performance issues that arise in a general-purpose graph storage with support for multiple data types and a varying set of vertex/edge attributes: (1) support for multiple data types is typically implemented either using dynamic polymorphism and virtual function calls or through function overloading, and (2) to access the actual attribute data container, the high-level attribute name has to be translated into a physical memory address for each access.

In the code generation step, we determine the data type of all accessed attributes and automatically select the correct, type-dependent access function from the API. In the graph API we provide specialized functions that allow retrieving single attribute values without having to cast value types and having to resolve the attribute by name in every function invocation.

A graph traversal operator could be naturally extended by passing a function object to the traversal and calling the contained function for each event from within the operator. Instead of keeping two separate compilation units, one for the traversal and one for the traversal hook (cf. Figure 8 (a)), we create for a TraveL script a customized traversal operator during runtime in a single compilation unit (cf. Figure 8 (b)). This



(a) Traversal hook in separate compilation unit.

(b) Traversal hook compiled into traversal operator compilation unit.

Figure 8: Extending traversal operators in TraveL.

has the advantage that the JIT compiler can perform a holistic code optimization by tightly integrating the traversal hook code into the traversal operator.

5.3 Travel Query Rewriting

One of our main goals is to optimize the execution of the traversal hooks, i.e., the code path that is executed for every discovered vertex or every traversed edge. There are two main directions to enhance the performance of traversal hook invocations: (1) reducing the number of instructions performed for each traversal hook invocation and (2) batching multiple, independent traversal hook invocations.

The TraveL compiler automatically detects predicates in traversal hooks that do not depend on runtime computations and moves them out of the traversal hook function into the traversal operator. The rewriting logic of the TraveL compiler moves the predicate evaluation out of the traversal hook and rewrites the traversal configuration to use an additional edge filter restricting the traversal. Another fundamental rewriting technique merges multiple traversal hooks registered for a single traversal operator into a single traversal hook. Finally, the user is not forced to write TraveL scripts using solely traversal hooks to express graph algorithms. To cope with this issue, we search for code patterns in a TraveL script that mimic the intended functionality of traversal hooks and rewrite them into canonical traversal hooks.

6 CONCLUSION

In this thesis, we described Graphite, a performance-oriented graph data management system at the core of an RDBMS allowing to seamlessly combine graph data with relational data in the same system. For customers, this offers an interesting alternative to specialized GMS that lack many of the features demanded by enterprise applications and require expensive data replication and maintenance processes.

We proposed a relational storage representation for graph data based on *column groups* to leverage the already existing query processing infrastructure of RDBMSs. Since the representation of vertices and edges in wide column groups might lead to sparsely populated columns, effectively resulting in a higher memory consumption, we developed a light-weight compression technique called Tetris. Tetris identifies entities that expose a similar set of attributes automatically, reorders them within a column group, and finally applies RLE to compress `NULL` values in each column.

To support efficient query evaluation on large graphs, we proposed a logical graph traversal operator that can be configured to run k-hop traversals on (sub) graphs and an accompanying set of graph traversal implementations, namely the *level-synchronous* (LS) and the *fragmented-incremental* (FI) traversal. The LS-traversal relies on repetitive parallelized full-column scans on the edge column group and provides good perfor-

mance for graphs with a low diameter and a power-law degree distribution resulting in large intermediate results during the traversal. For extremely sparse graphs and graphs with a large diameter, such as road networks, we developed FI-traversal, an index-assisted graph traversal implementation. To accelerate neighborhood queries and traversals, we developed two secondary graph index structures, namely the *Block-based topology index* and the *Adjacency-based topology index*.

Finally, we developed a traversal-based programming model and an accompanying domain-specific graph query language called TRAVEL. We build TRAVEL on *traversal hooks*, which are well-defined extension points of traversal operators allowing the user to execute custom code during a traversal query. We used the LLVM compiler framework to create a specialized graph traversal operator on-the-fly during runtime as the combination of highly tuned, built-in traversal operators and user-specified code that acts on a vertex- or edge level.

REFERENCES

[1] C. Mohan. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *Proceedings of the International Conference on Extending Database Technology*, EDBT '13, pages 11–16, 2013.

[2] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, 2012.

[3] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web*, pages 403–420, 2013.

[4] Marko A. Rodriguez and Peter Neubauer. The Graph Traversal Pattern. *CoRR*, abs/1004.1001, 2010.
http://arxiv.org/abs/1004.1001 (Last accessed: December 2016).

[5] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 731–742, 2012.

[6] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.*, 6(11):1080–1091, August 2013.

[7] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL Server Column Stores. In *Proc. SIGMOD'13*, pages 1159–1168, 2013.

[8] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 29:1–29:12, 2015.

[9] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, 2012.

[10] Marcus Paradies, Michael Rudolf, Christof Bornhövd, and Wolfgang Lehner. GRATIN: Accelerating Graph Traversals in Main-Memory Column Stores. In *Proceedings of Workshop on Graph Data Management Experiences and Systems*, GRADES '14, pages 9:1–9:6, 2014.

[11] Matthias Hauck, Marcus Paradies, Holger Fröning, Wolfgang Lehner, and Hannes Rauhe. Highspeed Graph Processing Exploiting Main-Memory Column Stores. In *Proceedings of the Euro-Par 2015 International Workshops*, pages 503–514, 2015.

[12] Neo4j project website. http://neo4j.org/ (Last accessed: December 2016).

[13] Sparksee project website. http://www.sparsity-technologies.com/ (Last accessed: December 2016).