GRATIN: Accelerating Graph Traversals in Main-Memory Column Stores

Marcus Paradies[1]   Michael Rudolf[1]   Christof Bornhövd[2]   Wolfgang Lehner[1]

[1]Database Technology Group
TU Dresden, Germany
{m.paradies,michael.rudolf01}@sap.com
wolfgang.lehner@tu-dresden.de

[2]SAP Labs, LLC
Palo Alto, CA 94304, USA
christof.bornhoevd@sap.com

ABSTRACT

Native graph query and processing capabilities have become indispensable for modern business applications in enterprise-critical operations on data that is stored in relational database management systems. Traversal operations are a basic ingredient of graph algorithms and graph queries. As a consequence, they are fundamental for querying graph data in a relational database management system.

In this paper we present GRATIN, a concise secondary index structure to speedup graph traversals in main-memory column stores. Conventional approaches for graph traversals rely on repeated full column scans, making it an inefficient approach for deep traversals on very large graphs. To tackle this challenge, we devise a novel and adaptive block-based index to handle graphs efficiently. Most importantly, GRATIN is updateable in constant time and allows supporting evolving graphs with frequent updates to the graph topology.

We conducted an extensive evaluation on real-world data sets from different domains for a large variety of traversal queries. Our experiments show improvements of up to an order of magnitude compared to a scan-based traversal algorithm.

1. INTRODUCTION

Efficient graph traversals are crucial to every graph database management system and the foundation for a large variety of graph algorithms, such as finding shortest paths, detecting (strongly) connected components, and answering reachability queries. Although graph traversals have been an area of active research for more than four decades, novel algorithms for traversing graphs with billions of edges within seconds received greater attention more recently again [2, 3, 20].

More and more enterprises from various industries are starting to explore and analyze the connections between data records in traditional customer-relationship management and enterprise-resource-planning systems. For example, enterprises want to optimize a supply chain network, update an employee hierarchy or query a product batch traceability graph. Existing solutions for perform-

ing graph operations on these systems use SQL in combination with application logic to process the data. Since the majority of enterprise-critical systems exclusively run on relational DBMSs, employing a specialized system for storing and processing graph data is typically not sensible: Beside the maintenance overhead for keeping the two systems in sync, combining graph and relational operations is hard to realize as it requires data transfer across system boundaries.

Especially for performance-critical analytical applications, main-memory column stores have proven to outperform traditional disk-based, row-oriented DBMSs by multiple orders of magnitude. They are also increasingly used as a data platform for a wide range of different data models and allow combining them seamlessly in *cross-data-model* operations [10, 21]. Surprisingly, native support for storing, querying, and manipulating graphs in (main-memory) column-oriented DBMSs has received only little attention in the database community so far.

In this paper we explore one specific step in the direction of seamlessly integrating graph processing functionality into a column-oriented DBMS. By providing an implementation for efficient graph traversals as a central building block, developing more complex graph algorithms and graph query support can be facilitated. To this end, we introduce an index data structure for speeding up graph traversals in a main-memory, column-oriented DBMS. The tradeoff between memory consumption and traversal speed improvement is configurable and can therefore be adapted to the system's workload. The contributions of our work can be summarized as follows:

- We introduce our configurable block-based index data structure GRATIN for speeding up graph traversals.
- We conduct an extensive experimental evaluation for a large variety of real-world data sets and traversal queries to show the effectiveness and efficiency of our approach.

The remainder of this paper is structured as follows: in Section 2 we describe the graph traversal as the underlying building block of many graph algorithms. We present our index data structure for graph traversals in Section 3 and provide an extensive experimental evaluation in Section 4. Finally, we discuss related work in Section 5 before we conclude the paper in Section 6.

2. GRAPH TRAVERSALS

A graph traversal algorithm visits vertices of a graph in an ordered way and keeps track of discovered vertices and their level of discovery. We define a graph traversal as a tuple $\kappa = (S, c, r)$, where S is the set of start vertices, c is the collection boundary, and r is the recursion boundary. A traversal visits vertices in a breadth-first manner and outputs visited vertices discovered between level c and level r. For example, $\kappa = (\{3\}, 2, 2)$ traverses from vertex 3 and

(a) Example graph.

id	type	name	...	title
1	User	John	...	?
2	Product	?		The Shining
3	Product	?	...	The Stand
4	Category	Horror		?
5	Category	Literature	...	?

(b) Vertex table.

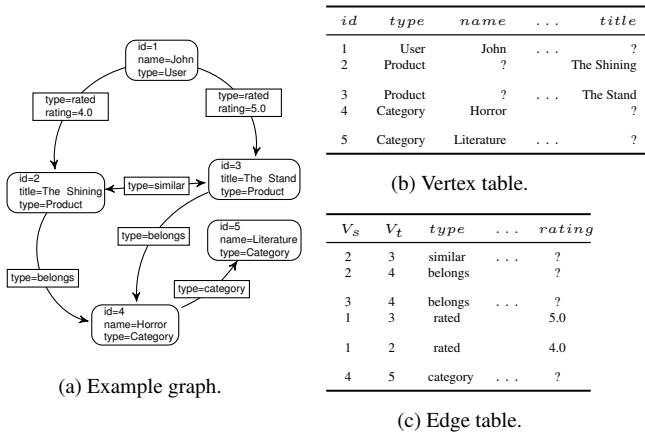| V_s	V_t	type	...	rating
2	3	similar	...	?
2	4	belongs		?
3	4	belongs	...	?
1	3	rated		5.0
1	2	rated		4.0
4	5	category	...	?

(c) Edge table.

Figure 1: Mapping of a property graph to relational tables.

collects only vertices with a distance of two from the start vertex. We expose data being stored in relational tables as a property graph. The property graph data model has emerged as the de-facto standard for general purpose graph processing in enterprise environments [16]. It represents multi-relational directed graphs where vertices and edges can have assigned an arbitrary number of attributes in a key/value fashion. Figure 1 illustrates the mapping of a property graph into two universal tables, one for the vertices (*Vertex Table*) and one for the edges (*Edge Table*), respectively. Each vertex and edge is mapped to a single database record and each attribute is mapped to a separate column. Each vertex has a unique identifier as the only mandatory attribute.

A straight-forward implementation of a graph traversal in a column store could run repeated full column scans and positional lookups to process the traversal. On modern hardware, column scans can be efficiently parallelized on thread and instruction level.

3. GRATIN

For graph traversals with a large traversal depth a scan-based approach with repeated full column scans is intolerably expensive. However, we can speed up traversals and avoid costly full column scans by constraining the scan range to only a small fraction of the entire column. A traversal accelerated by a secondary index can significantly outperform a scan-based traversal for sparse graph topologies or vertices with a small neighborhood. To that end, we propose GRATIN, a secondary index to speed up graph traversals in main-memory column stores. GRATIN is an efficient and concise block-based index structure that maps each distinct value in a column to one or multiple blocks. The blocks divide the column logically in non-overlapping and continuous column fragments. We represent a block as a tuple ⟨*id, start, end*⟩ and uniquely identify it by its *id*. GRATIN accelerates the most crucial operation in graph traversals, that is, to return all neighbors of a given vertex. Our design goals are:

- **Maintainability.** Index maintenance is the most critical operation for any index structure aiming at providing fast access to the graph topology. In the worst case, a single insertion or deletion can trigger a complete recreation of the index.
- **Integrability.** GRATIN has to be integrable into a main-memory column store. Therefore, it has to provide a concise representation and efficient positional access to the fragments of a column.
- **Simplicity.** GRATIN should rely on simple data structures that provide fast access in constant time and simple synchronization across multiple threads.
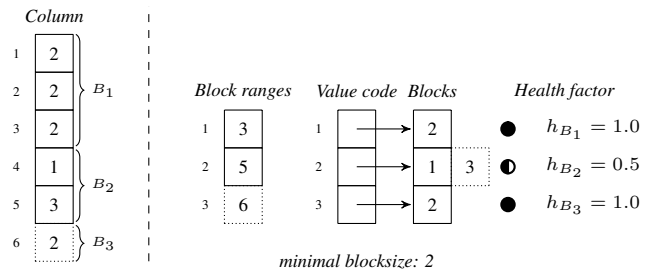

Figure 2: Updating the index after a new value has been inserted.

Figure 2 depicts a column and the corresponding GRATIN with a minimal blocksize 2. Initially, we assume that the column is clustered by value. A GRATIN consists of two main data structures, a *block index* (bi) and a *block range vector* (brv). The block index maps each value code to a set of blocks. If the column is perfectly clustered, each value code points to exactly one block. The block range vector is a concise data structure for representing block ranges. Since the blocks are continuous and non-overlapping, we store only the end position of the block. The block boundaries of a block b can be directly derived from $[\text{brv}[b-1]+1, \text{brv}[b]]$. For example, block 2 spans the range $[4, 5]$ in the column. Block IDs are only represented implicitly in the block range vector and therefore do not consume any memory.

A request to GRATIN extracts for a given value the corresponding blocks from the block index and materializes the block ranges by using the block range vector. The output is a set of blocks, where each block is described by its ID, its start, and its end position. Algorithm 1 details the lookup routine.

Algorithm 1: Index lookup in GRATIN

   Input : Set of values S.
   Output: Set of blocks C with range information.
1 begin
2   **forall the** $s_i \in S$ **do**
3 $B \leftarrow \text{bi}[s_i]$;
4   **forall the** $k \in B$ **do**
5 $C \leftarrow C \cup \langle k, \text{brv}[k-1]+1, \text{brv}[k] \rangle$;

3.1 Index Construction

We construct GRATIN by first scanning the complete column and storing temporary information about block boundaries and values in these blocks. This initial step can be efficiently implemented in a column store with a single parallelized full column scan operation.

GRATIN does not rely on a fixed block size, but instead increases the block size such that all occurrences of a value can be retrieved from a single block. This is due to the fact that many graphs exhibit a power-law distribution of the vertex outdegree. We provide an adaptive mechanism that allows handling low outdegree vertices and high outdegree vertices equally well. Initially, the construction algorithm receives a minimal block size k and creates blocks of at least that size. If the outdegree of a vertex is larger than k, the construction algorithm increases the block size accordingly.

3.2 Update Handling

In many realistic scenarios graphs change over time and demand mechanisms to efficiently support insertions, updates, and deletions of edges and vertices, respectively. Thus, an index structure that aims at accelerating graph traversals on evolving graphs also has to provide efficient update operations. GRATIN has been carefully designed to handle both static graphs and evolving graphs with frequent changes to the graph structure.

For most graph applications, in-place updates of the graph topology are relatively rare events [12] while insertions and deletions can occur frequently. We handle deletions through the visibility control layer of the DBMS on top of the index structure. Deleted records are marked as invalid and are not visible anymore for incoming queries. Although in-place updates are rare, we cannot exclude them from our study. In GRATIN we process an in-place update as a deletion followed by an insertion.

When a new edge or vertex is to be inserted, we append it to the end of the table. Append operations are a common strategy to allow column stores to preserve the compression of the static fraction of the column while still providing acceptable performance for insertions. However, appending a value can violate the edge clustering criterion. Edge clustering groups edges by source vertex and increases the spatial locality for neighborhood queries. Although GRATIN does not rely on edge clustering, it shows the best performance for an optimal or nearly optimal clustering. In a perfectly clustered column we map each value code to exactly one block. If the column is not perfectly clustered, we map each value code to at most $|B|$ blocks, where B denotes the set of blocks.

Figure 2 illustrates the insertion of a new value and the subsequent updates to the index structure. An update consists of two operations: 1) If there is an unfinished block with a block size smaller than the minimal block size, we update the block range vector. Otherwise we create a new block and append its stop position to the block range vector, and 2) we append the ID of the newly created block to the block list of the value code.

Frequent insertions increase the number of blocks per value and also the number of blocks that have to be scanned for a single value. Therefore, we use a measure to quantify the quality of the index structure with respect to query performance. For each entry i in the block index vector we define the so-called health factor as $h_i = \frac{1}{|B_i|}$, where B_i refers to the set of blocks for value i. The health factor h_i reaches its maximum ($h_i = 1.0$) if all occurrences of a value code can be fetched from a single block. For sets of blocks, the health factor decreases inversely proportional to the size of the block set. The health factor of the index structure is then defined as follows.

$$ h = \frac{1}{|V|} \sum_{i=1}^{|V|} h_i \tag{1} $$

If the health factor h of the index is below a threshold τ, we consider the index structure as not suitable anymore to considerably speed up graph traversals. In this case, a reorganisation task creates a new edge clustering and rebuilds the index. To rebuild the edge clustering, the old index structure can be leveraged.

3.3 Graph Traversals with GRATIN

GRATIN can be used to accelerate the scan-based traversal as described in Section 2. It replaces the full column scans with an index scan followed by a set of block scans. We sketch the revised level-synchronous traversal accelerated by GRATIN in Algorithm 2. Instead of using a parallelized full column scan, we consult GRATIN with a set of vertices and retrieve a set of candidate blocks. Next, we scan each block and output the hit positions into P. Since the block sizes are usually quite small, a parallelization of a single block scan operation is typically not sensible. Instead, we parallelize across the calls of the block scan routine and run multiple block scans in parallel.

4. EXPERIMENTAL EVALUATION

We evaluate GRATIN on three different data sets for a variety of traversal queries. Specifically, we report on the execution time of

---

Algorithm 2: Graph traversal with GRATIN

Input : Traversal configuration $\kappa = (S, c, r)$.
Output: Set of discovered vertices R.

```
1  begin
2      if c = 0 then
3          R ← S;  // Add start vertices to result
4      p ← 1; D_w ← S;
5      while p ≤ r do
6          if D_w = ∅ then
7              return;  // No more vertices to discover
8          P ← ∅;
9          B ← ∅;  // Set of blocks
10         V_s.lookup(D_w, B);
11         forall the b_i ∈ B do
12             V_s.scanBlock(b_i.start, b_i.end, D_w, P);
13         D_w ← ∅;  // Reset working vertex set
14         V_t.materialize(P, D_w);  // Materialize vertices from P
15         if p ≥ c then
16             R ← R ∪ D_w;  // Add vertices from D_w to result R
17         p ← p + 1;
18     return R;
```

---

traversal queries for static and dynamic graphs, the time to construct GRATIN, and its memory footprint.

Experimental Setup

We have implemented GRATIN as a prototype in the context of the in-memory column-oriented SAP HANA database. All values in the vertex and edge tables are dictionary-encoded, allowing the traversal algorithms to operate directly on the value codes. Deletions only invalidate records, which are then removed during periodic reorganization processes. Newly inserted values are always appended to the end of the corresponding columns. In-place updates are implemented as a deletion followed by an insertion.

Initially, we loaded the data sets into the corresponding vertex and edge tables. We ran all experiments on a single server machine operating SUSE Linux Enterprise Server 11 (64 bit) with Intel Xeon X5650 running at 2.67 GHz, 12 hardware threads, 32 kB L1d cache, 32 kB L1i cache, 256 kB L2 cache, 12 MB L3 cache shared, and 48 GB RAM. To evaluate GRATIN on a wide range of different graph topologies, we selected three real-world graph data sets from the domains co-purchasing networks (AM), citation networks (PA), and road networks (CR). For each data set, we report the number of vertices $|V|$, the number of edges $|E|$, the average vertex outdegree $\bar{d}_{out}$, the maximum vertex outdegree $\max(d_{out})$, the estimated graph diameter $\tilde{\delta}$, and the raw size in Table 1.

4.1 Construction Time and Index Size

We investigated the impact of the graph topology and the block size on the construction time and the memory footprint of GRATIN and report our results for all three data sets in Table 2. GRATIN is very space-efficient and consumes only up to 3% of the raw data size. The *block range vector* is a preallocated array and keeps one entry per block. Each block requires only 4 bytes to encode its ID, start,

Table 1: Evaluated data sets with their topology statistics.

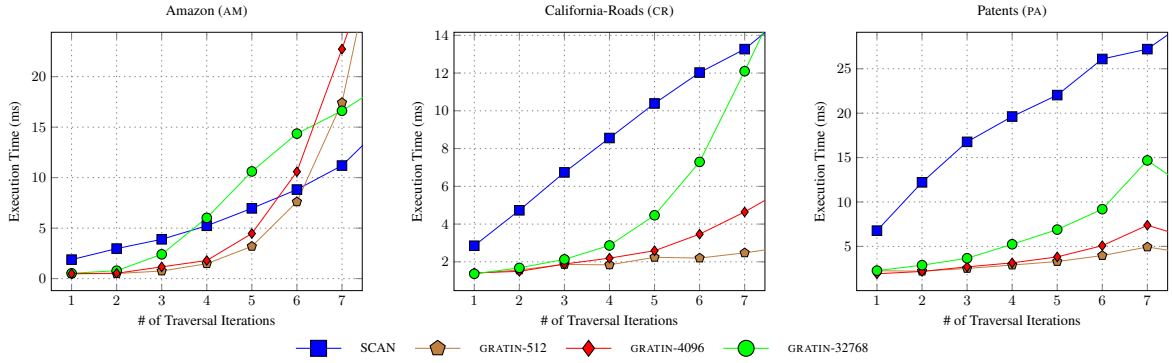| ID	$|V|$	$|E|$	$\bar{d}_{out}$	$\max(d_{out})$	$\tilde{\delta}$	Raw Size (MB)
AM	0.4 M	3.3 M	16.8	2.7 K	7.7	81.2
CR	1.9 M	2.7 M	2.8	12	495.0	132.1
PA	3.7 M	16.5 M	8.7	793	9.4	346.3

Figure 3: Comparison of scan-based traversal and GRATIN-based traversal for different block sizes.
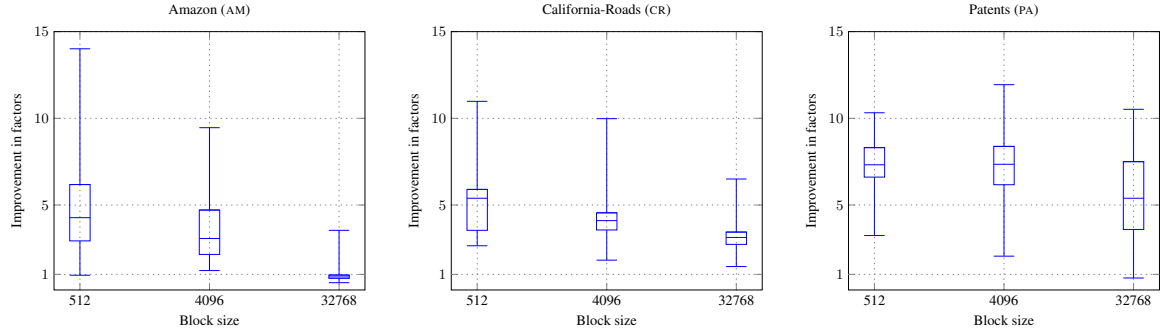

Figure 4: Improvement of GRATIN-based traversal over scan-based traversal for traversal queries over 4 hops.

and end position. In general, larger block sizes result in a smaller memory footprint of the *block range vector*. The *block index* is internally represented as an array with a size equal to the number of distinct elements in the column. For each entry, the block index maintains a list of 4 byte keys denoting the mapped block IDs.

GRATIN can be efficiently constructed with a single scan pass over the entire source vertex column. This can be trivially parallelized and allows creating the block range vector and the block index with minimal overhead. All GRATIN instances took considerably less than a second for construction, ranging from 60 ms to about 500 ms.

4.2 Traversal Performance

We compared the execution performance of a GRATIN-based traversal with a functionally equivalent scan-based traversal. For each run, we randomly selected 100 different start vertices and generated single-source vertex traversal queries with varying depth ranging from 1 to 10. When we compared GRATIN with the scan-based traversal, we used the same query configurations for both runs. For each run, we excluded the slowest and the fastest query, and report execution times averaged over 100 runs.

Static Graphs. We compare GRATIN with different block sizes {512, 4096, 32768} against a scan-based traversal and present the results in Figure 3. For all data sets, the GRATIN-based traversal clearly outperforms a scan-based traversal by up to a factor of 5. Surprisingly, GRATIN performs worse than a scan-based traversal

Table 2: Memory footprint and construction times for all data sets with block size 512.

ID	Memory Footprint (in MB)	Construction Time (in s)
AM	0.80	0.06
CR	3.97	0.30
PA	4.30	0.50

for traversals with depth greater six on data set AM. If the intermediate result is very large, a scan-based traversal can outperform an index-based traversal due to a large number of single index lookups. Moreover, smaller block sizes result in a better execution performance than larger block sizes. Especially for deep traversals with many block scans, smaller blocks reduce the amount of data to fetch from memory into the caches of the CPU.

Figure 4 depicts the improvement of the GRATIN-based traversal over a scan-based traversal. We report the improvement in multiples of the scan-based approach and evaluate traversal queries on block sizes {512, 4096, 32768}. For all data sets, the improvement factor decreases for increasing block sizes.

The execution performance of a traversal query highly depends on the selected start vertex and the size of intermediate results. We executed 1,000 queries from randomly selected start vertices and present the execution times for scan-based traversal (∘) and GRATIN-based traversal (▵) for all data sets in Figure 6. For data set CR the execution times show only a low variance, which is caused by the low variance in the vertex out-degree distribution. In contrast, the results on data sets PA and AM show an improvement by multiple factors for many traversal queries. However, some traversal queries exhibit a worse performance for GRATIN-based traversal due to large intermediate result sizes.

Dynamic Graphs. To evaluate the execution performance of GRATIN on dynamic graphs, we ran a fixed set of traversal queries. After each run, we performed a batch insert of 20,000 edges. In Figure 5, we report the slowdown factor of the execution of the set of traversal queries. We use the initial execution time as a basis and relate the following execution times to the first measurement. In addition, we report the health factor of GRATIN after each batch insert. For data set AM, the traversal queries show the largest slowdown with factor 2.2 after the insertion of 100,000 edges in total. For the
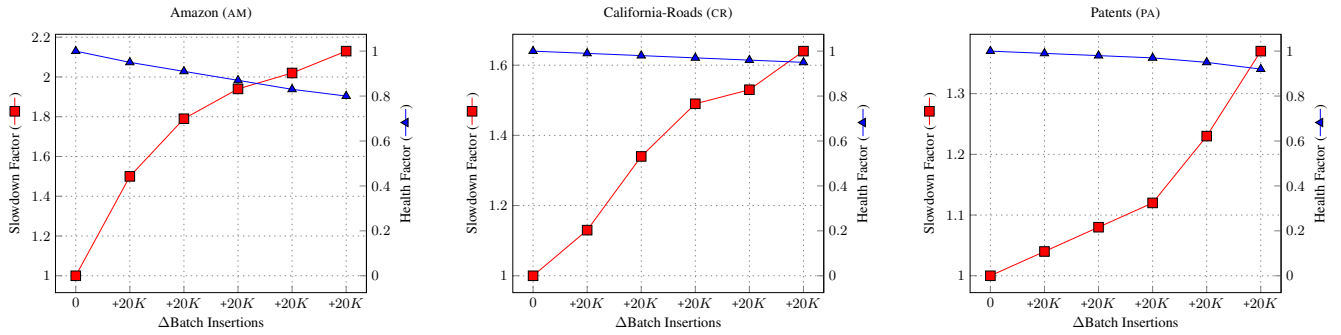
Figure 5: Evaluation of traversal performance on dynamic graphs. Slowdown factor describes the relative execution time in multiples of the execution time on a static graph.

other data sets, the slowdown factor is between 1.3 and 1.6. Even after a large number of insertions, GRATIN remains usable and still provides an acceleration over a scan-based traversal algorithm.
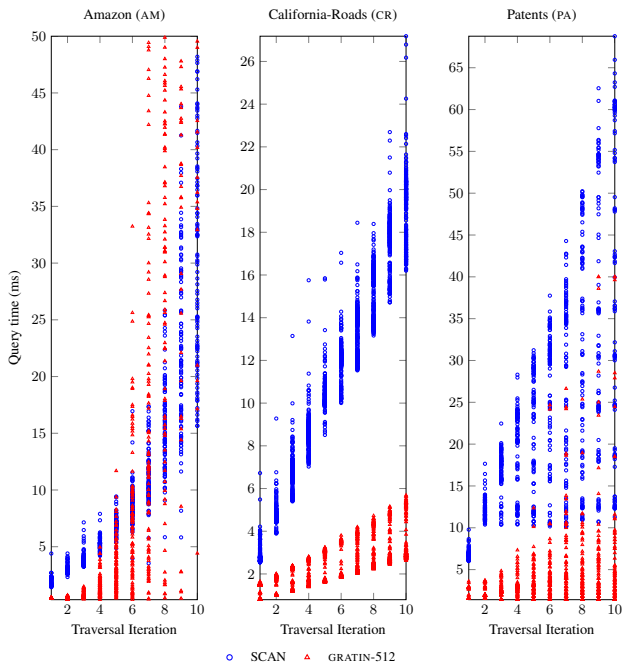

Figure 6: Query time for scan-based traversal and GRATIN-based traversal for different data sets.

5. RELATED WORK

We overview related work covering parallel and distributed breadth-first search, graph indexing, and database indexing in this section.

Graph Processing. The question of how to traverse graphs has been a topic in research for about 40 years. Christofides introduced first approaches for graph traversals in 1973 [4], which where, due to the existing hardware, only constructed serially.

The first parallel approaches where introduced by Crauser et al. [7]. They are based on a distribution of vertices over several computing nodes and initially only realized as theoretical PRAM algorithms. An implementation of a distributed *breadth-first search* relying on high parallelism and distribution over an immense number of computation nodes was done by Yoo et al. [25]. They are able to process about 30 billion edges. But as their algorithm is based on splitting an adjacency matrix into parts, it is not applicable to a

graph storage concept based on an edge list.

With the growth of data graphs, additional highly optimized, matrix decomposition based approaches have been presented, such as the one by Buluç and Madduri [2].

A common property of all these approaches is, that they only work on static data and compute the optimal data split for a fixed number of computation nodes once. They do not provide any update support and, combined with the underlying storage structure, are not applicable to our problem statement.

The update supporting approach proposed by Prabhakaran et al. [14] is also based on a graph specific storage structure, which cannot be established in our case.

So all existent, efficient, and parallel traversal approaches, even though possibly supporting updates, rely on a specialized storage structure, which does not fit our problem statement. Similar to our implementation in the context of SAP HANA, all these approaches rely on a level-synchronous traversal, which builds the resulting traversal tree level by level.

Index Structures. To support graph access and improve the performance of traversal queries, there are several different graph indexing approaches. Zhao et al. [28] introduced an index structure that is based on simple graph structures and was improved by them a few years later [27]. They increase the performance of traversals on spatial road networks by reducing the number of accesses to external storage. Even though our data is held in memory, this approach may be applicable to our problem statement as well. Although I/O performance is not the fundamental problem of in-memory algorithms, we can adapt the basic principle of regrouping data.

Corneil et al. [6] apply an ordering to the vertices based on the idea by Rose et al. [18]. They try to determine an ordering based on the detection level of the vertices and create a path-based index structure, which may consume too much memory to be applicable. A similar idea is to use an index on reachability as proposed by Trißl et al. [22]. Their index structure is based on pre- and post-ordering values, which are determined during an initial traversal process that calculates the transitive closure. Besides the fact that their index is not capable of storing distances, which is an important implicit part of our problem statement, the only guaranteed update time is a complete recreation of the index.

A different way of indexing is introduced by Yan et al. [24] and Zhao et al. [29]. In contrast to existing path-based methods they try to identify frequent substructures. An extension proposed by Zhang et al. [26] applies a similar idea to frequent subtrees, which inherit more structural information.

All those data structures may be invalidated by insertion or deletion, which may require a complete index recreation. As a direct consequence, their approach cannot guarantee an acceptable up-

per boundary for inserts. Also, further performance improvements, which where done by Wang et al. [23] and use edit distances between similar substructures, cannot solve the problem of a too high boundary for manipulation time.

As established by Sakr et al. [19] and as explained above, the existing graph indices are not applicable to dynamic data. Therefore, most graph databases measure their performance only on static data without guaranteed update performance, as discussed by Ciglan et al. [5] and Dominguez-Sal et al. [8]. This contradicts our goal to be capable of handling dynamic data and provide strict upper boundaries for manipulation time.

Next to a distinct indexing of graph structures, there is the possibility of using a data based indexing. Most relational databases that do not reside in memory rely on high performance index structures to improve data access. First thoughts about the application of existing index structures to data residing in memory were published by Lehman et al. [11]. They adapted B- and APL-Trees to memory, but did not consider applying a column-oriented instead of a row-oriented data layout as used by SAP HANA. Rao et al. [15] improved the performance of B-Trees for data in memory by taking cache sizes into account but did not present new ideas regarding index structures, as shown by Lu et al. [13].

A first, slight structural change was introduced by Bohannon et al. [1]. They propose to modify storage structures of index trees to reduce cache misses, but do not touch the actual tree structure of the index. The cache awareness of index trees was improved until patented by Rokicki [17]. Even though existing index structures have been very well adapted to in-memory use cases, they do not fit our problem. A graph traversal usually searches for column entries belonging to a certain set, so the use of a classical tree index would result in a single lookup for at least every source vertex and would thus be rather inefficient.

A more suitable approach is discussed by Faust et al. [9]. They propose a *Paged Index* to reduce the amount of data to be processed during column scans in main-memory column stores. The basic idea of splitting the whole column into smaller parts is also persued in this paper. But while their index structure consists of a consecutive bit-vector that stores the relevant pages for certain entries and relies on clustered data, we apply the idea to fit unclustered data and typical graph traversal requests.

6. CONCLUSION

We presented GRATIN, a secondary index structure to accelerate graph traversals in main-memory column stores. GRATIN is a concise block-based index that maps each distinct value to a set of blocks. For perfectly clustered columns, GRATIN maintains a 1:1 mapping. We proposed an efficient update mechanism to support graph manipulations while keeping the index maintenance overhead low. Our extensive evaluation shows that GRATIN can speed up graph traversals by up to an order of magnitude while still being memory-efficient and simple to maintain.

7. REFERENCES

[1] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD Record*, volume 30, pages 163–174, 2001.

[2] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. SC'11*, pages 65–77, 2011.

[3] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *Proc. IPDPS'12*, pages 378–389, 2012.

[4] N. Christofides. The optimum traversal of a graph. *Omega*, 1(6):719–732, 1973.

[5] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking traversal operations over graph databases. In *Proc. ICDE Data Engineering Workshops*, pages 186–189, 2012.

[6] D. G. Corneil and R. Krueger. Simple vertex ordering characterizations for graph search. *Electronic Notes in Discrete Mathematics*, 22:445–449, 2005.

[7] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science 1998*, volume 1450 of *LNCS*, pages 722–731. 1998.

[8] D. Dominguez-Sal, N. Martínez-Bazan, V. Muntés-Mulero, P. Baleta, and J. L. Larriba-Pey. A Discussion on the Design of Graph Database Benchmarks. In *Proc. TPCTC'10*, pages 25–40, 2011.

[9] M. Faust, D. Schwalb, and J. Krueger. Fast column scans: Paged indices for in-memory column stores. In *Proc. IMDM'13*, 2013.

[10] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL Server Column Stores. In *Proc. SIGMOD'13*, pages 1159–1168, 2013.

[11] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. *Proc. VLDB*, pages 294–303, 1986.

[12] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proc. KDD'05*, pages 177–187, 2005.

[13] H. Lu, Y. Y. Ng, and Z. Tian. T-tree or B-tree: Main memory database index structure revisited. In *Proc. ADC'10*, pages 65–73, 2000.

[14] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the USENIX Annual Technical Conference*, volume 12, pages 41–52, 2012.

[15] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD'00*, pages 475–486, 2000.

[16] M. A. Rodriguez and P. Neubauer. Constructions from Dots and Lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

[17] T. G. Rokicki. Main memory bank indexing scheme that optimizes consecutive page hits by linking main memory bank address organization to cache memory address organization. US Patent 6,070,227, 2000.

[18] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[19] S. Sakr and G. Al-Naymat. Graph indexing and querying: a review. *International Journal of Web Information Systems*, 6(2):101–120, 2010.

[20] H. Shang and M. Kitsuregawa. Efficient Breadth-First Search on Large Graphs with Skewed Degree Distributions. In *Proc. EDBT'13*, pages 311–322, 2013.

[21] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proc. SIGMOD'12*, pages 731–742, 2012.

[22] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. SIGMOD'07*, pages 845–856, 2007.

[23] X. Wang, X. Ding, A. K. Tung, S. Ying, and H. Jin. An efficient graph indexing method. In *Proc. ICDE'12*, pages 210–221.

[24] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. SIGMOD'04*, pages 335–346, 2004.

[25] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proc. SC'05*, pages 25–44, 2005.

[26] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *Proc. ICDE'07*, pages 966–975, 2007.

[27] J. L. Zhao and H. K. Cheng. Graph indexing for spatial data traversal in road map databases. *Computers & Operations Research*, 28(3):223–241, 2001.

[28] J. L. Zhao and A. Zaki. Spatial data traversal in road map databases: A graph indexing approach. In *Proc. CIKM'94*, pages 355–362, 1994.

[29] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. In *Proc. VLDB'07*, pages 938–949, 2007.