G-Store: A Storage Manager for Graph Data

Robin Steinhaus Dan Olteanu Tim Furche
Oxford University Computing Laboratory, Oxford, OX1 3QD, UK
{robin.steinhaus,dan.olteanu,tim.furche}@comlab.ox.ac.uk

ABSTRACT

Graph data is ubiquitous: Social networks, Semantic Web, pointer analysis in software engineering, and biological and chemical networks all rely on a graph representation of data. This paper makes the case for a native storage layer for graph data, rather than relying on relational or columnar stores. We propose a lightweight storage manager for graph data called G-Store. It exploits the structure of the graph for placement of data in pages that is optimized for a wide range of access patterns found in graph queries. Our placement approach partitions the data into pages using a multilevel partitioning algorithm and arranges the pages on disk to minimize the distance on disk between adjacent vertices.

Initial experiments show that G-Store can outperform existing graph database solutions by orders of magnitude. We believe that these results justify a promising avenue of research into storage-aware graph databases. We discuss some of these research directions.

1. INTRODUCTION

Have you ever wondered what the shortest co-author path to your scientific hero is? Or, what would be a minimal chain of people to introduce you to the head of department at that university you always dreamed of going to. Biologists want to know which genes affect the development of your arm, software engineers want to analyse dependencies between code fragments or whole business processes.

All these problems operate on directed graphs of tens of millions of vertices and edges, and need to traverse paths of arbitrary length, e.g., the citation edges, the part-of hierarchy of the human anatomy, or the dependency graph of a piece of software.

The importance of graph data has led to a revived interest in graph databases, with particular focus on (1) graph data models and query languages [3], (2) fast reachability indices [38, 7, 43], in particular for integration in relational or columnar databases, and (3) algorithms and indices for specific graph queries such as shortest path [40], subgraph isomorphism [15], and frequent subgraph discovery [21, 44]. Current graph databases that come with their own storage manager [17, 25, 41, 42, 24] are based either on B$^+$-trees or on column-stores, optimized for queries using only vertex properties and direct edge traversals. Neither approach is well suited for graph traversals where we need to pose conditions on the traversed path. Such traversals are common in analysis queries as in the introductory examples and often expressed by a form of regular path expressions [4, 27, 10]. The evaluation of regular path expressions over graph data is also needed for ad-hoc RDFS reasoning [27].

This paper introduces **G-Store**, a *lightweight disk-based storage manager*, that complements the existing disk-based or main-memory indices for graph data and large distributed solutions such as Google's Pregel [23]. We target ubiquitous and cheap hard disk drives for external storage of graphs where disk access, in particular random disk access, is orders of magnitude worse than memory access. To achieve good performance, G-Store maximizes sequential access to data for graph traversals by storing adjacent vertices close together on disk. Thus it can exploit locality of related vertices for graph traversal.

The contributions of this paper are as follows:

- A placement strategy for vertices and edges of any directed, labeled graph into disk pages that minimizes the time for neighborhood and graph traversal queries. It distributes the vertices over a sequence of pages on disk such that the number of adjacent vertices in different pages and the distance between pages with adjacent vertices are minimized.
- A novel heuristic, multilevel storage algorithm that approximates the above optimal placement strategy.
- A programming interface that supports vertex, neighborhood, reachability, shortest and regular path queries, depth-, breadth-, and page-first traversals.
- A lightweight storage manager (about 10k lines of C++-code) that implements the above features and is available from `http://g-store.sourceforge.net`. It is lightweight with no dependencies or needed infrastructure beyond standard system libraries to allow easy integration with existing graph analysis solutions.
- An initial experimental analysis that demonstrates that graph queries can significantly benefit from G-Store's storage layout and support for graph traversals.

G-Store illustrates the efficacy of a storage layer intimately optimized to the graph structure. We highlight several promising directions of research towards a storage-aware graph database, including cost-based optimization of queries.

1.1 Application scenarios

RDF, and ontologies in general, are a major reason behind the renewed interest in graph databases. Vertices and edges of RDF graphs are labeled with URIs or literals. There are many RDF properties as subClassOf that have a transitive semantics. Even for non-transitive properties (e.g., citations or friendship in a social network) we are often interested in whether there is a path of only such edges between two vertices. A number of RDF query languages have been proposed to express path traversals [4, 2, 27] including a W3C working draft for extending SPARQL [35]. None of the leading RDF stores [1, 25, 11, 41] addresses general path traversals. In recent work on columnar versus relational storage for RDF data [1, 36], only queries involving bounded path traversals (i.e., vertex selection and queries with a fixed number of joins) have been considered. SPARQ2L [4] describes algorithms and indices for path traversals, but relies on Berkeley DB's B^+-trees and is limited to fairly small graphs. G-Store is no full-fledged RDF database, but it is designed to support a wide range of path traversals and outperforms existing solutions for complex analysis queries.

Social networks have become a linchpin for most future consumer-oriented IT systems. Analyzing social networks involves reachability, shortest path and similar graph algorithms [39, 37] that can be implemented efficiently on top of G-Store. Previous approaches have focused on parallel systems and on replicating partitions of the graph data [30].

In **business process modeling (BPM)**, directed graphs of activities, events and decision points are used to describe the work flow in a process such as a loan application. In [5] a query language for BPM graphs allowing both fixed graph pattern and reachability queries is proposed, but no system aspects are discussed. For analyzing and verification of business process models expressive graph traversals corresponding to regular path queries are needed [14], e.g., "In every process execution (path) from an *implemented component* activity to an *integrate component* activity there must be a *perform test component* activity." With ever growing process models [14] and thousands of such models managed by a single company [5], efficient support by the storage layer for the employed graph traversal queries, as offered by G-Store, becomes essential. Graph traversal queries are also used in other areas of software engineering, e.g., for pointer analysis [8] or analyzing object graphs [5].

Biological and chemical networks are growing rapidly and contain "complex interconnected structures" [13]. Analyzing such networks is a complex and computationally extensive tasks beyond the abilities of G-Store. However, this analysis involves expressive, conditional path traversals, e.g., in genomics [13], for analyzing protein interaction networks [9] or molecular pathways [26]. They are also needed for querying biological ontologies such as the GeneOntology [31]. G-Store complements specialized analysis algorithms with fast evaluation of conditional path traversals.

2. G-STORE ARCHITECTURE

The basic architecture of G-Store has two layers: the data access layer and the storage layout layer, cf. Fig. 1. The data access layer consists of the page layout API, the buffer manager, and a library of primitive graph access patterns. The layout API consists of iterators for vertices and edges in a page and for pages on disk, as well as methods to access

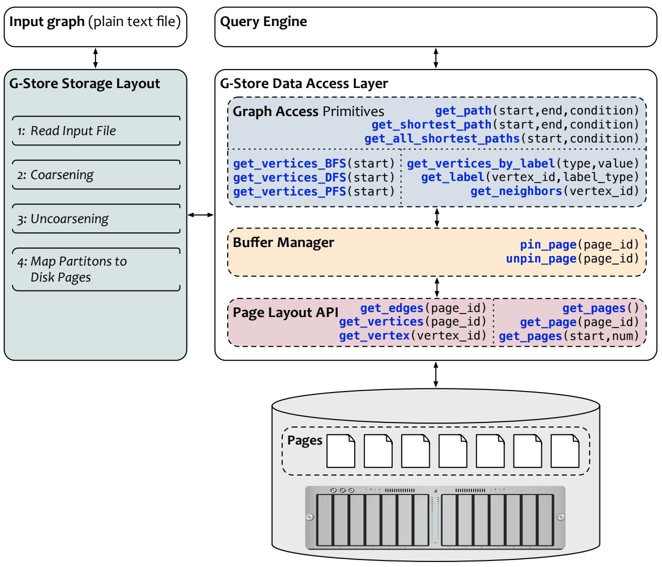


Figure 1: G-Store architecture

vertices or pages based on page and vertex identifiers.

Page layout. For each vertex, its properties and two sorted lists are stored in each page. The lists hold adjacent vertices from the same page ("internal edges") and from different pages ("external edges") respectively. Internal and external edges are represented as offsets within a page and by global vertex identifiers respectively, where such identifiers are made up of page identifiers and offsets. Every page consists of a fixed size region, followed by a variable size data region, followed by a header region. Free space accumulates between the latter two regions. The fixed size region stores information on the contents of a page. The data region stores the properties and edges of vertices.

The **buffer manager** is standard: All page requests go through the buffer manager, which allocates a pool of a fixed number of pages and uses a LRU algorithm for page eviction.

The **library of access patterns** is specific to graph data. Some of the basic methods are to retrieve vertices based on vertex identifiers, values for particular vertex properties, and neighbor vertices. A further class of methods is provided for retrieving shortest or any paths between two vertices such that the vertices along the paths follow conditions given in form of regular expressions, where the alphabet symbols are values of vertex properties. In addition, there are methods to traverse the graph, perform local computations at each vertex, and possibly collect vertices that satisfy given conditions. Graph traversal can be performed depth-first, breadth-first, or page-first. Page-first is a novel traversal order that explores as many vertices as possible in the buffered pages before loading new pages into the buffer.

To test the usability of the G-Store API, we implemented a query engine prototype for graph queries on top of G-Store.

The storage layout layer reads in chunks of graph data and writes them out to disk pages. At its core lies a multilevel storage algorithm that places vertices and edges into pages such that graph traversals are optimized.

3. STORAGE LAYOUT ALGORITHM

Accessing data stored on a hard disk drive takes orders of magnitude longer than accessing data stored in main mem-

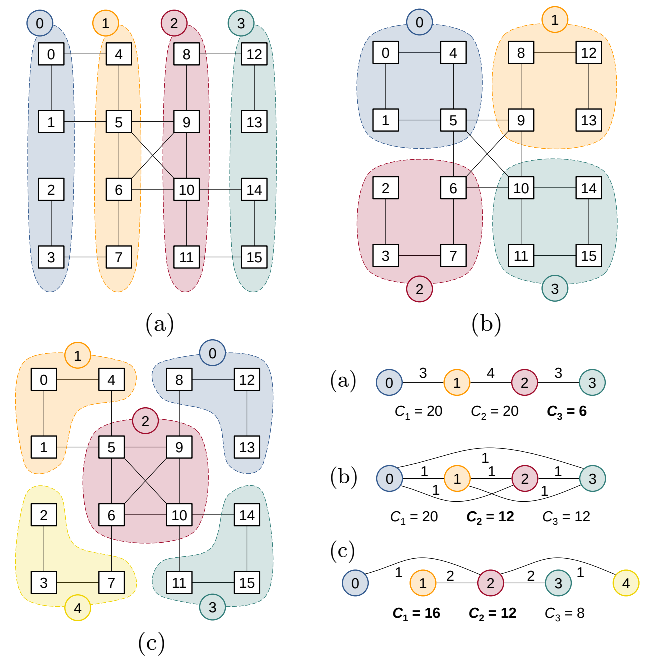

(a)
(b)
(c)
$C_1 = 20$ $C_2 = 20$ $C_3 = 6$
$C_1 = 20$ $C_2 = 12$ $C_3 = 12$
$C_1 = 16$ $C_2 = 12$ $C_3 = 8$

Figure 2: Placements and costs (|page| = 4, |vertex| = 1)

ory. The primary purpose of a storage manager for graph data is the strategic placement of data on disk so that the time needed for graph-specific access patterns is minimized. G-Store's storage algorithm has been designed to store vertices as close as possible to their edges and their neighbor vertices, that is, in the same page or in pages adjacent by their disk addresses.

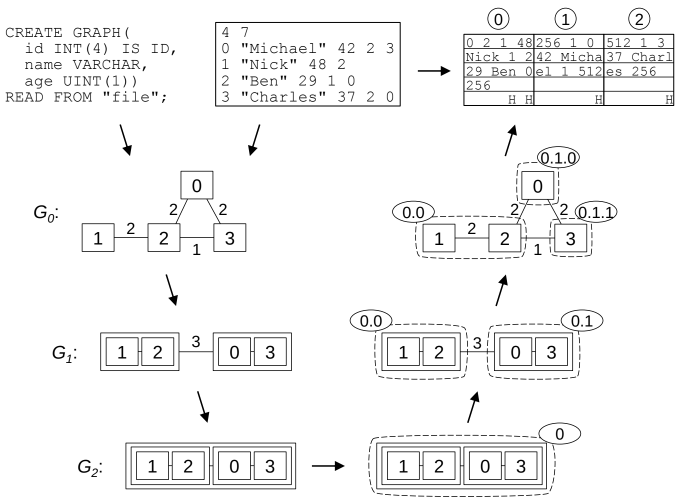
3.1 Optimal Vertex Placement

It is usually not possible to place each vertex together with its neighbors in the same page or in adjacent pages. An optimal placement is then one in which as many vertices as possible are stored as close as possible on disk. For our purpose, finding an optimal placement entails both partitioning a graph into pages and ordering such pages on disk.

Given a directed graph $G = (V, E)$, the page ordering problem is a variation of the *minimum linear arrangement problem* [16], which finds a bijective function $\phi : V \to [0..|V|)$ that minimizes $\sum_{(v,u)\in E} |\phi(v) - \phi(u)|$. In our setting, ϕ is not necessarily injective, since several vertices can be stored in the same page.

The graph partitioning problem is to divide V into k disjoint subsets, called partitions, such that no partition contains more than $|V|/k$ vertices and the number of adjacent vertices in different partitions is minimal. In the weighted variant of this problem used in our setting, each vertex is weighted and each partition may not exceed a given weight threshold, but parameter k is not known a priori.

Both problems are NP-hard [12]. Several heuristic algorithms have been proposed that achieve near optimal solutions for the linear arrangement problem, e.g., spectral sequencing [18], multilevel-based algorithms [20, 32], divide-and-conquer algorithms [6], and simulated annealing [29, 28]. The graph partitioning problem has traditionally been approached with recursive bisection. In the mid-90s, Karypis and Kumar [19] proposed a multilevel algorithm that achieved a k-way partitioning in one run. A survey of graph partitioning problems and heuristic algorithms can be found in [34]. Unfortunately, such partitioning algorithms have been designed for a relatively small constant number of partitions, which make them impracticable in our setting.



```
CREATE GRAPH(
  id INT(4) IS ID,
  name VARCHAR,
  age UINT(1))
READ FROM "file";
```

```
4 7
0 "Michael" 42 2 3
1 "Nick" 48 2
2 "Ben" 29 1 0
3 "Charles" 37 2 0
```

0	1	2
0 2 1 48	256 1 0	512 1 3
Nick 1 2	42 Micha	37 Charl
29 Ben 0	el 1 512	es 256
256		
H H	H	H

Figure 3: Multilevel storage layout algorithm

We define an optimal placement based on concomitant minimization of three distinct costs:

(1) C_1 is the sum of the differences between the page indexes of every pair of adjacent vertices.

(2) C_2 is the number of edges across pages.

(3) C_3 is the number of pairs of linked pages.

Minimizing the cost C_1 captures the page ordering problem, whereas minimizing the costs C_2 and C_3 captures distinct aspects of the partitioning problem. Fig. 2 shows three different placements of the same graph to disk pages. The graph is undirected and every edge may be viewed as two symmetric directed edges. The page arrangements of each of the three placements are represented schematically in the bottom right of the figure together with the three costs. For instance, under placement (c) we obtain five pages numbered 0 to 4. The weights of the edges between pages represent the number of edges between vertices placed in them.

Fig. 2 also shows that costs C_2 and C_3, although pertaining to the same overall graph partitioning problem, are distinct: When going from placement (a) to either (b) or (c), cost C_2 decreases while cost C_3 increases.

3.2 Multilevel Storage

G-Store approaches the placement problem through a multilevel storage algorithm that yields for any input graph an approximate solution to the following optimization problem:

$$\min \ [\alpha\, C_1 + \beta\, C_2 + \gamma\, C_3],$$

where α, β, and γ are parameters used to control the influence of each minimization goal on the overall optimization.

The algorithm proceeds as follows. It first *coarsens* the graph by iteratively collapsing connected subgraphs into compound vertices. The size of a compound vertex is an aggregate of the sizes of the vertices and the number of edges in the represented subgraph. The weight of an edge between two compound vertices is the number of edges from vertices of the original graph collapsed into the first compound vertex to those collapsed into the second. The absorption strategy

			G-Store ML				RND		
			Shortest Path		Reachability		S. Path	Reachability	
			16 MB	80 MB	80 MB		80 MB	80 MB	
Query	Data	Page	Time	Time	Time	Length	Time	Time	Length
		4 kB	35.04	30.26	1.40	7,862	80.27	1.29	9,984
id 1168745	S	16 kB	21.12	12.33	1.64	9,786	51.76	2.56	19,936
→		65 kB	12.07	8.07	1.25	2,383	25.55	1.81	14,543
id 751350		4 kB	90.41	76.19	4.38	14,835	354.23	17.64	13,287
	M	16 kB	61.92	52.41	4.84	18,253	292.53	12.83	15,078
Shortest path		65 kB	142.24	30.14	4.83	9,253	208.21	8.58	16,108
has length 264		4 kB	383.49	353.98	67.28	7,766	1,614.63	1,117.89	893
	L	16 kB	421.53	241.67	49.32	2,676	1,550.66	1,065.90	4,225
		65 kB	445.52	141.48	45.30	6,097	1,550.11	925.57	6,156
		4 kB	48.29	38.07	2.27	16,107	87.15	1.30	10,000
id 1110891	S	16 kB	28.44	14.69	1.64	9,723	44.24	2.51	19,863
→		65 kB	14.30	9.38	1.52	8,477	18.94	1.76	14,542
id 704315		4 kB	111.68	105.16	13.60	26,157	435.70	57.62	24,392
	M	16 kB	72.61	65.79	6.00	17,328	355.16	23.59	25,835
Shortest path		65 kB	136.42	39.12	10.31	10,722	242.41	23.18	26,046
has length 357		4 kB	484.57	463.57	184.51	861	2,125.09	1,377.93	452
	L	16 kB	544.73	318.63	92.88	11,072	2,022.65	895.72	7,967
		65 kB	581.78	178.44	39.29	28,710	2,022.45	781.57	8,295

(a) Shortest path and reachability

			G-Store ML		RND
Query	Data	Page	16 MB	80 MB	80 MB
Start at		4 kB	118.75	100.84	205.76
id 1430793	S	16 kB	85.07	59.40	128.28
		65 kB	60.53	42.60	70.15
Finds 1,957,027		4 kB	240.82	220.34	848.56
vertices	M	16 kB	171.64	131.70	695.62
		65 kB	360.25	90.13	503.45
Longest		4 kB	920.57	844.51	3,868.39
shortest path	L	16 kB	961.24	596.89	3,905.88
has length 717		65 kB	1,077.71	331.66	3,913.28
Start at		4 kB	113.48	97.71	194.26
id 1331368	S	16 kB	81.21	55.09	118.72
		65 kB	56.18	37.87	66.90
Finds 1,957,027		4 kB	241.35	221.99	822.39
vertices	M	16 kB	177.65	134.13	670.63
		65 kB	310.40	88.32	478.54
Longest		4 kB	905.94	831.35	3,796.16
shortest path	L	16 kB	935.93	587.89	3,886.54
has length 759		65 kB	1,051.66	324.08	3,937.26

(b) Shortest path to each reachable vertex

Figure 4: Comparison of execution time (in seconds) versus random page placement

is a variant of *heavy edge matching* [19], a greedy heuristic commonly used in multilevel partitioning algorithms: Heavy edge matching visits the vertices in random order and collapses each not-yet-visited vertex with its adjacent vertex connected by a maximum weight edge. This is repeated until the number of remaining vertices is below a given threshold. For G-Store we adapt this algorithm to allow for a large number of partitions and to reduce the number of iterations for handling of high degree vertex with a large number of very low degree neighbors. We achieve this by introducing θ, a limit to the maximum size of a compound vertex, and allowing the number κ of vertices to be collapsed in one step to vary (rather than fixing it to two). During the matching algorithm κ and θ are modified as follows:

- *Initialization:* $\kappa := 2$, $\theta :=$ block size.
- *Iteration:* If the ratio of low degree vertices with high degree neighbors reaches a certain threshold, increase θ up to $32\cdot$ block size. If θ is already at that limit, also increase κ.

Coarsening continues until each connected component is completely collapsed into a single compound vertex. The goal of the coarsening phase is to create compound vertices that can later be grouped into ordered partitions with low costs C_2 and C_3. So far, we construct bottom-up an ordered tree of compound vertices, each tree level corresponding to one iteration of the coarsening algorithm. Compound vertices can be seen as blocks whose sizes may exceed the size of a disk page, and tree levels as their possible orderings.

In the next phase, called *uncoarsening*, we traverse this tree top-down level-by-level and (i) assign partition numbers to each compound vertex, (ii) possibly swap partition numbers of groups of compound vertices, and (iii) possibly move a compound vertex from one partition to another. The assignment of partition numbers follows a Dewey numbering scheme, that is, if a vertex has a Dewey number α, its children are assigned in order $\alpha.1$ to $\alpha.n$. The swapping of partition numbers is used to decrease cost C_1. Vertex moving is employed to decrease costs C_2 and C_3.

During top-down traversal, we may stop the uncoarsening at a certain compound vertex if it is small enough to fit in a page. After uncoarsening is done, a final run over the pages seeks to merge less populated pages and also map the Dewey numbers to an interval of natural numbers such that their total order is preserved. Finally, the pages can be written contiguously on disk in this order.

Fig. 3 is an illustration of the algorithm. At the top, the input graph together with the schema definition is used to create a main memory representation of the input graph or fragments thereof. The figure shows the initial graph and the graphs after two consecutive coarsening steps. The coarsening phase produces the graph G_2 consisting of a single compound vertex. In the uncoarsening phase, we assign numbers to compound vertices, and obtain three pages which are then mapped to disk.

4. INITIAL EXPERIMENTAL EVALUATION

We have performed initial experiments to confirm G-Store's potential to speed up typical graph queries. For this, we have implemented an evaluator of (1) vertex selection queries (retrieve all vertices with a given label or id), (2) graph pattern queries (conjunctive queries), (3) reachability queries, (4) regular path queries, and (5) shortest path queries (and variants thereof).

We used an undirected, connected, real-world road network graph [22] with 2 M vertices, 5.5 M edges, and a diameter of 850. We created three datasets (S, M, and L) from the graph with an increasing number of properties at the vertices, and hence increasing raw sizes: 72 MB, 177 MB, and 1.04 GB, respectively. The parameter values for G-Store's placement optimization problem were $(\alpha, \beta, \gamma) = (.125, 1, 8)$.

We used an Intel Core Duo 2.53 GHz dual core processor with 3 GB main memory and a 320GB/5400 rpm SATA hard drive. Due to lack of space, we present only a subset of our experimental findings in this paper. More information on G-Store, including experiments with regular path queries on the Yago RDF repository, is available from the project website. We report wall-clock times and memory usage. All queries output the results to a standard buffered filestream.

Comparison with random page placement. Fig. 4(a) shows the performance for reachability and shortest path queries on a page layout created with G-Store's multilevel algorithm (G-Store ML) and on a random page layout (RND). For RND, we randomly allocated vertices to consecutive pages, while keeping pages as full as possible. On the disk, RND used approx. 10% less space than the raw format due to better data compression on pages. G-Store ML used approx. 10% more space than the raw format, since most pages were not full.

An increase in page size and buffer memory yields an improvement in performance of a factor up to six. For small page sizes, an increase of the buffer pool from 16 to 80 MB only shows a small performance increase.

For both shortest path and reachability, G-Store's storage algorithm brings a speedup of roughly 20 times when compared with random storage. We also report the length of the found path for reachability queries. Even where G-Store finds a path that is significantly longer than that found by RND, it still only needs a fraction of the time, which demonstrates its effective use of the locality of related nodes.

Fig. 4(b) shows a performance increase of up to an order of magnitude when searching for shortest paths from a given vertex to each reachable vertex (also called shortest path tree queries). Such queries are common in social network analysis: e.g., the shortest path in her network to each person she needs to get involved with in her next project. In this scenario, longest shortest paths are a measure of how "central" a person is.

Comparison with Neo4j. Neo4j [24] is an open-source graph database. It provides a Java interface that lets users create and access a graph representation on disk.

Fig. 5 gives a performance comparison between G-Store and Neo4j 1.1 for four types of queries: (1) selecting a vertex with a given id, (2) using DFS to compute from a given vertex all paths of length at most 15, (3) computing the shortest path between two vertices with given ids, and (4) computing the shortest paths from a given vertex to every other vertex in the graph (shortest path tree). Except for (2), all queries are natively supported by the Neo4j API; we implemented (2) on top of Neo4j.

We used the Apache Lucene index 2.9.2, as recommended by Neo4j. This is a high-performance text search engine library written in Java. Without this index, query evaluation with Neo4j took about 30 and 90 seconds longer for each query on dataset S and M, respectively. G-Store does not use such an index and therefore needs to scan the whole graph to find vertices with given ids.

Compared to the size of the raw data files, storing the road network graph using Neo4j needed 3.5 and 5.5 times more space without and with the index, respectively.

In our experiments, Neo4j needed at least 256 MB or 512 MB of heap space to execute some graph queries, although the raw size of dataset M was 177 MB. G-Store with an 80 MB buffer pool needs an order of magnitude less time for depth first search and shortest path tree queries than Neo4j with index and 512 MB of heap memory. For shortest path queries the same holds if comparable buffer pool and heap size are used. For vertex selection queries, the presence of the Lucene index gives Neo4j an advantage over G-Store, as they can be solved by a single look-up in that index.

We have also benchmarked the graph queries used in the previous experiments on PostgreSQL 8.3.4, a row-store relational database. Except of graph queries expressible as classical conjunctive queries, all other queries require additional constructs. We have tried several variants, including SQL constructs for transitivity, PL/pgSQL, and recursive common table expressions with or without a limit on the maximum depth. We do not report timing here, since we aborted the execution of any of the shortest path queries after several hours.

			G-Store ML 65 kB		Neo4j 1.1 and Lucene Index 2.9.2		
	Query	Data	16 MB	80 MB	128 MB	256 MB	512 MB
Vertex selection	id 1643877	S	1.20	1.20	0.13	0.13	0.13
		M	4.07	4.12	0.13	0.13	0.13
	id 1298992	S	1.23	1.22	0.13	0.13	0.13
		M	4.11	4.09	0.13	0.13	0.13
	id 1615137	S	1.20	1.21	0.13	0.13	0.13
		M	4.03	4.08	0.13	0.13	0.13
DFS all paths up to length 15	id 1718445	S	15.11	13.85	139.46	130.40	130.92
		M	19.34	19.19	145.83	130.35	129.12
	id 1052301	S	92.94	88.70	773.17	748.61	752.08
		M	87.36	84.77	757.67	742.25	739.41
	id 1690532	S	15.66	16.36	155.17	148.68	156.92
		M	19.31	20.00	153.50	145.92	148.65
Shortest path	id 831121	S	15.43	11.19	98.87	65.99	38.26
	→ id 1573723	M	199.74	43.49	100.03	61.32	42.53
	id 1475007	S	13.63	9.74	134.97	76.11	45.74
	→ id 672199	M	197.88	38.89	135.74	79.68	50.26
	id 957154	S	14.98	11.05	**	157.12	78.12
	→ id 119863	M	196.05	39.46	**	147.43	83.95
Shortest path tree	id 233348	S	40.68	31.87	**	**	370.38
		M	358.36	92.70	**	**	523.45
	id 1008508	S	37.38	29.70	**	**	339.07
		M	389.94	89.96	**	**	375.82
	id 1866411	S	56.44	32.10	**	**	344.04
		M	335.37	96.76	**	**	398.03

**java.lang.OutOfMemoryError. Note: The memory figures are not directly comparable. In G-Store, the figures represent the size of the page buffer. In Neo4j, the figures represent the size of Java's heap.

Figure 5: Comparison of execution time (in seconds) versus Neo4j with index

5. CONCLUSION AND FUTURE WORK

G-Store makes the case for a storage layout aware of the structure in the input graph data. It employs a multilevel algorithm to generate a page layout for the input graph data such that adjacent vertices are stored close together on disk and the I/O cost for traversing edges is close to minimal. We verify experimentally the benefits of this approach: The time needed to answer common types of graph queries can be orders of magnitude less for G-Store compared to a compact but random layout, and even compared to a mature graph database such as Neo4j.

We see several promising avenues of research.

1. *Storage-aware query optimization.* Query optimization can benefit from a storage-aware cost model that takes into account the distribution of vertices and their neighbors over pages, such as the average amount of pages for neighborhoods of radius k, or the maximum number of pages to traverse in order to find the shortest paths from vertices with a given property value. Such statistics can be useful estimates for the average I/O cost involved in the evaluation of graph traversal primitives. The query optimizer can then choose a query plan that favors the evaluation of graph traversals with a low I/O cost.

2. *Query compilation and optimization for G-Store access primitives.* Once a high-level query plan is derived, it is useful to investigate the gains of compiling that plan into a low-level program using G-Stores graph traversal primitives and optimizing that program, e.g., by folding a sequence of simple graph traversals into a single, conditional graph traversal. In general a few conditional graph traversal operations outperform in terms of I/O cost a larger sequence of simpler graph traversals in G-Store.

3. *Secondary indices.* In some of the application domains considered, vertex property values are in the majority unique and meaningful identifiers (e.g., URIs in RDF graphs). In these cases, G-Store already provides a primary index for the data. Additional layers of secondary indices for vertex property values could benefit query evaluation, and increase

the optimization search space.

4. *Updates.* G-Store cannot support update facilities yet. Updates, in particular insertion of new edges, can affect prior optimized placement of vertices and require re-optimization. A promising approach is to draw on incremental graph partitioning methods [33] and adapt diffusion-based repartitioners that minimize the redistribution of vertices.

5. *Edge compression.* Interval lists are a staple for the compact representation of graphs and are used in many reachability indices [38]. A key challenge when using interval lists is to find an order (or labeling) of the vertices, such that the number of intervals each vertex needs in order to reference its adjacent vertices is minimized.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] R. Angles and C. Gutierrez. Querying RDF data from a graph database perspective. In A. Gómez-Pérez and J. Euzenat, eds., *The Semantic Web: Research and Applications*, *LNCS* 3532. Springer, 2005.

[3] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[4] K. Anyanwu, A. Maduko, and A. Sheth. SPARQ2L: towards support for subgraph extraction queries in RDF databases. In *WWW*, 2007.

[5] A. Awad and S. Sakr. Querying graph-based repositories of business process models. In M. Yoshikawa et al., eds., *Database Sys. for Adv. Appl.*, *LNCS* 6193. Springer, 2010.

[6] R. Bar-Yehuda, G. Even, J. Feldman, and J. Naor. Computing an optimal orientation of a balanced decomposition tree for linear arrangement problems. *J. Graph Algorithms Appl.*, 5(4), 2001.

[7] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, 2008.

[8] M. Buss, D. Brand, V. Sreedhar, and S. Edwards. Flexible pointer analysis using assign-fetch graphs. In *SAC*, 2008.

[9] B. Dost, T. Shlomi, N. Gupta, E. Ruppin, V. Bafna, and R. Sharan. Qnet: a tool for querying protein interaction networks. *J Comput Biol.*, 15(7), 2008.

[10] A. Dries and S. Nijssen. Analyzing graph databases by aggregate queries. In *MLG*, 2010.

[11] O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In T. Pellegrini et al., eds., *Networked Knowledge—Networked Media*, *SCI* 221. Springer, 2009.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[13] M. Graves, E. Bergeman, and C. Lawrence. Graph database systems. *Eng. in Med. and Bio.*, 14(6), 1995.

[14] G. Grossmann, M. Schrefl, and M. Stumptner. Modelling inter-process dependencies with high-level business process modelling languages. In *APCCM*, 2008.

[15] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. In *VLDB*, 2010.

[16] L. H. Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1), 1964.

[17] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: a federated repository for querying graph structured data from the web. In *ISWC*, 2007.

[18] M. Juvan and B. Mohar. Optimal linear labelings and eigenvalues of graphs. *Discrete Appl. Math.*, 36(2), 1992.

[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1998.

[20] Y. Koren and D. Harel. A multi-scale algorithm for the linear arrangement problem. In *WG*. Springer, 2002.

[21] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, 2001.

[22] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Technical report, University of Stanford, 2008.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[24] Neo4j, the graph database. Online only, 2010. `http://neo4j.org/`, accessed 2010/09.

[25] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *Proc. VLDB Endow.*, 1(1), 2008.

[26] N. J. Ozsoyoglu ZM, Ozsoyoglu G. Genomic pathways database and biological data management. *Anim Genet.*, 37(Suppl 1), 2006.

[27] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. In *ISWC*, 2008.

[28] J. Petit. Combining spectral sequencing and parallel simulated annealing for the minla problem. *Parallel Processing Letters*, 13, 2003.

[29] J. Petit. Experiments on the minimum linear arrangement problem. *J. Exp. Algorithmics*, 8, 2003.

[30] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. In *SIGCOMM*, 2010.

[31] L. Royer, B. Linse, T. Wächter, T. Furche, F. Bry, and M. Schroeder. Querying the Semantic Web: A Case Study. In C. Baker and K.-H. Cheung, eds., *Revolutionizing Knowledge Discovery in the Life Sciences*. Springer, 2006.

[32] I. Safro, D. Ron, and A. Brandt. Graph minimum linear arrangement by multilevel weighted edge contractions. *J. Algorithms*, 60(1), 2006.

[33] S. Schamberger and J.-M. Wierum. Graph partitioning in scientific simulations: Multilevel schemes versus space-filling curves. In *Parallel Computing Technologies*, *LNCS* 2763. Springer, 2003.

[34] K. Schloegel, G. Karypis, and V. Kumar. *Graph partitioning for high-performance scientific simulations*, Morgan Kaufmann Publishers Inc., 2003.

[35] A. Seaborne. Sparql 1.1 property paths. Working draft, W3C, 2010. `http://www.w3.org/TR/sparql11-property-paths/`.

[36] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proc. VLDB Endow.*, 1(2), 2008.

[37] R. Soussi, M.-A. Aufaure, and H. Baazaoui. Towards social network extraction using a graph DB. In *DBKDA*, 2010.

[38] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.

[39] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto. Efficient search ranking in social networks. In *CIKM*, 2007.

[40] F. Wei. TEDI: efficient shortest path query answering on graphs. In *SIGMOD*, 2010.

[41] C. Weiss and A. Bernstein. On-disk storage techniques for Semantic Web data - Are B-Trees always the optimal solution? In *SSWS*, October 2009.

[42] G. Wu, J.-Z. Li, J.-Q. Hu, and K.-H. Wang. System Π: A native RDF repository based on the hypergraph representation for RDF data model. *Journal of Computer Science and Technology*, 24, 2009.

[43] H. Yildirim, V. Chaoji, and M. Zaki. Grail: Scalable reachability index for large graphs. In *VLDB*, 2010.

[44] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching. In *VLDB*, 2010.

G-Store: Vertex Villages for Fast Graph Analysis (Demonstration)

Robin Steinhaus Dan Olteanu Tim Furche
Oxford University Computing Laboratory, Oxford, OX1 3QD, UK
{robin.steinhaus,dan.olteanu,tim.furche}@comlab.ox.ac.uk

ABSTRACT

Wikipedia is one of the most extensive artefacts of human knowledge. Through Yago this knowledge has become available for automatic analysis and exploration. It is just one example of an increasing range of large graph datasets from social networks over business process models to biological and chemical networks. To support the analysis and exploration of such graphs, expressive graph traversal queries such as regular and shortest path queries are needed.

In this demonstration, we illustrate a novel lightweight storage manager for graph data, that is optimized specifically for such traversals. We show visually along the example of Yago how it distributes vertices over pages, such that related vertices are placed close together (as neighbors in a village). We pose a few analysis queries (with surprising results) and show how the locality of related vertices is used during the evaluation of such queries in G-Store to minimize I/O (the number of visited pages). This is achieved by visualizing access and eviction of pages from the buffer manager side-by-side with the traversed vertices.

1. INTRODUCTION

Do you know how climate change relates to Aristotle? What the Battle of Issus in 333 B.C. and Silvio Berlusconi have in common? Turns out there are connections between them. How do we know? From analyzing the graph of relations extracted from Wikipedia. You could do this game for almost any (reasonably famous) entity on Wikipedia.

But to actually do this analysis, you have to be able to process complex path traversal queries on large graphs such as Yago [1], the RDF graph containing millions of entities and relations extracted from Wikipedia. Complex graph traversals where we search for (shortest) paths between nodes and possibly pose additional conditions on these paths are at the heart of many graph analysis problems from business process modeling and verification, over protein interaction networks to social network analysis.

G-Store is a *lightweight storage manager* for graph data that organizes vertices on disk in villages: related vertices are treated as neighbors in a village and stored as close together as possible, ideally in the same page. G-Store is available from `http://g-store.sourceforge.net`.

2. DEMO DESCRIPTION

In this demonstration, we illustrate how G-Store's storage layout algorithm organizes vertices from Yago and how it exploits the locality of related vertices for efficient evaluation of shortest and regular path queries. We use a significant part of Yago [1] omitting literal nodes not useful for human analysis. The resulting graph contains over 3 M entities connected by 4.5 M edges. Yago is well suited for this demonstration as it contains mostly familiar entities and relations.

The demonstration is divided in two parts: We start with **(1)**, a visual exploration of how the vertices from Yago are stored on disk by the G-Store layout algorithm. We first give a visual overview of the distribution to disk pages and then explore the detailed storage from an interesting start vertex, e.g., "Aristotle". For that start vertex, we illustrate how the neighboring vertices are stored close by, either in the same or in nearby pages.

This locality of related vertices is exploited by G-Store for efficient graph traversals. We illustrate this in part **(2)** of the demonstration for shortest and regular path queries:

What is the shortest path between "Aristotle" and "Aldous Huxley"? To evaluate this query, G-Store performs a graph traversal starting from "Aristotle". We visualize in the demonstration how during this traversal pages are loaded, buffered, and finally evicted as more and more nodes in the graph are explored. This makes the central premise of G-Store visible: Placing related vertices in close proximity on the disk minimizes the necessary I/O for graph traversals.

Who forms the academic descendants of "Albert Einstein"? Or, formulated as a regular path query, retrieve all nodes reachable from "Albert Einstein" via only "hasAcademicAdvisor" edges. For this query we again visualize the buffer manager as G-Store traverses the graph. We conclude this demonstration with a *live mini-benchmark* of G-Store on such regular path queries against a storage layout that does not take the graph structure into consideration to give an impression of the achieved performance gain.

3. REFERENCES

[1] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *WWW*, New York, NY, USA, 2007. ACM Press.