Graph databases in the browser: using LevelGraph to explore New Delhi

Antonio Maccioni
Roma Tre University
Rome, Italy
maccioni@dia.uniroma3.it

Matteo Collina*
NearForm
Tramore, Ireland
matteo.collina@nearform.com

ABSTRACT

The pervasiveness of graphs on the Web is growing; however, the difficulty of managing complex graph structures curbs the development of web-oriented applications that embed network data. The open source project, LevelGraph, aims to overcome the obstacles that web developers face with graph data management. LevelGraph is an easy-to-use graph database layer for web applications.

To demonstrate various capabilities of the system, we developed a web-based application that utilizes a graph database of a tourist network in New Delhi. The application allows users to move around the city while LevelGraph executes graph queries on the underlying database. In this demonstration, we show how LevelGraph's features facilitate development and maintainance of web applications that embed graph data.

1. INTRODUCTION

Graphs establish a general way to represent information. Social networks, Semantic Web and bioinformatics are examples of domains in which it is natural to represent data in the form of graphs, and queries are expressed in terms of matching or traversal over those graphs. When applications have to present the graph-modeled data on the Web, a different approach with respect to traditional (semi-)structured data management should be considered. It is preferable to use "ad-hoc systems" for the persistence of graph databases. In the context of web development, people often rely on Graph Database Management Systems (GDBMSs), one type of NoSQL systems.

The web browser has become a reliable platform to run applications in order to reduce the workload of servers [1]. As a result, the functionalities that were previously split into 3-tiers of web applications are currently shared between client and server. Recent recommendations from the W3C suggest precisely this for the persistence of web application data by standardizing how to store structured and hierarchical data in the browser [2]. Consequently, it is increasing the need to use graph-modeled data within the client-side of web applications, counting possibly on a GDBMS. The system would be, in this case, tight-coupled with the web application running directly on the browser.

Unfortunately, while it is fairly easy to develop server-side applications backed by a GDBMS through the use of wrappers, web developers encounter several obstacles when relying on graphs on the client side. Since there are few effective systems available for this purpose, developers have no other choice but to always construct a graph-based persistence layer from scratch, often resulting in bugged and difficult-to-maintain software.

In order to overcome these limits, a community of researchers, engineers and developers has launched in 2013 an open source project, called LevelGraph[1], with the goal of building a robust and efficient graph database layer for web applications.

The flexibility required by web applications opens several challenges when adapting graph data management to web development. In particular, the design of LevelGraph has to take into account, at least, the following requirements: (i) integrate graph data with IndexedDB, the W3C guidelines for storing (semi-)structured data in the browser [2], (ii) employ a lightweight query planner and a lightweight query executor, given the limited resources of the platforms where applications embedding LevelGraph might run (e.g., mobile devices), (iii) the possibility to move the application from the client-side to the server-side and vice versa, implying possibly, also the change of the persistence layer, from disk-based to in-memory and vice versa, (iv) provide an event-handling system, and (v) support different kinds of web apps using graphs, such as those using RDF graphs.

LevelGraph is natively implemented in Javascript as a layer on top of a key-value store and can run in embedded mode within the web application[2]. However, it remains fully independent from an individual store and is currently pluggable to several existing key-value stores. Another distinctive feature is the possibility to use the IndexedDB of the browser to store graph data (in-browser working mode) [2]. Although LevelGraph has competitive advantages when running on the client side, it is worth noting that it is configurable to work server-side as well.

*Work partially done while at University of Bologna

[1]Released under the MIT licence at https://github.com/mcollina/levelgraph.

[2]The database runs in the same process of the application.

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

Contribution and Outline. With this demonstration:

- we give a comprehensive vision on LevelGraph and we sketch the decisions undertaken in its design (Section 2);
- we show to the audience how easy it is to employ graphs in web apps with LevelGraph. We developed a tourist web application where different routes are suggested to the user according to his current position (Section 3). The application is backed by LevelGraph that persists a simplified tourist road network of New Delhi;
- we present different demo scenarios using the app in order to highlight the main aspects of LevelGraph (Section 4);
- we outline future developments in Section 5.

2. LEVELGRAPH

This section briefly introduces the architecture (see Figure 1) and the main features of LevelGraph.

Architecture. The design of LevelGraph follows the general principles of NoSQL. Like other NoSQL systems, it reduces the impedance mismatch between the database and the business logics by bringing the persistence layer "closer" to the business layer of the application. One way to achieve that is by embedding the database within the application itself.

The data modeling follows the classic approach of disk-based graph databases where the graph comes in the form of triples representing its edges [3, 7]. Then, we adapt this scheme for in-memory and in-browser configurations. The *Indexing* component indexes each permutation of the triples (i.e. *spo, sop, pso, pos, osp, ops*) to enable direct access to data independently of the position of the variables, if any, in the query. We encode the position of the elements in the triple within the permutation itself and rely on a *Ordered Key-Value store* to persists the graph data. Let us consider a triple `t` for the edge `<vldb2016 locn:location leela>` of the graph in Figure 2, which represents that *Leela Ambience Gurgaon* is the *location* of *VLDB2016*. LevelGraph creates one different key for each indexing (i.e. `key1 = spo::vldb2016::locn:location::leela`, `key2 = sop::vldb2016::leela::locn:location` and so on) before inserting the corresponding six key-value pairs (i.e. `<key1, t>`, `<key2, t>`, ..., `<key6, t>`) in the store. Ordering the keys allows for a binary search which can be performed in place of a longer full scan. This approach to store the edges in a ordered key-value store was later adopted by other databases such as Google Cayley[3].

Now, let us suppose a query Q_1 = {`vldb2016 locn:location ?x`} containing one triple pattern for retrieving the *location* of *VLDB2016*. To solve Q_1, LevelGraph searches in logarithmic time for the triples in the store associated with the keys starting with `spo::vldb2016::locn:located::`, which returns the set of final answers to Q_1 (i.e. `x`=*Leela Ambience Gurgaon*).

LevelGraph uses, by default, *LevelUP*[4], a wrapper of *LevelDB*[5] for Node.js. It should be noted that it is possible, as long as the keys can be ordered, to plug-in LevelGraph to any key-value store without changing the core of the database engine or of the application, as shown in Figure 1. Currently, LevelGraph interfaces to *AWS DynamoDB*, *Redis*, *MongoDB* and *MySQL*.

[3] http://google-opensource.blogspot.it/2014/06/cayley-graphs-in-go.html

[4] https://github.com/rvagg/node-levelup

[5] https://github.com/google/leveldb

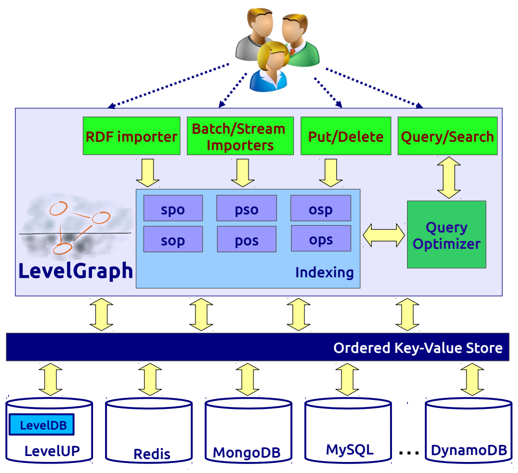


Figure 1: Architecture of LevelGraph.

A GDBMS should answer graph pattern matching queries. In order to do so, it must join different sets of edges. Although the underlying key-value store gives us access to these sets of edges, it is clearly not enough to compute the answers.

LevelGraph has a Query Optimizer component that takes the queries and generates a plan containing the sequence of accesses to the store. The plan accounts for how the triple patterns are connected to each other and joins intermediate set of triples accordingly. Since the triples are ordered, we can often use merge joins among sets of triples instead of nested joins. For example, in Figure 2, to determine how to reach *Qutb Minar* from *VLDB2016*, we might run a query Q_2 = {`vldb2016 ?p1 ?x . ?x ?p2 ?y . ?y gn:nearby qutb-minar`}, where the question mark "?" indicates variables and the full stop "." concatenates two query triple patterns. With Q_2, the optimizer produces a plan involving three accesses to the store (i.e. corresponding to the query triple patterns) and two joins (due to the two intersections in the query) between the corresponding intermediate result sets. At the end, Q_2 produces one answer having `p1`=`locn:location`, `x`=*Leela Ambience Gurgaon*, `p2`=`gn:nearby` and `y`=*Chaatarpur Temple*. In case of memory limitations, this query answering procedure is executed in *stream* mode.

Features. This section shows the main programming features of LevelGraph.

Client and Server: LevelGraph works in a client-side mode as a library for Javascript and in a server-side mode as a module for Node.js [6], the framework for running Javascript applications on the server.

Disk-Based, In-Browser and In-Memory: LevelGraph is able to persist the graph in different ways by switching the underlying key-value store. The default configuration is disk-based, but alternatively, LevelGraph can work in-memory and in-browser. When in-memory, we use *memdown*[6]. When in-browser, we use *level-js*[7], a LevelUp adapter to IndexedDB [2]. In this case, the database is stored in the IndexedDB of the browser. Google Chrome implements IndexedDB with LevelDB, Firefox Mozilla with SQLlite.

Querying Primitives: LevelGraph has two functions for querying the database. The function `get` can be used for asking single triple patterns (e.g., the query Q_1 in the previous example), while the function `search` accepts queries involving variables and joins between triple patterns (e.g., query Q_2). Both of them support conditions through the `filter` function and the management of large result sets with the parameters `limit` and `offset`.

Event Handling: one of the most important features of web programming is event-handling. In Javascript/Node.js one binds a *listener* function to the occurrence of an event. Analogously, active database systems use triggers to execute a certain task when a particular event happens (e.g., a tuple insertion).

LevelGraph is an active database where the event-handling system is inherited from Javascript and Node.js. In fact, a programmer can register a function to be launched when the execution of an operation on LevelGraph terminates (e.g., when the import finishes). The programmer can also register an operation on LevelGraph when an event on the web application occurs.

LevelGraph can be considered a "lightweight" active database because it relies on the already-running notification system of the operative system without using concurrent threads inside the application. Because LevelGraph can be used client-side where resources are limited, it is advantageous to use a "lightweight" active database. Currently, there is no other graph database system that supports event-handling. We found only a tentative implementation in the Blueprints library that produces notifications when the content of the graph database changes[8].

Maintenance Primitives: LevelGraph has a series of primitives for easily maintaining the database. The functions `put` and `del` are for adding and deleting triples, respectively. With their joint use we can update the triples in the database.

Data Formats: for extending the use of LevelGraph to Semantic Web applications, we implemented the add-ons LevelGraph-N3 and LevelGraph-JSONLD to operate with the two RDF serialization formats, N3/Turtle and JSON-LD, respectively.

Portability: since LevelGraph is built on top of Javascript and Node.js, it is a highly portable framework. We can also use the *browserify*[9] module available for Node.js to bundle the application with all the dependencies (i.e. including LevelGraph itself) into the same package.

Data Streaming: out-of-memory errors may occur when the query results or the data to be imported are large. In order to avoid this, we developed a version of `put`, `del`, `get` and `search` operations in *stream* mode. The query processing in stream mode is implemented as a pipeline, where each query triple pattern streams the results to the next pattern(s).

[6]http://github.com/rvagg/memdown
[7]http://www.npmjs.com/package/level-js
[8]https://github.com/tinkerpop/blueprints/wiki/Event-Implementation
[9]https://github.com/substack/node-browserify

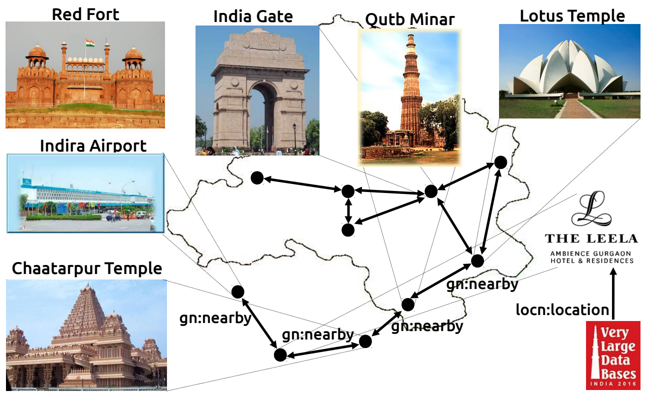


Figure 2: Portion of the graph database.

Related systems. GDBMSs store graph databases modeled as property graphs and work as servers or are embedded within applications [4]. In order to be used as a server, a GDBMS needs wrappers for different programming languages. For example, Neo4j [4] has several interfaces: Neo4jPHP[10] for PHP, Neography[11] for Ruby, Bulbs[12] for Phyton, node-neo4j[13] for Javascript. There are other server-side GDBMSs that specialize in managing RDF data and running SPARQL queries rather than providing features for developing web applications [3, 5]. RDF-3X [3] has a similar indexing scheme with respect to our and a more sophisticated query planner. In contrast, RDF-3X approach does not allow flexibility on the choice of the data store in the backend. gStore [5] has a C++/Java interface and uses also a key-value store. It has specialized indexes for analytical queries that are more fitting to querying large datasets rather than graphs for web applications.

When it is embedded in applications, a GDBMS runs inside the same process as the application [4]. Usually, GDBMSs runs in Java or C++ applications (e.g., Sparksee[14] works embedded in mobile and desktop apps), which are not widely used for the Web. HelioJS[15] is another Javascript GDBMS; however it works only in-memory. This means that once the browser is closed, the graph data is lost.

Finally, the multi-model databases rely on a simple data structure (e.g., a key-value store) on top of which they build interfaces for more complex data models. For example, OrientDB[16] is a graph database on top of a document-hybrid store, ArangoDB[17] has several data model layers on top of a document-oriented database and Cayley[18] is a graph database on top of a key-value store. They can all work embedded in applications but do not consider work in-memory, in-browser and disk-based at the same time.

[10]https://github.com/jadell/neo4jphp
[11]https://github.com/maxdemarzi/neography
[12]https://github.com/espeed/bulbs
[13]https://github.com/thingdom/node-neo4j
[14]http://sparsity-technologies.com/
[15]http://zuudo.github.io/helios.js/
[16]http://www.orientechnologies.com/
[17]http://www.arangodb.org/
[18]https://github.com/google/cayley

3. A TOURIST APP ON LEVELGRAPH

We have developed a web application to explore New Delhi. It uses LevelGraph as database management system in backend. The app uses a database that contains points of interest, which are connected to each other through a recursive relationship, thus, justifying the use of a graph database. The app proposes several routes towards a given destination as well as attractions on the way.

The graph databases in LevelGraph are schema-less in order to allow agile development. However, we show our road network in Figure 3 through a template that describes the graph modeling of the database used by the app. The template describes tourist attractions, each one having a name, a description, an address, two geographical coordinates (for computing distances among points of interest), suggestions for nearby places and associated events. In Figure 3, instances of the database are depicted with green circles and are connected through predicates (i.e. labels of the edges) to data values, here represented with blue rectangles. The instances belong to a class whose name is written in bold. For instance, Figure 2 depicts a part of the graph database used by the app and it is conforming to the template in Figure 3.

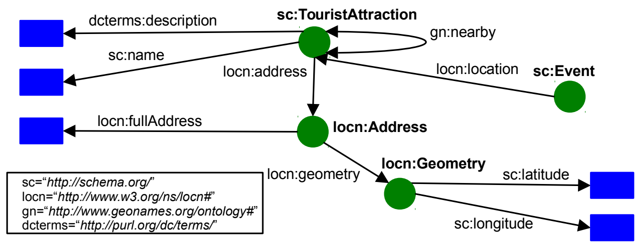


Figure 3: Template for a tourist network.

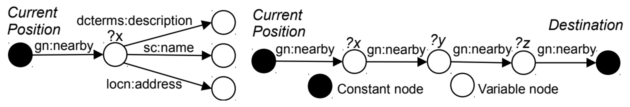
The road network template uses popular ontologies for publishing data on the Web (i.e. schema.org, dublin core, etc.). This allows us to show the add-ons for RDF data during the demonstration. The tourist application can embed LevelGraph since the database is small and can be downloaded at runtime.

4. LIVE DEMONSTRATION

We present different demo scenarios on the tourist web app. By interacting with the app, users can practice using LevelGraph as the main features are shown. The demonstration highlights that LevelGraph is an effective technology for developing the persistence layer of web applications.

Scenario A [Installation and Setup]. It shows the setup of the application on both the client-side and the server-side, pointing out that this configuration does not affect the source code. This scenario includes the installation of LevelGraph and shows some of the features introduced in Section 2 such as the *browserify* and the use of data importers. It also shows how the network database can be stored in the browser's IndexedDB. The takeaway of this scenario is to see how well-integrated LevelGraph is with Javascript.

Scenario B [Querying]. The app presents two possibilities to the user. When the user selects his current position, the app suggests nearby attractions and displays their description and relevant information. The query, in this case, has the current position as constant node and the rest of the nodes as variables. It contains a "star query" around the nearby place (see Figure 4(a)). Another option is to discover the sequence of points of interest between the current position and a given destination. The query is, in this case, a path with a constant in the first and in the last node, and all variables in the middle (see Figure 4(b)).



(a) Star query (b) Chain query

Figure 4: Example queries.

In addition to those submitted through the app, users can execute general queries over the road network via the command line interface of LevelGraph.

The takeaway of this scenario is to learn how to write queries and understand how easy it is to develop an application using LevelGraph.

Scenario C [Event Handling]. In this scenario we show how the event handling works by simulating a case where the database sends updates to an external server. An update contains the tracking of the current position of the user.

Scenario D [Maintenance and Streaming]. It shows how data can be updated while the application is running without interfering with the current session. We also show the streaming features of the system.

5. FUTURE DIRECTIONS

LevelGraph is an on-going project. Future work depends partly on the supporting community. However, we have identified several interesting directions. First, the set of available features will be expanded, such as the implementation of well-known query language interfaces that are becoming the standard languages for graph databases (i.e., Cypher and Gremlin). Second, it will be useful to improve the performance of the system by extending the indexing schemes (e.g., adding full-text indexes) and by improving query planning.

6. REFERENCES

[1] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, and D. McBeath. XQuery in the browser. In *WWW*, pages 1011–1020, 2009.

[2] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell. Indexed database API. Technical Report W3C Recommendation, World Wide Web Consortium, January 2015.

[3] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[4] I. Robinson, J. Webber, and E. Eifrem. *Graph databases.* " O'Reilly Media, Inc.", 2013.

[5] X. Shen, L. Zou, M. T. Özsu, L. Chen, Y. Li, S. Han, and D. Zhao. A graph-based RDF triple store. In *ICDE*, pages 1508–1511, 2015.

[6] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[7] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.