Technical Report

UCAM-CL-TR-820
ISSN 1476-2986

Number 820

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A unified graph query layer for multiple databases

Eiko Yoneki, Amitabha Roy

August 2012

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

A Unified Graph Query Layer for Multiple Databases

Eiko Yoneki and Amitabha Roy
University of Cambridge Computer Laboratory
Cambridge, United Kingdom
{eiko.yoneki}{amitabha.roy}@cl.cam.ac.uk

Abstract

There is increasing demand to store and query data with an inherent graph structure. Examples of such data include those from online social networks, the semantic web and from navigational queries on spatial data such as maps. Unfortunately, traditional relational databases have fallen short where such graph structured data is concerned. This has led to the development of specialised graph databases such as Neo4j. However, traditional databases continue to have a wide usage base and have desirable properties such as the capacity to support a high volume of transactions while offering ACID semantics. In this paper we argue that it is in fact possible to unify different database paradigms together in the case of graph structured data through the use of a common query language and data loader that we have named Crackle (a wordplay on Gra(ph)QL). Crackle provides an expressive and powerful query library in Clojure (a functional LISP dialect for JVMs). It also provides a data loader that is capable of interfacing transparently with various data sources such as PostgreSQL databases and the Redis key-value store. Crackle shields programmers from the backend database by allowing them to write queries in Clojure. Additionally, its graph-focused prefetchers are capable of closing the hitherto large gap between a PostgreSQL database and a specialised graph database such as Neo4j from as much 326 (with a SQL query) to as low as 6 (when using Crackle). We also include a detailed performance analysis that identifies ways to further reduce this gap with Crackle. This brings into question the performance argument for specialised graph databases such as Neo4j by providing comparable performance on supposedly legacy data sources.

1 Introduction

Graph structured data has become massively popular of late. Driving factors for this development include growing bodies of graph structured information (Wikipedia), large social networks (Facebook) [7] and geographic mapping (Google Maps), as well as biological research such as protein folding.

In online systems, this collection of information is critical for marketing, sales, and recommendation systems. There has been an effort to characterise and understand user behaviour in social networks [3][11][1], including information flow [9] and social communities [16]. In addition,

managing collected data and processing it in real-time is becoming important. This is driving the creation of specialised graph databases for storing and querying graph structured data.

In traditional database systems, storing and searching such data is difficult. Conventional schema (e.g. columns or fields) are required to be predefined thereby limiting flexibility. Google introduced Pregel [20] in 2009 as its large scale graph processing platform. Various other graph-based databases have emerged over the past few years, including Neo4j [22], InfoGrid [15], AllegroGraph [8], HypergraphDB [18], InfiniteGraph [14], and recently Trinity [21] from Microsoft Research. These databases are normally schema-less, allowing a set of nodes with dynamic properties to be arbitrarily linked to other nodes through edges. Many of these schema-less databases are based on key-value stores. The underlying assumption driving research in this area is that compared to relational database structures that do not support recursive relationships and are optimised for joins, graph databases allow for far more rapid traversal of edges. This allows for efficient analysis of complex relationships in a large set of data.

However, decades of research have been spent on the relational model that still dominates data storage, reliable high volume batch data processing and repeated transactions [29]. This motivates one to ask the question: is it not possible to store and query graph structured data in a traditional database such as PostgreSQL [24][31]? This paper is a careful evaluation of a typical query workload (such as navigation) on typical graph structured data (such as maps). As baselines we compare two complete database systems. The first is a complete PostgreSQL solution with the graph held in a PostgreSQL database. The second is a specialised graph database Neo4j, incorporating an entire storage stack specialised for storing and querying graphs. As expected there is a vast difference in performance between the two: Neo4j outperforms PostgreSQL by as much as 326 .

The key contribution of this paper is query language and query processing layer called Crackle (a corruption on Graph Query Layer or GraQL). Crackle is not a database on its own but rather designed to access and manipulate graph structured data without regard to the actual backend database storing the graph. Crackle runs in a Java Virtual Machine (JVM) and allows expressive queries in Clojure (a LISP dialect for JVMs)[4]. It also consists of a special query processor consisting of prefetchers developed specifically for graphs (a key contribution of this paper) and a data loader that can interface (thus far) with either PostgreSQL relational databases or Redis key-value stores[27]. We show in this paper that in-spite of being a general purpose way to query graph databases, Crackle running with the same PostgreSQL data source cuts the gap from the specialised Neo4j graph database to 6 from 326 . We also include a careful performance analysis that suggests ways to close this gap even further.

The rest of this paper is organised as follows. We begin with an overview description of the three database systems we have used in this paper (Section 2). We then describe our motivating workload (A* search) and dataset (Section 3). We then describe each of the three database systems in some details describing their operation and layout of data in Sections 4, 5 and 6. We then devote a section 7 to discuss the graph-specific prefetching strategies that we have built into Crackle. Finally we evaluate the three systems (Section 8). We then describe related work (Section 9) before concluding.

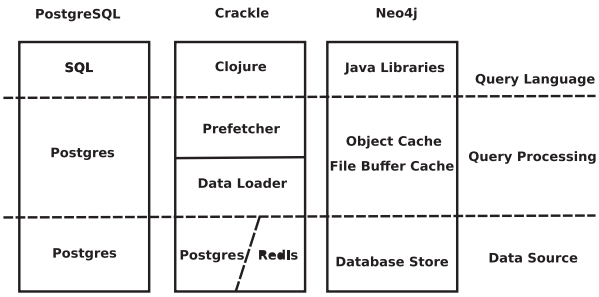

Figure 1: Compared Database Systems

2 System Overview

This paper focuses on the evaluation of three complete database systems on graph structured data. The three complete databases and the layers of interest in their software stacks is shown in Figure 1. Each database consists of a durable data source. On top of that lies the in-memory query processing and caching layers unique to each model. The topmost layer is a query language that aims to provide flexible and expressive access for programmers to the underlying databases. The query language also aims to shield the programmer from many of the lower level details such as indexes, caches and the provision of ACID attributes.

The first system we consider is a traditional PostgreSQL database. PostgreSQL is a standard open source SQL database. It has been constructed and optimised for the relational model, where all data is stored in tables of fixed dimensions with rows in different tables linked by common keys (columns). It is the leftmost system depicted in Figure 1. The second system we consider is the Neo4j graph database, depicted as the rightmost system in Figure 1. Neo4j deals exclusively with (and is optimised for) graph structured data, consisting of nodes (with associated information) and links (edges) between the nodes.

The third system and the key contribution of this paper is Crackle, depicted between the PostgreSQL and Neo4J solutions in Figure 1. Crackle is a query processor and query language designed for graph structured data, but crucially does not impose any requirement on the underlying data model. The functional requirement for the system as a whole was for it to accept the definition of a backend data source in a (structured) plain text format and allow it to be queried as a graph as if it were locally available. One of the possible data sources for running queries in Crackle is relational data such as a PostgreSQL database (show in the Figure). Databases such as Neo4j use key-value stores, where data is stored as a single key coupled with arbitrarily sized values rather than as a relational table. To reflect this possible paradigm shift in how graph data might be made available, we also support the Redis [27] in-memory key-value store. Crackle is therefore able to unify to two very disparate sources of data: a relational database and a key-value store. The abstraction of these data sources is the role of the data loader component.

Workloads that traverse graphs often show large amounts of what can be termed as “semantic” locality. Consider a navigational query: “nearest police station”. Such a query is likely to traverse the road network graph starting from the current location with the next node (data item) to be fetched being spatially contiguous to the current set of roads being considered. Another example might be “find a person on my social network who likes sushi and are currently located near me”. The next data item to be fetched must be in the friends list of the current set of items

(people) being considered. This semantic locality exposed by graph traversal algorithms is dissimilar to spatial or temporal locality generally exploited by relational models or simple key-value stores, where an order function on a key serves as a hint to arrange data in a table or index. There is no simple order function in a graph that places linked nodes close to each other (it is impossible for a complete graph).

A portion of Crackle's design is therefore devoted to building prefetchers that can exploit this semantic locality (the prefetcher block in Figure 1) and is discussed extensively in the paper. Indeed, as we show in the evaluation, the prefetcher is key to closing the performance gap when operating with the PostgreSQL data source in comparison to Neo4j.

In later sections (4, 5 and 6) we describe the internals of each of these databases in the context of our graph traversal workload. In the next section however, we first describe the graph traversal workload and the data sources we use in the paper. This aids in the understanding of later sections through specific examples from the workloads, particularly for readers who may not be familiar with the internals of relational tables and key-value stores.

3 Query Workload

In this section we describe the query workloads used to evaluate the database systems. Queries on graph structured data inevitably lead to a traversal of the graph, seeking a specified goal. We wished to use the exact same traversal algorithm in all of the three database systems ensuring that we fairly compare them. At the same time the traversal algorithm needs to be representative of real-world usage of graph databases. Keeping this in mind we used the A* search algorithm, which is fairly representative of how one might go about implementing a goal seeking query on graph structured data. A* search is a commonly implemented algorithm for path finding in networks (such as navigational queries on maps), and decision making using game trees.

A* search can be used to seek paths between distant nodes without exhaustively traversing the whole graph due to the heuristic function guiding the search. This tests the capacity of the database to supply data to the query as searches may follow a large number of edges within the graph, even when the requests may not have traditional spatial or temporal locality. The heuristic calculation and priority selection also tests the expressivity of the query language. As we show later, this brings out the relative difficulty of using SQL compared to Clojure (in Crackle) or Neo4j. Since A* is fundamental to understanding this paper, we include a detailed description for the reader who may not be familiar with it in Section 3.1.

Finally, we used queries on a real world dataset to drive the search. We describe the dataset we have used in Section 3.2.

3.1 A* search

The A* search algorithm [12] aims to find a path of minimum cost in a graph from a given source node to a given goal node; it is an extension of Djistra's shortest path algorithm. The core of A* [as implemented in this paper] is a priority queue of paths. The priority queue of

Algorithm 1 A* search

```
01      while true:
02       current_path = prio_queue.pop() // path with lowest heuristic cost
03       path_vertex = last(current_path.path)
04       if is_target(path_vertex):
05         return current_path
06       else if path_vertex in closed_set:
07         continue
08       else:
09         closed_set.add(path_vertex)
10         for edge in outward_edges(path_vertex):
11           if edge.destination in closed_set:
12             continue
13
14           new_path = current_path.copy()
15           new_path.path.append(edge.destination)
16           new_path.cost = current_path.cost + edge_cost(edge)
17           new_path.heuristic = new_path.cost + heuristic_cost(edge.destination)
18           prio_queue.insert(new_path)
```

paths e is ordered by a function that attaches a weight to each path:

$$\text{Weight}(e) = \text{PathCost}(e) + \text{Heuristic}(\text{e.destination})$$

Thus in addition to exploring paths with the minimum cost (a greedy approach), A* search also uses a heuristic to estimate the distance of the end of located paths from the goal. The path with minimum cost is picked at each iteration and expanded by exploring all neighbours of the vertex at the end of the path.

Pseudocode for the A* search algorithm is shown in Algorithm 1 . If the current path reaches the goal of the algorithm, then it is returned as the result (lines 4–5). Otherwise, new paths are generated by following each edge connected to the end of this path, and their new accumulated cost and heuristic value calculated. These new paths are then added to the priority queue (line 18). Additionally, the algorithm needs to maintain a closed set containing all the vertices already visited, so that no vertex if considered more than once (lines 6–7 and line 9).

3.2 Datasets

The evaluation in this paper was done using the DIMACS [6] dataset. This is an openly available dataset specifying the latitude and longitude of road junctions in the United States and the connectivity (through roads) between these junctions. The road junctions thus serve as nodes and the roads themselves as edges in a graph. We drove the system through route finding queries. The latitude and longitude information attached to nodes is also retrieved on accesses and used as inputs into the A* search that is attempting to construct shortest routes. The query needs to calculate both the length of each path (road) and the heuristic which we chose to be the distance of the current end of the path to the goal. Both of these can be calculated from the latitude and longitude of the involved points (road junctions at either end of the road and the lat-long information for the start and end points). We used the great circle distance (assuming

the Earth to be a perfect sphere):

$$\begin{aligned}&\text{Distance}(a, b) = \\ &6371 \quad cos^{-1}(sin(a_{\text{lat}})sin(b_{\text{lat}})+ \\ &cos(a_{\text{lat}})cos(b_{\text{lat}})cos(a_{\text{long}} \quad b_{\text{ long}}))\end{aligned}$$

In addition to the DIMACS dataset we have used other datasets for testing. However since they are not relevant to this paper we omit any discussion of them.

4 Crackle System Architecture

Crackle is written entirely in Clojure [4]: a LISP dialect that can run in a Java Virtual Machine. The main features of interest in Clojure are that it is functional, has easy Java interoperability and has language features for concurrency, such as immutable data structures. Immutable data structures mean that code is naturally thread-safe except when using mutating structures from the Java standard library. Clojure also provides strong implementations of many of the data structures we require. Finally, Clojure provides direct access to the Java ecosystem and thus can also access Neo4j. This means that we were able to reuse query algorithm code across the two and since both run in a JVM, produce a more direct comparison of speed.

As Clojure runs on the JVM, being able to test code quickly can be troublesome. Therefore we built a tool using the cake build tool for Clojure, which supports per-project persistent JVMs and a REPL (read-evaluate-print-loop). This makes it possible to have an instantly-available environment into which code can be entered or loaded/reloaded from file. This availability removes the common pain point of a compiled language, as the compilation step is effectively removed, and it allows interactive development and inspection of running code.

We now describe individual components of Crackle: the query language, query processing and data loaders.

4.1 Query Language

Crackle is generic enough in its design that one can write queries in any language such as python or C/C++ simply by adding appropriate adapters. Currently however, queries in Crackle are also written in Clojure. They are geared towards lazy loading of data (as is natural with a functional language). Algorithm 2 contains some snippets of Clojure code that are key to the ideas in this paper.

The first function `astar-route-find` implements the A* workload used in the evaluation for this paper. As the comments indicate, it uses a stadard Java priority queue (with appropriate wrapping) at its core (line 2). Being able to use standard Java classes is one of the benefits of using Clojure to write queries. from the graph. We initialise this priority queue to contain a path with only the start node and of cost `0` (line 04). The next few lines set up a bounded loop (lines 5–8), the bound is for sanity checking that is necessary to handle situations such as when the target is altogether absent The remaining part of the A* implementation is a straightforward

Algorithm 2 Clojure snippets

```
01 (defn astar-routefind [graph start target limit]
02  (let [q (java.util.PriorityQueue. 1000 (Comparator. (gen-Heuristic target))) ;Use Java prioq
03        closed (atom #{})]
04    (.add q {:dist 0 :path [start] :h (latlong-distance graph start target)}) ;Initialise prioq
05    (loop [i 0] ; Place a limit (sanity check) on number of visited nodes
06      (if (== i limit)
07        (do #_(println limit "expansions, giving up.")
08          {:found nil :considering nil})
09        (let [{p-dist :dist p-path :path :as p} (.poll q) ; Dequeue the least cost item
10              p-head (peek p-path)]
11          (if (get @closed p-head)
12            (recur i) ; If the node we pulled out has been visited, continue
13            (if (== p-head target)
14              {:exp i :found p :considering nil} ; found our target ! return path
15              (do
16                (let [closed-set (swap! closed conj p-head)] ; add node to the visited set
17                  (doseq [[edge node] (c2/get-edges-out graph p-head :road)] ; Every neighbour
18                    (when (nil? (get closed-set node)) ; add extended path to the prioqueue
19                      (let [n-dist (+ p-dist (latlong-distance graph p-head node))]
20                        (.add q {:dist n-dist :path (conj p-path node)
                                   :h (+ (latlong-distance graph node target) n-dist)}))))))
21                (recur (inc i)))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

01 (defn- ensure-node [graph id]
02  (let [g @(:atomref graph)]
03    (if-let [node ((:vertices g) id)] ; If node already in our loaded graph just return graph
04      g
05      (let [{nodes :n node-edges :e} ((:node-load graph) id) ; Call specialised node-load
06            insertions (atom #{})]
07        (doseq [{:keys [id props]} nodes]
08          #_(println "Adding node" id)
09          (when-not ((:vertices @(:atomref g)) id) ; Add loaded node details to the graph
10            (add-node graph id props)
11            (swap! insertions conj id)))
12        (doseq [{:keys [from to label props]} node-edges]
13          #_(println "Adding edge " from "--" label "-->" to)
14          (when (or (get @insertions from) (get @insertions to))
15            (add-edge graph from label to props)))
16        @(:atomref graph)))))
```

implementation of the pseudocode description of Algorithm 1. We iterate by picking out the path with lowest heuristic cost (via the `poll` function, line 9) and considering the node at the end of the path (`p-head`). We reject this path if `p-head` is already in the closed set (lines 11–12) and return this path if `p-head` is the target node (lines 13–14). We then add `p-head` to the closed set (line 16) and consider extensions of this picked path by appending all possible neighbours of `p-head` (line 17). If the neighbour is not in the closed set (line 18) then we compute the heuristic for this extended path and add it to the priority queue (lines 19–20).

The second function (`ensure-node`) shows how Clojure interfaces with the backend data source. This is a generic function called to ensure that the needed node (specified by `id`) is in main memory. The most important step is to call `node-load`), which is specialised depending on the data source. The remaining steps update main-memory data structures that hold information about loaded nodes.

```
{
 "server": "127.0.0.1",
 "database": "roads",
 "user": "chris",
 "node-space": "homogeneous",
 "node": {
   "sql": "SELECT lat_f as lat, long_f as long
       FROM junctions WHERE id = ?",
   "properties": ["lat", "long"]
 },
 "edges": {
   "road": {
     "sql": "SELECT j_to, distance
       FROM roads WHERE j_from = ?",
     "target": "j_to",
     "properties": ["distance"]
   }
 }
}
```

Figure 2: JSON definition for the road network from PostgreSQL

4.2 Query Processing

The first key consideration in the query processing layer is an in-memory representation of the graph. We treat vertices and edges as first class objects. Each vertex has an ID, a set of properties, and a set of labelled edges. Vertices are stored in a hash table (dictionary) structure indexed by the ID for quick lookup. The set of labelled edges is maintained as a hash table indexed by label. Each edge is a 3-tuple of a start ID, end ID, and a hash of properties. Edges only refer to their start and end vertices by an ID rather than a reference, because the data may not be present yet in the graph. This allows for lazily loading vertices as required, but means that vertex IDs are duplicated by edges.

4.3 Data Loading

The data loading format used was JSON [17]. JSON is a lightweight markup based on Javascript notation that supports numbers, strings, hashes, and arrays. A third-party library[5] was used in order to parse JSON into Clojure objects. We specify the database and scheme in JSON, which then acts as an interface to the appropriate data source.

The JSON specification for connecting to the PostgreSQL road network database is shown in Figure 2. The first part deals with retrieving road junctions (vertices) given their ID. Also retrieved are the latitude and longitude. The second part deals with retrieving roads (edges) given their originating junction.

The other data source we used for Crackle is Redis, which is a key-value store with a flat key structure. Values can be strings or higher-order structures with string values (hashmaps, sets, lists). Properties for a vertex or edge thus naturally fitted to a hashmap. Figure 3 therefore describes the road dataset for the Redis data source.

```
{
 "node-space": "homogeneous",
 "node": {
   "properties": ["lat", "long"]
 },
 "edges": {
   "road": {
     "suffix": "out"
   }
 }
}
```

Figure 3: JSON definition for road network from Redis

5 PostgreSQL System Architecture

One of the databases evaluated is a pure PostgreSQL solution. PostgreSQL supports a large subset of the SQL standard for querying data in the database. It also supports a procedural language PL/pgSQL as an extension which allows for finer programming control.

PostgreSQL is a standard open source SQL database, and supports a wide range of data types, including ‘numeric’. This type allows for arbitrary precision numbers, at the cost of performance of calculations on them. This means that the latitude and longitude can be converted to a decimal representation at insertion rather than at computation time. The structure of the table used to represent road junctions is shown below.

Property	*Type*	*Indexed?*
id	bigint	yes
lat	bigint	no
long	bigint	no
lat_f	numeric	no
long_f	numeric	no

Edges are stored in a separate table, with indices on ‘from’ and ‘to’ as these will be used to find edges during traversal of the graph, shown below.

Property	*Type*	*Indexed?*
j_from	bigint	yes
j_to	bigint	yes
distance	bigint	no

We wrote a native SQL query to implement the A* search algorithm using the ’WITH RECURSIVE’ syntax in PostgreSQL. A recursive SQL query is composed of an initial query, and a secondary query that is used to iteratively add rows to the result. This is because the algorithm needs to obtain a new set of vertices based on a dynamic current set (rather than a static starting set for the query).

The major problem encountered in implementing this query in PL/pgSQL was the lack of data structures supported, requiring the use of temporary tables in the implementation of a priority queue and a closed set.

The closed set is a simple table using a unique index to quickly check for the existence of some

key in the set. The priority queue is more involved and is implemented using the table below.

Property	*Type*	*Indexed?*
id	bigint	yes
distance	real	no
hval	real	yes
path	bigint[]	no
head	bigint	no

In the table, HVAL is the heuristic cost of the path, and used as the priority measure, with HEAD as a convenience as the last vertex in PATH, and DISTANCE the total accumulated real cost. ID is necessary so that a row can be removed from the table after it has been selected. An index is maintained on the HVAL field to speed up locating the node with lowest cost. As we show in Section 8.3 of the evaluation, this index is a key bottleneck in PostgreSQL performance.

6 Neo4j System Architecture

As Neo4j runs on the JVM, it can be used directly from Clojure code, which the rest of the project was written in. This also means that we were able to use the A* query implemented in Clojure to access the Neo4j database with only minor changes. The only other piece of work needed with Neo4j was setting up the DIMACS road database in it.

Neo4j supports storing values as any of the Java primitives (or arrays thereof) and strings. Storing a decimal representation of the latitude or longitude in a float or double would lose precision and cause inconsistent results, and so for Neo4j the latitude and longitude were inserted as integers, with division down to degrees done at computation time. Neo4j is schema-less, and so requires indices to be defined at insertion time for properties. For the road network, the only index required is for junction IDs so they may be retrieved arbitrarily. Once the vertices were inserted, the edges were inserted in a second pass. The start and end vertices are retrieved and a 'road' labelled edge added between them, with the distance between the junctions as an integer property. This gives the properties for each vertex in the graph as in the following table:

Property	*Type*	*Indexed?*
id	int	yes
lat	int	no
long	int	no
distance	int	no

7 Prefetchers

A key contribution of Crackle is the introduction of prefetchers as part of the query processing phase. One of the more difficult problems in dealing with graph traversal is the fact that accessing nodes often leads to random accesses, leaving no clear way to optimise IO. In addition, the fact that we wish to interface Crackle seamlessly with multiple data sources requires that any

solution to this problem should be generic enough to apply to any backend database, particularly those indexed or ordered by keys such as Postgres. In the case of a graph it is difficult to obtain locality on the keyspace for adjacent nodes, particularly for dense graphs.

Our solution to the problem of random access is to introduce the capability in Crackle to prefetch nodes based on adjacency in the graph. The basic idea is simply to issue a prefetch request for adjacent graph nodes whenever any node is accessed. We have implemented two kinds of prefetchers in Crackle. Although both are intuitive, we show in the evaluation that prefetching leading to large boosts in performance.

The first prefetcher (called `lookahead`) simply does a limited depth-first traversal of the graph from the provided node. It takes a depth parameter that limits the maximum path-length of the explored graph from the given node. For example, specifying a depth of 1 leads all neighbours of the given node to be prefetched. A key limiting factor of lookahead prefetching is that it is ultimately limited by the cost of bringing in a single node, since a prefetch request for a neighbour cannot be issued until the current node has been fetched and its neighbours determined. Crackle provides an alternative lookahead mechanism called block prefetching that can ameliorate this problem.

Block prefetching is based on the idea that certain keys in the underlying datastore *often* have a strong correlation with adjacency in the graph. Consider the case of a social network where nodes (people) are indexed by their postcodes. People with adjacent postcodes have a higher than normal probability of being connected. Another example of this is the prefetching strategy we used for queries on our roadmap dataset in the evaluation. For any given road junction (`x`) , depending on the underlying data store, it is possible to retrieve all road junctions (vertices) within a certain distance (`d`) : $y : \text{dist}(x, y) < d$. Although there need not be a path from the current road junction to these block prefetched road junctions there is a high probability that there is one and this proves to be greatly beneficial to A* searches on the road network.

We focus on prefetchers for Postgre in this paper, as it is the only on-disk database that we consider. Both prefetchers therefore, are implemented as SQL procedures called from Clojure code.

8 Evaluation

We evaluate the three databases on an Intel Core2 Duo 2.4 Ghz system with 8GB of RAM. In order to obtain reliable results, all benchmarks were run ten times independent of one another. The JVM has global pauses for garbage collection, and so before every test a manual garbage collection was triggered to try and ensure that any pauses were at the fault of the test being run, rather than previous conditions. The fastest and slowest times for each run were also discarded to reduce outlier impact, and then the mean of the remaining eight results was taken. The JVM was also provisioned with minimum and maximum RAM set to two gigabytes, hence time should not be lost when trying to acquire more memory from the system. We did one pre-run of each benchmark in order to give the JVM JIT compiler time to optimise the code.

As mentioned, the core benchmark is an A* traversal on the DIMACS dataset. For repeatability, we fixed 6 queries formed by randomly selecting start and end points such that they are

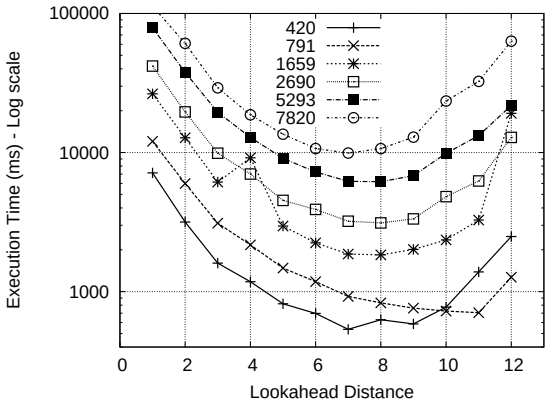

Figure 4: Varying prefetch lookahead depth

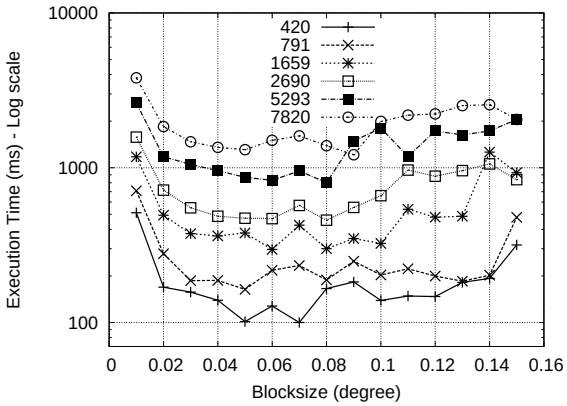

Figure 5: Varying prefetch block size

separated by an increasing number of nodes. The objective is to exercise the databases with A* searches of various necessary depths to reach the goal: our queries ended up having A* search subgraphs of sizes 420, 791, 1659, 2690, 5293 and 7820 vertices.

8.1 Prefetchers

The first piece of evaluation we did was to determine whether the prefetchers are indeed effective in improving the performance of graph traversal and if so, determine the optimum settings for the prefetchers. We determined early on that Redis being an in-memory key value store did not show any benefits with prefetching and therefore we focus only on the PostgreSQL data source for the prefetchers. We evaluate the lookahead and blocking prefetchers previously described in Section 4.2.

Figure 4 shows an analysis of the effect of changing the depth of the lookahead prefetcher. For all the A* search sizes, increasing the lookahead distance gives higher performance as it returns more vertices per call, reducing the overhead of querying PostgreSQL. However, the cost of the prefetch procedure itself then begins to grow larger than the avoided overhead. This occurs because the prefetch recursion on the server side is not linear in growth, and hence

smaller lookaheads are more efficient than larger prefetches. Also, although larger prefetches may mean fewer queries later, they also mean overlap with previously obtained vertices. Based on this experiment, we chose a lookahead depth of 7 for later experiments.

The next piece of evaluation we did was to determine the optimum query range for the blocking prefetcher. For the DIMACS dataset, for every query for a particular vertex, the blocking prefetcher also queried for all nodes within a square containing the fetched vertex at the centre. The side of the square is measured in degrees (recall that we are querying based on latitude and longitude that are themselves measured in degrees). Figure 5 shows the results of varying the size of the block (square). Increasing the block size gives an initial large performance improvement, and then levels off, and then tends towards an increase in execution time. The increase in execution time is more pronounced for longer traversals suggesting that the increase for individual queries is not much but the additive effect of many queries for longer traversals is larger. Another conclusion is that block prefetch is less wasteful in terms of fetching nodes that have already been visited as compared to lookahead prefetch. Based on this experiment, we chose block sizes of 0.05° and 0.1° for subsequent experiments.

8.2 Database systems

We now evaluate the three database systems against each other for A* search on the DIMACS dataset for the largest query: ~8k visited nodes. Figure 6 examines performance for the following database systems and configurations:

1. Neo4j
2. PostgreSQL
3. Crackle with PostgreSQL data: labelled Crackle [no prefetch]
4. Crackle with PostgreSQL data and prefetcher type lookahead 7: labelled Crackle [lookahead 7]
5. Crackle with PostgreSQL data and prefetcher type blocking 0.1: labelled Crackle [block 0.1]
6. Crackle with PostgreSQL data and prefetcher type blocking 0.05: labelled Crackle [block 0.05]
7. Crackle with Redis data: labelled Crackle [redis]
8. Crackle preloaded: labelled Crackle [preloaded]

The last item requires some explanation. One of the database systems we evaluated was Crackle with the entire graph traversed by the A* search preloaded into memory obviating the need to make any queries to any backend data source. This lets us evaluate the cost of query processing without including the cost of data access.

Returning to Figure 6 we see that Crackle preloaded is the fastest meaning that the cost of data access is indeed the dominating factor. The next fastest is Neo4j while the slowest is Crackle when running with PostgreSQL data and PostgreSQL. This underlines the classic argument that "legacy data sources" such as PostgreSQL are not performant enough for graph data necessitating the construction of specialised databases such as Neo4j. However, we see that Crackle

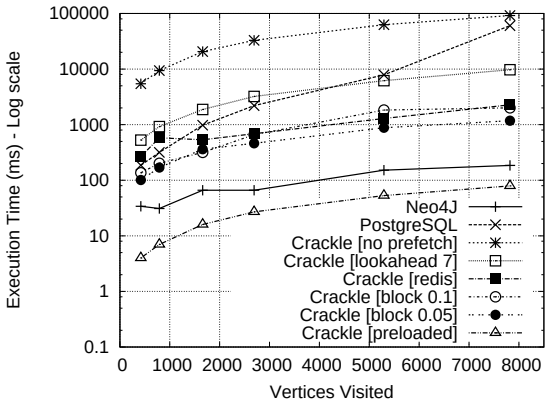

Figure 6: Comparing database systems on A* searches (New York State)

running with the same PostgreSQL data source but with a prefetcher closes the gap from 326 worse with a pure SQL query to only about 6 worse with the block prefetcher. It also performs similarly to Redis an in-memory key value store suggesting that relational databases are no worse at storing graphs than key-value stores provided an intermediary such as Crackle is used.

8.3 PostgreSQL Priority Queue Analysis

One of the core interests of this study was in determining why a pure relational database system performs badly on graph traversals. One might ask the question that if engineering resources were to be devoted to improving the performance of PostgreSQL on A* search, what should be the first angle of attack?

To answer this question we used timestamps to evaluate exactly where PostgreSQL was spending its time when executing A* search. We measured the relative time spent in 4 zones. Zone A covers finding the next path in the priority table and removing it, Zones B and C cover checking if the path's head is in the closed set table, and adding new vertices to the closed set. Zone D covers creating new possible paths and calculating their heuristic values, and adding them to the priority table.

We found that for the largest query (~8k nodes) the time spent in Zone A rapidly grows and occupies about 75% of the total time spent in the 4 zones. This is because the priority queue is maintained as a SQL table with an index on the ordering field (Section 5). The performance of this priority queue is limited by the performance of the index on the HVAL property, since that index is consulted to remove the path of lowest cost from and the table and must be updated for every path added. Internally, the indices in PostgreSQL use the B-tree data structure. The B-tree is a tree where every path from root to leaf has the same length (guaranteeing logarithmic lookup times) and each node is large with multiple children for optimal transfers to and from disks with large sector sizes. This turns out to be excellent for traditional SQL databases where joins are important but sub-optimal when used as a priority queue. A property of B-trees is that the leaves essentially contain the elements in sorted order. In the case of a priority queue where one repeatedly removes the smallest element, it causes a leaf at one of the extremities to continually

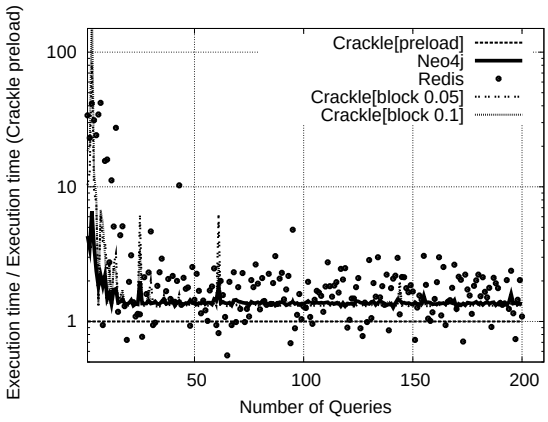

Figure 7: Normalised query time for a series of A* queries

lose elements. This in turn causes repeated rebalancing of the B-tree due to the depleted leaf node being merged with its neighbour. This is one reason for the suboptimal performance of the PostgreSQL solution. However we saw no other alternative to using the index since otherwise locating the smallest element in the queue would be prohibitively expensive.

This would suggest that the first order of business in improving PostgreSQL performance is to improve the availability and performance of intermediate data structures such as priority queues without falling back to the traditional indexing methods. Clojure for example, uses priority queues implemented as traditional binary heaps.

8.4 Performance with a Hot Cache

Thus far, the evaluations have focused on queries with a cold cache where the prefetcher played a key role in determining query performance. A hypothesis here is that if the same (set of) queries were to be run repeatedly, the set of traversed nodes should be in Crackle's cache, leading the best performing versions (Crackle with block prefetching) to perform the same as Crackle with the entire graph preloaded. To this end we evaluate the best performing subset of database systems described in the previous section: Crackle preloaded, Neo4j, Crackle with PostgreSQL and block prefetching; and finally Redis. We used a fixed set of 20 A* queries each traversing about 8k nodes. We repeated this series of queries again and again to observe whether database performance converged as expected.

Figure 7 shows the results. We normalised the runtimes to that of Crackle preloaded. The first surprising aspect is that Crackle with block prefetching converges to Neo4j speeds (1.3 slower than Crackle preloaded) rather than Crackle preloaded speeds. The most likely explanation for this is that in Crackle preloaded we have removed calls to the backend database allowing the JVM more room for optimisation. This also suggests that there is no essential difference between the performance of Neo4j and Crackle were data loading to be removed from the picture. Another aspect that surprised us was the large variability in performance of Redis, something we had not seen with a cold cache. At this point we are not clear about the reasons for variability with Redis. It only occurs for long runs such as this benchmark rather than with cold caches as above. In our setup, the Crackle dataloader communicates with Redis over a

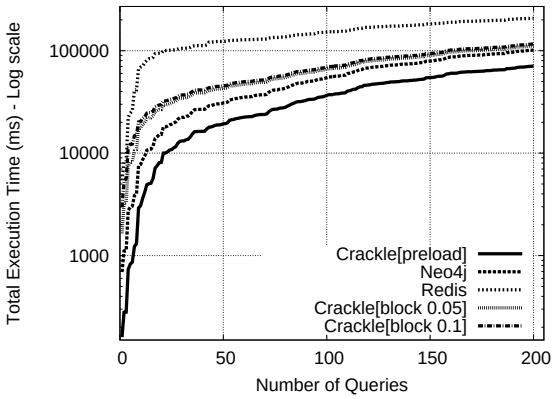

Figure 8: Aggregate time for a series of A* queries

network socket. It is possible that the upswings in query time with Redis are due to the Java garbage collector contending for CPU time and cache with the Redis process. Moving them to different CPUs (or even different systems) to provide better isolation between the JVM running Crackle and Redis might mitigate this problem.

A clearer picture is obtained if we plot the cumulative query time, which helps to smooth out the variation in Redis runtimes. Figure 8 shows the aggregate time for the A(*) queries. It clearly brings out that Crackle with block prefetching converges to Neo4j performance, while Crackle with Redis is slower and Crackle with preloading is the fastest.

8.5 Large Graphs

We now report on results when using the *full* DIMACS dataset that covers all roads junctions across the United States (24 million nodes). We determined that a prefetch block width of 1.0° is more efficient for this larger graph, reflecting heterogenity in the roadmap data set between the small subset considered thus far (New York) and the whole dataset.

We compare database systems for A(*) searches on the whole roadmap data in Figure 9. Crackle with block prefetch outperforms running Crackle directly against the Postgre databases, whose runtimes are so large for the entire graph that we are unable to report them. This underlines the utility of prefetching when traversing large graphs, a key point of the paper. One surprise is the relatively poor performance of Crackle when compared to Neo4j. In order to understand this we reran the experiment with Neo4j limited to 1GB of main memory (compared to 2GB for Crackle). In this situation Crackle *outperforms* Neo4j on this large dataset. This investigation uncovered inefficiencies in our Clojure code to do with object bloat and large amounts of temporary object creation that triggers frequent runs of the Java garbage collector. We believe that carefully optimising both the Clojure runtime and our own code should be instrumental in reducing or eliminating the gap in performance we now have with Neo4j.

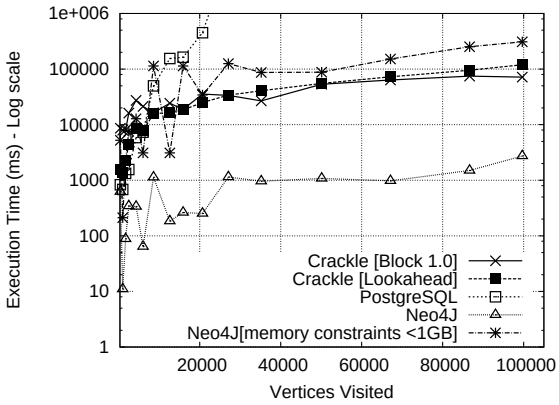

Figure 9: Comparing database systems on A* searches (Whole USA)

9 Related Work

Distributed Hash Tables (DHT) [13] and key-value stores are popular methods for addressing the scalability problems of large-scale data processing. The recent popularity of Online Social Network analytics has renewed interest in such mechanisms (e.g. Twitter uses Cassandra [19]). While these methods achieve scalability by random partitioning of data stores, they do not exploit the strong semantic locality that is present in these vast datasets.

SPAR [25] shows that semantic locality in back-end data can be exploited for partitioning data into independent components, especially for online social networks. This clustering assists in improving application scalability. Using social information to improve the scalability of centralised OSNs such as Facebook or Twitter is described, especially in terms of how co-location of data in social proximity can be maintained in servers at the same physical location for reducing network traffic. The motivation of Crackle is based on a similar concept, where search, update, and dissemination of data would occur in many cases in preloaded subgraph of the whole graph. 64-bit addressability with large memory makes in-memory processing of these vast data sources possible. Keeping partitioned subgraphs in memory will improve performance significantly.

Ficus [10], Farsite [2] and Coda [30] are distributed file systems which make files available by replication. Distributed RDBMSs (e.g. MySQL) and Bayou [32] provide eventual data consistency. In contrast, if semantically related data is grouped locally, it can be maintained without distributing the data for processing. SPAR's approach is based on this principle, and it is more efficient for OSNs as it requires fetching data from multiples servers constantly. Crackle could be used by the distributed file system for its performance and availability.

Understanding the dynamics of the graph topology and social aspects of different online social networks (e.g. Twitter, Facebook, Google+) is key to building more efficient computer systems [28]. It has effects on storage management, reduction of network traffic, and service availability. Understanding locality is an important aspect for building large scale applications, especially in terms of improving the system design and performance. Partitioning data can be based on query patterns but there will be further issues when both data and the network are dynamic. A series of topics in this research domain is emerging.

Relational databases and graph databases support extensive query support. Existing relational database products include SQL Server, Oracle [23], MySQL, PostgreSQL, and SQLite. In our evaluation, we selected freely available systems that work on all platforms. This left MySQL, PostgreSQL and SQLite. PostgreSQL and MySQL are quite similar. Investigating their SQL support, however, PostgreSQL has support for recursive queries while MySQL does not. Our implementation of queries relies on such functionality, we decided that PostgreSQL would be the best representative choice for our performance evaluation with Crackle.

Recently a series of general graph databases has been emerging including Neo4j[22], AllegroGraph [8], HypergraphDB [18], Trinity[21] from Microsoft Research, and Pregel [20] from Google. Crackle is complementary to these systems and its approach can be integrated into them. Pearce et al [26] has taken an approach to store part of graph in memory together with distributed computation.Kang et al [33] has demonstrated scalable graph processing algorithm for map/reduce operation.

10 Conclusion and Future Work

We have presented Crackle, a graph query layer, which allows graph-structured data to be lazily loaded from multiple data sources. The Crackle language allows queries to be performed on these datasets. Crackle is implemented in Clojure, and currently supports Redis and PostgreSQL data sources.It is instrumental in closing the gap between supposedly "legacy" data sources such as PostgreSQL and specialised graph databases such as Neo4j.

There are two primary future directions of development we are pursuing with Crackle. The first is to completely close the performance gap between the PostgreSQL data source and Neo4j. The second direction is to expand the applicability of Crackle, which in turn should improve the expressivity of queries in Clojure. We have already created additional data source connectors to connect to Facebook and Twitter, thereby forming the substrate to support ongoing research into these two social networks. Crackle is also capable of unifying views from disparate data sources.

Acknowledgment The research is part funded by the EU grants for the Recognition project, FP7-ICT-257756 and the EPSRC DDEPI Project, EP/H003959. We would like to thank members of Systems Research Group, University of Cambridge for their comments and suggestions.

References

[1] A. MISLOVE AND H. S. KOPPULA AND K. P. GUMMADI AND P. DRUSCHEL AND B. BHATTACHARJEE. Growth of the Frickr Social Networks. In *WOSN* (2008).

[2] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., R., J., LORCH, THEIMER, M., AND WATTENHOFER, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI* (2002).

[3] A.MISLOVE AND M. MARCON AND K.P. GUMMADI AND P. DRUSCHEL AND D. BHATTACHARJEE. Measurement and Analysis of Online Social Networks. In *ACM SIGCOMM IMC* (2007).

[4] CLOJURE. http://clojure.org/.

[5] DAN LARKIN. Json encoder/parser for clojure. https://github.com/danlarkin/clojure-json.

[6] DIMACS. 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/~challenge9/download.shtml.

[7] FACEBOOK. Facebook's memcached multiget hole: More machines = more capacity. In *http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more- machinesmore-capacit.html* (2009).

[8] FRANZ INC. Rdf graph database. http://www.franz.com/agraph/allegrograph/.

[9] G. KOSSINETS AND J. KLEINBERG AND D. WATTS. The Structure of Information Pathways in a Social Communication Network. In *ACM KDD* (2008).

[10] GUY, R. G., HEIDEMANN, J. S., MAK, W., JR., T. W. P., POPEK, G. J., AND ROTHMEIER, D. Implementation of the ficus replicated file system. In *USENIX* (1990).

[11] H. CHUN AND H. KWAK AND Y. EOM AND Y. AHN AND S. MOON AND H. JEONG. Comparison of Online Social elations in Volume vs Interaction: A case Study of Cyworld. In *ACM SIGCOMM IMC* (2008).

[12] HART, P. E., NILSSON, N. J., AND RAPHAEL, B. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *SIGART Newsletter 37* (1972), 28029.

[13] IAN CLARKE. A Distributed Decentralised Information Storage and Retrieval System, 1999.

[14] INFINITEGRAPH: DISTRIBUTED GRAPH DATABASE. www.infinitegraph.com/.

[15] INFOGRID: WEB GRAPH DATABASE. http://infogrid.org/.

[16] J. LESKOVEC AND K.J.LANG AND A. DASGUPTA AND M. W. MAHONEY. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-defined Clusters. In *CoRR, abs/0810.1355* (2008).

[17] JSON. http://www.json.org/.

[18] KOBRIX SOFTWARE. Directed hypergraph database. http://www.hypergraphdb.org/index.

[19] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review 44-2* (2010).

[20] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *PODC* (2009).

[21] MICROSOFT RESEARCH. Trinity project: Distributed graph database. http://research.microsoft.com/en-us/projects/trinity/.

[22] NEO TECHNOLOGY. Java graph database. http://neo4j.org/.

[23] ORACLE. http://www.oracle.com/index.html.

[24] POSTGRESQL. http://www.postgresql.org/.

[25] PUJOL, J. M., ERRAMILLI, V., SIGANOS, G., YANG, X., LAOUTARIS, N., CHHABRA, P., AND RODRIGUEZ, P. The little engine(s) that could: Scaling online social networks. In *SIGCOMM* (2010).

[26] R. PEARCE AND M. GOKHALE AND N.M. AMATO. ultithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *IEEE International Conference on High Performance Computing, Networking, Storage and Analysis* (2010).

[27] REDIS. http://redis.io/.

[28] ROBERTO GONZALEZ AND RUBEN CUEVAS AND CARMEN GUERRERO AND ANGEL CUEVAS. Where are my followers? understanding the locality effect in twitter. *arXiv:1105.3682v1* (2011).

[29] ROGER L. HASKIN AND RAYMOND A. LORIE. On extending the functions of a relational database system. In *ACM SIGMOD* (1984).

[30] SATYANARAYANAN, M. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39* (1990), 447–459.

[31] STONEBRAKER, M., AND ROWE, L. The Design of POSTGRES. *Technical Report, UC Berkeley November* (1985).

[32] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP* (1995).

[33] U. KANG AND CHARALAMPOS E. TSOURAKAKIS AND CHRISTOS FALOUTSOS. Pegasus: A peta-scale graph mining system. In *IEEE International Conference on Data Mining* (2009).