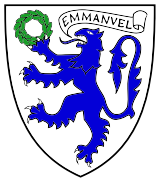
UNIVERSITY OF
CAMBRIDGE

Grapht: A Hybrid Database System for Flexible Retrieval of Graph-structured Data

Christopher J. O. Little



Emmanuel College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Engineering in Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: cl554@cam.ac.uk

May 31, 2016

Declaration

I, Christopher J. O. Little of Emmanuel College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11438

Signed:

Date:

Abstract

There has recently been an increasing need for fast analysis of graph-structured data, which has led to the development of several graph-centric alternatives to traditional relational databases. Although these ensure the fast execution of queries which fit within this graph-centric model, inevitably a compromise has been made, and other queries perform less efficiently than they would have done with a relational database. I propose the development of a query language enabling intelligent dispatch of optimised queries either directly to a relational database, or through a graph-centric query processing layer.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph databases are an alternative to traditional Relational Database Systems (RDBMS), which improve access speeds for some queries by prioritising access to related entities, rather than expecting traversal through a fixed schema. This design can more naturally represent relationships between data in many fields, from the analysis of gene networks in biology to social network recommendation engines. It can provide huge performance improvements for graph-centric queries, such as shortest path queries.

For other queries, however – such as aggregation of otherwise unrelated data items – it is far more efficient to rely on a known schema over a fixed set of rows for fast access. There are many trade-offs to be made between the RDMBS and Graph Database models, but the core division is that where a graph database prioritises access to topological information, an RDBMS equally prioritises attributes. In practice, however, although it is certainly the case that some queries benefit more than others from this prioritisation, it is rarely the case today for a business' operations to rely on a single type of query.

Road networks are often used as an important example of a highly connected dataset. However the reality is that, although relationships between junctions can be well modelled as a graph, this is not the most efficient data format for

all queries. Calculating the shortest path between two vertices, for example, will often perform faster using a graph database, since much of the graph can be ignored during the search, where a relational database must consider each edge equally. On the other hand, performing queries which concern all junctions equally – such as finding some point of interest in the network – is much more efficient using a relational database with a fixed schema, since computation can rapidly jump through memory using known intervals to retrieve desired attributes. This heterogeneity of possible query workloads, coupled with the common uncertainty at design-time about which part of an application will be most frequently used, makes it difficult to choose an ideal database system for any particular application.

I will return to this example of a road network throughout this paper, as it provides helpful explanation for the projects' motivation. A company building a navigation application, for example, may be wary of relational databases due to the risk of of high execution times for topology-focussed queries such as pathfinding. Users also need to be able to search for points of interest in the map. Since the searches may contain errors, the company cannot simply use the search terms as indexes for fast information retrieval, but must instead perform a string comparison against every point individually. For this kind of attribute-focussed workload, a graph database would perform poorly. The most common kind of user query, however, is likely to be of the form "What is the shortest path to get to 'Emmanuel College'?" This query has both attribute-focussed and topology-focussed components, which will run into problems using either of the approaches currently commercially available. Grapht aims to provide a middle ground between the two which avoids these bottlenecks.

1.1 Research Goals

The optimisations made to increase the performance of graph databases for certain problem necessarily sacrifice performance for others. Thus it seems

optimistic to aim to create a system which performs both as well as production graph databases for topology-focussed problems *and* as well as PostgreSQL for attribute-focussed problems. Instead, the aims for this project were to primarily improve on the worst-case performance of these systems, to provide a more homogeneous performance profile across both types of query. By doing this, we can provide a system which provides a higher average-case performance in situations where graph-centric queries and row-centric queries are both used. More concretely, there were two main goals for this project. Note that for this research project, we limit our attention to data-retrieval queries only.

1: Increase performance of hybrid queries compared to both relational and graph databases

Some queries have multiple components to them, some of which would best be handled by a graph engine, while others best by a relational system. As discussed previously, we may wish to identify some junction by name, and find a path to it through a road network. Under either a relational or graph database, some component of the query would necessarily perform badly and cause a performance bottleneck. A hybrid system ought to improve on these worst-case performances and not be affected so badly by either bottleneck. In so doing, it may thus outperform both systems for this combination queries.

2: Expose a coherent interface for this hybrid system

Relational databases have a well-established query interface in the form of *SQL*. Queries in SQL aim to filter out particular rows from a global set, rather than limiting attention to a connected subset of entities, making it difficult for a programmer to even express graph-centric queries. Although the graph database community have not yet settled on a single query language to act as SQL's graph-centric counterpart, the two most popular languages – Gremlin and Cypher – suffer from the opposite problem: they are not well suited to

expressing row-centric queries which consider all vertices in the graph. Since a hybrid system must aim to be equally well suited in terms of performance to either sort of query, it should not give precedence to either in terms of interface. Graph queries should be as easy to express using the hybrid system as row-centric ones.

In the next chapter, I will first give a brief introduction to the technologies involved in both relational and graph databases, including their relative strengths and weaknesses. I will continue by providing a brief overview of the surrounding research landscape, particularly focussing on previous attempts to bring together graph and relational systems. Chapter 3 discusses the implementation of a research prototype aiming to satisfy both of my research goals: *Grapht*. The success of this prototype is experimentally evaluated in Chapter 4, along with direct comparisons between Neo4J and PostgreSQL. These comparisons serve not only to provide a baseline against which to measure Grapht's performance, but also to validate the claims made above about the relative performance of the two systems in satisfying different types of query. Finally, Chapter 5 summarises the findings made, and discusses possible directions for future research.

Chapter 2

Background

2.1 Relational Databases

The relational database model first came to the fore in in the 1970s with Codd's famous *Relational Model of Data* [1]. The simplicity of this model brought immediate success, and it quickly found favour in industry at the time. Since then, the model has been carefully refined, with numerous implementations each offering incremental optimisations.

Along with simplicity, most RDBMSs provide desirable guarantees such as atomicity, consistency, isolation and durability (ACID). These properties were formally defined by Gray [3], and are now highly prized in a database system, since they give a high degree of confidence in the integrity of data, and allow database queries to be parallelised and distributed across several machines without risk of conflict between multiple concurrent queries.

These factors have lead to widespread adoption of the relational database model, and a corresponding proliferation of implementations, including MySQL, SQL Server and PostgreSQL. I will be evaluating PostgreSQL in this report, and using it as a base RDBMS upon which to build a graph-centric hybrid store. PostgreSQL is a popular, open-source database system supporting most of the SQL standard. It also supports a procedural language called

PL/pgSQL which extends SQL to offer more programming control. In particular, PostgreSQL supports recursive SQL queries which are crucial for efficiently finding paths through an edge relation without performing exhaustive `JOIN`s.

2.2 Graph Databases

Although graph databases have enjoyed a recent resurgence in popularity, the principles themselves are not much younger than Codd's relational database model. Indeed, both models aim to represent relationships between entities with only slightly different focusses. Chen [2] provided one of the earliest incarnations in the form of his *Entity-relationship Model.* This found popularity alongside the development of object-oriented programming in the 1980s, and several variations to this model were presented at the time [7][4]. Kunii [5] presented the first properly-called graph database model, called *G-Base.* Interest waned in the late 90s for a variety of reasons, including a general move by the database research community towards semistructured data. Recent times have seen a resurgence in popularity, as alternative strategies are explored to handle the "Big Data" explosion in data size and complexity.

The need for graph representations of data can be seen through the number of large technology companies who have recently developed in-house solutions to the problem. In 2010, Facebook released Open Graph, and Twitter released FlockDB[15], both projects aiming to make traversal through the social networks more efficient. In 2012 Google revealed their Knowledge Graph [10], aiming to connect search terms to real-world entities. The benefit of these graph databases is that queries can be localised to a particular area within the graph. Without direct links between related entities, every entity in the database would need to be considered to determine whether it is related to another. This approach is inefficient, particularly as the size of a database grows, and the cost of traversing the entire store increases.

It is important to note that graph database systems are different to graph pro-

cessing frameworks such as Ligra [12] or Pregel [9]. Where a graph database system is concerned with fast and frequent storage and retrieval of graph entities, a graph processing system will typically perform more complex analysis and transformations of the graph, often in a lower volume. The distinction is similar to that between Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) systems. An example query for a graph database may be to find a route between two points in a graph, but an example query for a graph processing system might be to calculate pagerank for the network. This is an important distinction for our purposes, as the systems we discuss here are concerned mainly with data retrieval.

Some well-known graph databases include Titan, OrientDB and Neo4J. McColl et al. [13] provide a recent overview of these and others, as well as a performance comparison. In this paper, I restrict my attention to Neo4J. Neo4J is a schema-less database based on a key-value store, which runs on the JVm and presents a Java API.

2.3 Related Work

The incumbency of relational databases coupled with the potential scalability of graph databases naturally prompts the question of whether it is possible to efficiently store and query graph structured data in a relational database. Indeed, several efforts have been made in this direction.

Object-relational mapping (ORM) frameworks such as Hibernate ORM for Java allow programmers to map an object-oriented domain model to a relational database. This allows semantic links to be easily followed from one entity to another by abstracting away from the underlying RDBMS. Although ORM frameworks allow programmers to feel as though they are using a graph database, they do not tend to focus on bringing any of the performance or scalability advantages of graph databases to the relational world.

Another approach is taken by OQGraph[**oqgraph**], which provides a plugin for MariaDB and MySQL. Users query a proxy table, which OQGraph

interprets as graph traversal instructions by imitating a storage engine. Although this somewhat improves the programmer experience by providing a more graph-centric interface, performance is not comparable to pure graph databases. Additionally, since the plugin imitates a storage engine, no parser extensions are possible and queries must be expressed in plain SQL. Although this means that no new query language need be learnt, it also means that graph-centric queries are awkward to express.

Biskup et al. [6] described in 1990 a SQL extension which could be used to query graph relations. No implementation is provided, and again, no performance considerations are made, but the work nonetheless provides a valuable inspiration for Grapht's query language *gSQL*. Biskup's work is further discussed alonside gSQL in Chapter 3.

A more performance-oriented approach to bringing relational and graph databases together can be found in Sun et al.'s [14] SQLGraph. This approach uses relational storage for adjacency information, and JSON storage for vertex and edge attributes. Although this approach does improve performance for graph-centric queries, moving attribute information to JSON files hurts performance of normal relational queries, which rely on fast and local access to attribute data to filter out rows.

Most recently, a Spark package has been developed called *GraphFrames* which extends the DataFrames library to handle graph data[**graphframes**]. DataFrames are a way of abstracting any underlying data source by loading it into a Spark *Resilient Distributed Dataset* (RDD). The GraphFrames extension extends these RDDs by allowing them to be interpreted as graph adjacency tables, and providing graph traversal methods. The query language provided for GraphFrames is somewhat limited, however, primarily designed for simple pattern-matching. It is not possible to guide the search direction through the graph, which is an ability that Grapht does have. Unlike Grapht, by building directly on top of Spark, GraphFrames queries are easy to distribute and parallelise across a number of machines.

Finally, my work also takes inspiration from Yoneki et al.'s *Crackle* [11],

which improves graph performance of a relational store by maintaining a limited graph store in memory, which is periodically filled using graph-aware prefetchers to ensure topologically local vertices are preserved in the store. Crackle does not support pure-relational or hybrid queries, however, nor does it attempt to present a declarative query interface in the way that Grapht does. We replicate and extend some of the Crackle results in Chapter 4.

Chapter 3

Implementation

In aiming to produce a hybrid system somehow equally well suited to both relational and graph-centric problems, two approaches are possible. The first would be to create a novel system from scratch which fully commits to neither underlying storage strategy; following instead some middle path to achieve optimality. A second strategy – and the one employed here in the design of Grapht – is to extend the capabilities of one system to mitigate its weaknesses by taking inspiration from the other system. In particular, Grapht extends a durable relational data store with a graph-centric in-memory store.

In this chapter, I first give a brief introduction to the different components involved in resolving a query through Grapht. Following that, I describe the two main components in more detail: the graph store prefetcher and the query processor.

3.1 System Overview

As mentioned earlier, Grapht extends an RDBMS – a graph query layer is "grafted" onto an underlying RDBMS. The underlying RDBMS should contain two relations in particular: one representing vertices in a graph, and one representing edges, with endpoints represented by keys to the vertex relation.

This data layout is already a common way to describe a graph in a relational environment, with self-`JOIN` queries on the edge relation used to explore paths consisting of multiple hops. In this way, Grapht often requires no change to the data format, and can simply be added to a relational database with very little work. Internally, Grapht is made up of two parts: a row-centric and a graph-centric handler. Both components communicate directly (and separately) with the underlying RDBMS via standard SQL queries. This allows any SQL-compatible RDBMS to be used fairly indiscriminately.

In addition to this, Grapht also includes a query processor, allowing simple queries to be expressed in a syntax similar to SQL, but harnessing the additional power of the graph-centric handler. Graph algorithms and queries vary widely in requirements, such that it would be difficult to produce a single query language able to express the full spectrum of possible queries. full diversity using a single query language. For this reason, the query processor is only designed the be used as a lightweight complement to the Grapht API directly.

(TODO: Diagram of main components)

The Grapht API has a relatively small interface. Method invocations to the API are for the most part handled exclusively by either the row-centric handler or the graph-centric handler, allowing each to be highly optimised for the task at hand. A few higher-level methods then allow client applications to combine the results of graph and row-centric queries. The *row-centric handler* performs relatively little work itself, relying instead on the decades of optimisations made in relational databases to efficiently retrieve data by passing queries directly on to the underlying RDBMS.

Within the *graph handler*, data is laid out in memory to optimise graph-centric access patterns. The handler contains an in-memory store which is by design of limited size so that Grapht can be used on systems even with small amounts of memory available. When a vertex is requested through the API which is not available in the store, the vertex data is obtained from the relational database. At the same time, a prefetcher is used to populate

the store with adjacent vertices, so that subsequent requests can hope to be fulfilled directly from the store.

Finally, the *query processor* receives string queries from a client application and breaks them apart into row-centric subqueries targeting the relational database directly, and graph-centric subqueries targeting the graph handler. Client queries are written using an extension of SQL I call *gSQL* (as in *graph*-SQL) inspired by Biskup et al's *Graph Manipulation Kernel*[6]. The extension is a strict superset of SQL, allowing pure-SQL subqueries to be passed on directly to the underlying RDBMS. Pure-graph queries end up being handled exclusively by the graph handler, and hybrid queries are broken apart and subqueries dispatched appropriately.

3.2 Row Handler

As mentioned above, the row-centric handler performs relatively little work. Instead, queries are passed unchanged to the underlying RDBMS, and the resulting relations are passed back. One optimisiation is that queries to the row-handler are lazily evaluated by default. This allows queries to be transformed or dropped altogether when doing so would be advantageous. One situation in which this may arise is as part of a more complex API method call. The method may need to join together results from a graph query and a row-centric query. If the graph query returns an empty result set, the row query does not need to be evaluated – the joined result will itself be empty – so the row query may be dropped altogether.

3.3 Graph Handler

The graph-centric handler is slightly more involved than the row-handler. It contains an in-memory store, in which vertices are looked up before falling back to the underlying RDBMS. Internally, the store maintains an indexed

list of vertices in memory. Each of these vertices contains a list of edges, allowing for quick traversal without needing to pass through an index again. If the full graph could be loaded into memory, these edges would contain a pointer directly to the target vertex, to avoid indirection through the vertex index. In practice however, we do not expect to be able to always be able to contain the entire graph in memory, and instead the edges contain a key to the vertex index, representing a vertex which may not yet be loaded into the store.

When a vertex is requested through the handler, the store's vertex index is first checked to try to fulfill the request. If possible, the vertex is returned directly. If this is not possible, the graph handler queries the underlying RDBMS to obtain the vertex data. At the same time, a prefetcher is used to obtain data for related edges and vertices. The requested vertex data is immediately returned to the user and the prefetched related data is added to a work queue. This work queue is processed in a separate background process which is responsible for adding new items to the store, and removing them when the size of the store exceeds the memory allowed by the current Java virtual machine (JVM). When the size of the store does exceed the allowed size, vertices are removed from the index according to a least-recently used strategy. Each time a vertex is requested, the access is logged, and the vertex with the least recent access time is chosen when a vertex must be eliminated from the index. The idea behind this is that vertices which have been recently visited by the query processor are likely to be visited again soon in the future, since the traversal of the graph will remain local to the most recent vertices. Thus these vertices should be kept in the store, in preference to vertices which were only visited a long time ago.

The performance of two different strategies (*lookahead* and *block*) for prefetching was analysed in detail by Yoneki et al. [11], but I have here repeated and extended these results with a third strategy.

3.3.1 Lookahead Prefetcher

The first strategy examined by Yoneki et al. simply performed a limited depth-first traversal of the graph, starting from the requested vertex. This traversal was performed using a recursive "Common Table Expression" (CTE). These types of query allow a result set to be built up by recursively joining a partial result set with another relation. As the lookahead depth increases, the size of the partial result set grows exponentially, such that the cost of joining becomes itself exponentially larger. For this reason, Yoneki identified that after a certain depth, the cost of the prefetch procedure grows larger than the avoided overhead. I replicated this finding, which is further discussed in Chapter 4.

3.3.2 Block Prefetcher

The second strategy used by Yoneki et al. exploited the fact that there is often a strong correlation between certain vertex attributes and graph locality. For example, in a road network, the latitude and longitude of junctions can be expected to correlate with graph locality, since nearby junctions are far more likely to be connected by a road than distant ones. Yoneki terms this "semantic" locality, and uses it to quickly prefetch large blocks of the graph by simply filtering on latitude and longitude. The cost of this procedure does not grow as fast as for the lookahead prefetcher: for a block of size b, we would expect the cost of fetching all junctions within the block to be proportional to the number of junctions within the block – i.e. the cost will be $O(b^2)$. Although this is advantageous from a performance perspective, it places an important limitation on the data – namely that there must exist some attribute which is tightly correlated with graph locality.

3.3.3 Iterative Lookahead Prefetcher

I examined a third possible strategy for this report, inspired by the first two. The lookahead prefetcher is most portable, since it can be indiscriminately applied to any graph. However, it does not scale as well, such that when the store is large, and it would be advantageous to prefetch many vertices in one go, the performance cost is too high, and a block prefetcher becomes more effective. The source of this cost penalty is that many edges are unnecessarily traversed repeatedly. For example, after one level of recursion, edges one hop away from the source are traversed, and their destinations included within the CTE table. After five levels of recursion, we should only be examining edges which are precisely five hops away from the source. However the recursive CTE does not distinguish between the edges which were added at the previous level of recursion and those which were added at the start of the procedure (and which do not need re-examination).

We can improve performance, then, by only expanding the fringes of the search – and even then only those fringes which have not been previously examined. To achieve this, I implemented a slightly more involved prefetcher, which maintains for each request a set of vertices explored this time. It then performs a number of iterations – one for each level of lookahead – requesting just those vertices which were added to the fringe during the previous iteration. Requests look like this:

```
SELECT *
FROM edges
JOIN vertices
   ON vertices.id = edges.from_id
WHERE edges.from_id IN (1286, 1373, 1141, ...)
```

These operations are very fast, since the `from_id` field of the edge relation is indexed such that retrieval is a constant time operation. This type of indexing is not possible for the recursive CTE table. The trade-off here is that a large number of requests are made to the RDBMS, such that the communication and query-parsing latency are incurred repeatedly. However,

as is discussed in Chapter 4, experiments showed that in practice this latency was comparable the join cost of the CTE approach, for much larger lookahead depths. In fact, this iterative lookahead approach allowed lookaheads of almost ten times the optimal depth of the original lookahead prefetcher for the same overall performance cost.

Despite this, a difference was encountered in that the optimal depth for a query was much more variable than for the CTE lookahead prefetcher. It is likely that this optimality will vary based on factors such as store and graph size as well as the degree of interconnectedness of the graph (since a graph with many high-degree vertices will have a faster-growing fringe during exploration). It was also found that the performance of the lookahead prefetcher was higher than the block prefetcher, which contradicted earlier results. As a result of these findings, a CTE lookahead prefetcher was used as the default Grapht prefetcher with a depth of five hops.

3.4 Grapht API

The Grapht API allows client applications to query the underlying RDBMS, while taking advantage of the row and graph-centric handlers to optimise performance according to the expected access patterns. There is one API method targeting the row-centric handler directly: `getRelation(sql)`. This method takes a string parameter `sql` representing a standard SQL query, and returns an iterable list of rows representing the result relation. In order to target the graph-centric handler, one must first initialise the graph store with a particular source. This is done by using the `createGraph(vertices, vertexKey, edges, edgeKey, sourceKey, targetKey)` method, which takes four string arguments. The arguments `vertices` and `edges` correspond to SQL queries representing the edge and vertex relations respectively. Normally, these strings will simply be the names of the tables corresponding to edges and vertices in the underlying store, but it is also possible to pass in a SQL query which will dynamically create an edgelist. The remaining

arguments, `vertexKey`, `edgeKey`, `sourceKey` and `targetKey` are used to determine which fields of the chosen relations correspond to vertex, edge, source and destination identifiers respectively.

Once the graph is initialised, two simple methods target the graph-centric handler directly to retrieve the relevant entities from the graph according to a given ID: `getVertex(id), getEdge(id)`. The objects returned by these method calls have accessor methods of their own. Vertex objects have access to the `getEdgeList` method to obtain a list of outgoing edges, and Edge objects have access to a `getTargetVertex` method to obtain a reference to the target vertex of this edge. As mentioned earlier, vertices are not guaranteed to be present in the graph store when they are requested, as they may have been displaced by the LRU policy. A request to an absent vertex will trigger a fetch from the RDBMS. On the other hand, edges will always be present in the store as long as their source vertex is. Finally, both Edge and Vertex objects share a `getAttribute` method, used for retrieving arbitrary data stored alongside the edge. This type of attribute accessor is common in graph databases where attributes are treated as having only secondary importance. This is tolerable here since queries where attributes are the priority can often pass through the row handler. Between these five methods, it is possible to traverse the entire graph efficiently, starting and ending at any arbitrary vertex, and collecting any desired attributes along the way.

Almost any algorithm can be efficiently expressed using this interface for graph traversal and retrieval of attributes. For convenience, a few further methods are provided, which use the methods described previously for themselves. The most powerful of these is the `getPaths` method, which is used to discover paths branching out from a source vertex. The design of this method is closely tied to the design of the query processor (discussed in the next section), since this method provides the bulk of the implementation for the query processor. However, a client application can also call `getPaths` independently.

(TODO: Diagram)

The operation of `getPaths` essentially follows a best-first exploration of the graph starting from the source vertex. To achieve this, a simple priority queue of partially-discovered paths is used. The priority of a vertex is determined by a user-provided *prioritiser* function. If no prioritisation strategy is specified, an index is maintained such that all elements inserted have a higher index than all those previously inserted. By using this index as priority, the queue devolves into a last-in, first-out stack as and search proceeds as a depth-first search. Although this is less efficient than a native stack, since insertion becomes an $O(logn)$ operation compared to $O(n)$, using a priority queue in this way offers more flexibility to programmers.

Rather than provide a list of edges and vertices as output, a collection of *accumulators* are provided to `getPaths`. These are in effect stateful functions, which will receive new data at each step of traversal, and update their internal state accordingly. For example, a `count` accumulator could be provided, which simply increments its state with each new vertex encountered. Alternatively, a `sum` accumulator could be provided which will extract some attribute from the edges traversed, and add the values together.

Along with these accumulators, *evaluator* functions can be provided, which determine whether the state of the accumulator is such that the current path should be saved for inclusion in the result set. This allows us to, for example, limit traversal to a depth of 3 by using a `count` accumulator in conjunction with an evaluator which rejects paths where `count` > 3. An opportunity to cut the search space down is also provided to the user, who can optionally specify a rate of change for each accumulator, or failing this at least a direction of change. For example, a `count` accumulator is guaranteed to increase by one with each step through the graph, while a `sum` accumulator wil at least increase each step if the attribute it is accumulating is always positive. If an evaluator is provided which rejects sums greater than 100, then we do not need to continue considering a paths' possible expansions once it has already reached a sum of 100, since longer paths will never have a smaller sum.

Two boolean values may also be provided, specifying that only unique edges

or vertices be used along the path. This is determined by maintaining a "closed" set of already-visited entities for each path. If a vertex or edge which is already in a closed set is expanded, it is skipped rather than being added to the queue as a new partial path. Finally, a *limit* parameter is used to determine how many valid paths should be retrieved before returning. By default, the limit is unset, and the search for valid paths will be exhaustive.

To summarise, then, `getPaths` receives as mandatory input a graph and start vertex, and the following optional parameters:

- A *prioritiser* function to guide traversal
- A list of vertex *accumulators* (optionally with a rate of change specified)
- A list of edge *accumulators* (optionally with a rate of change specified)
- A list of *evaluators* to determine valid paths
- Two boolean values specifying whether vertices and edges must be unique along a path
- An integer *limit* on the number of paths retrieved

In the special case where a single path is requested, an important optimisation is possible. When multiple partial paths pass through the same vertex, we need only consider the single route with the highest priority. In highly interconnected graphs, this allows us to discard a large number of possible paths early on. In practice by doing this, and requesting only unique vertices, this optimisation allows the best-first traversal to degenerate into a variation on the A* search algorithm, as long as an admissible heuristic is used in the prioritiser function to guide traversal.

On each iteration of the traversal, a partial path is fetched from the priority queue. This path is considered to see if it satisfies the conditions required to be output. If it does, the path is saved, and we check whether the desired number of paths have been found, stopping execution if this is the case. If not, we consider all possible extensions of this path by iterating through the outgoing edges of the final vertex in the path. If it is possible that these

edges may lead to a valid solution (which can be determined by examining the attributes' rates of change, as discussed earlier), then the target vertices are fetched from the store. A new partial path is created, accumulating attributes from both the edge and the target vertex. A priority is calculated for this path according to the prioritiser function, and the new path is added to the queue, ready for selection in some future iteration.

The eventual output of `getPaths` is a list, where each entry contains the accumulated results of traversing a valid path from the source vertex. In effect, this result can be treated as a relational table, where each accumulator is a column in the relation. This allows paths through the graph to be manipulated just like any relation in the underlying RDBMS, and in particular allows the results of a path-finding query to be joined with a normal relation. This ability is a significant contribution of this work, since obtaining this kind of result efficiently is not possible in either a purely relational system (where the path-finding component would take too long), nor in a purely graph-oriented system (where there exists no notion of a "normal" relation).

Ideally, the result relation of the `getPaths` method would be created directly within the RDBMS, so that joins could literally be performed as between normal relations. The current Grapht prototype is not sufficiently tightly coupled enough to the PostgreSQL for this to be possible, however. As a workaround in the meantime, the results of the `getPaths` method can be loaded into the store after they are calculated as a temporary in-memory relation. This does carry an inevitable performance penalty compared to being constructed directly in PostgreSQL's memory space.

It may seem as though only providing a path-searching method does not provide a very powerful abstraction beyond the vertex and edge-querying methods provided. However, the aggregation mechanism means that we are able to consider attributes belonging to only vertices reachable from the source vertex. For example, we may wish to retrieve a list of all junctions reachable within 100km of some source junction. This can be achieved by using a `sum` accumulator as described above and a `last` accumulator, which

simply retrieves attributes from the *last* vertex along a path. An evaluator can then be attached to the `sum` accumulator, checking whether the total path distance is less than 100km. Since the sum is non-decreasing, an exhaustive search will terminate as paths will cease to be expanded after 100km. In this way, we can quickly retrieve a relation containing just the details of reachable nearby junctions using a single API method call.

3.5 Query Processor

The aim of the query processor is to provide some of the power of the Grapht API in a more lightweight manner, so that clients can simply express retrieval queries as strings, rather than needing to interact with a more low-level API. Biskup's Graph Manipulation Kernel[6] (GMK) unites relational and graph models in a way similar to what we would like to achieve. It does this by defining paths over graphs as relations with three implicit attributes: `START`, `GOAL` and `LENGTH`. By describing paths in this way, we write queries over the possible search space of paths through the graph, using a syntax very similar to SQL. For example, a "friend-of-friend"-style query to find vertices two hops away from the vertex with ID 1 would be written this way in Biskup's GMK:

```
SELECT *
    FROM PATHS OVER my_edge_relation
WHERE
    START = 1 AND
    LENGTH = 2
```

In this example, `my_edge_relation` is some relation with precisely two attributes which are interpreted as start and end identifiers of edges in a graph. If the relation contains more than two attributes, the attributes indicating start and end may be included in parentheses, as in "`... PATHS OVER my_edge_relation (id1, id2)`." Other attributes of the edge relation can then be aggregated along the path for inclusion in the output path rela-

tion. Although this provides a good starting point for Grapht, a few notable shortcomings prevent us from adopting the solution directly.

Firstly, Biskup's solution has no direct support for vertex attributes. Vertices are in fact not treated as first-class entities at all, simply being described implicitly as start and end attributes within the edge relation. It is possible to include vertex attributes in a query by joining the edge relation together with the vertex relation, however this results in duplication of vertex data when several edges lead to the same vertex, and can cause confusing "off-by-one" errors since any path will have one more vertex than it has edges.

Secondly, Biskup's extension leaves the traversal mechanism used to find valid paths unspecified. In many ways this is a good thing, and mimics the design of SQL itself. By ensuring that the query is fully declarative, an implementation can select any method to find possible paths, and the client application need not worry about efficiently finding solutions. In the absence of cycles in the graph, the set of paths discovered will be constant, although they may be discovered in a different order. This is no different to the way in which the order of rows retrieved by a SQL query is undefined unless an `ORDER BY` clause is present in the query. The difference is that the performance gain from using an appropriate traversal order through the graph is much more significant than the gains made by iterating through a set of rows in a different order. If we want to find a particular neighbour of a vertex, a depth-first search (DFS) might visit every single other vertex in the graph before returning, while a breadth-first search (BFS) can terminate almost immediately.

Adding cycles to the graph makes appropriate selection of a traversal order all the more critical, since DFS here may result in non-termination, while BFS may have no problems. A final related problem with GMK is that it is difficult to express uniqueness constraints on the vertices visited. By enforcing that the vertices visited along a path be unique, it becomes possible to traverse a graph with cycles without non-termination, so expressing these constraints is very valuable.

3.5.1 Improving on the Graph Manipulation Kernel

To solve the issue of vertex attributes, two small changes must be made. Firstly, defining a graph according to an edge relation is no longer sufficient, and graphs must now be specified as a pair consisting of an edge relation and a vertex relation. As before, we must specify the attributes to use as start and end points for the edge relation, and a vertex identifier for the vertex relation. We must also now change the syntax of calculated attributes of the output relation, to explicitly state whether values are being accumulated along the edges of the path, or along the vertices. For example, to accumulate the `name` attribute of vertices as a concatenated string, along with the total `cost` of edges along the path, a gSQL query may look like this:

```
SELECT START, END, LENGTH,
    (ACC VERTICES CONCAT(name, " -> ")) path,
    (ACC EDGES SUM(0, cost)) cost
FROM PATHS OVER (my_edge_relation(id1, id2), my_vertex_relation(id))
WHERE
    START = 1
```

In this way, the accumulation is made explicitly over vertices or edges (`ACC VERTICES` in the former case, `ACC EDGES` in the latter). Following the `ACC` keyword is the name of a function to use as an accumulator (`CONCAT` or `SUM` in the example). For the prototype version of Grapht, only a few functions were defined but others would be trivial to add. The `CONCAT` function, for example, takes as arguments the name of a attribute to accumulate at each step and a separator string, returning the concatenated value of all named attributes along the path.

To solve the remaining problems, I have added a `TRAVERSE` clause to the end of the query format, which allows users to specify a prioritisation order for traversing new vertices in search of a valid path. The clause is optional, and if omitted a breadth-first search is used. An example use of this clause would be:

```
SELECT *
FROM PATHS OVER (my_edge_relation(id1, id2), my_vertex_relation(id))
WHERE
    START = 1
TRAVERSE
    UNIQUE VERTICES
    BY -INDEX
```

This example specifies a breadth-first traversal, avoiding cycles by disallowing repeated vertices along the same path. The `UNIQUE` modifier is again optional, and may specify either "`VERTICES`", "`EDGES`" or "`VERTICES,EDGES`" if duplicate edges and vertices should both be avoided. "`INDEX`" refers to the insertion index within the internal priority queue discussed in Section 3.4. By ordering according to the negative index, we get a first-in, first-out queue, and hence follow a breadth-first search.

One limitation has been introduced to gSQL queries which was not present in GMK: that queries should always specify a start vertex for the search. This is included for performance reasons, following the implementation of the `getPaths` method of the Grapht API. In general it would be inefficient to consider every possible path through the graph when the query can be solved using a local traversal. It would be possible to reduce the search space without using a known start vertex (for example by working backwards from a known end vertex) but these strategies have not been implemented in the initial prototype, such that a start vertex must always be provided.

3.5.2 Executing gSQL

The main task of the query processor is to extract information from a gSQL query in order to call the `getPaths` method of the API. When a gSQL query is received, the first task is to separate components of the query into row-centric and graph-centric queries. To achieve this, the query processor examines the data aource specified in the query. If data is being queried from one of the relational tables, then that data is fetched through the row handler directly.

If data is being queried from a "PATHS OVER" clause, then the query will instead be resolved through the graph handler. This provides a very simple way to break apart a complex query and reliably identify what kind of access pattern is expected for the query.

When a query has been identified as targeting the graph handler, it is further broken down into actionable components. First, the data source is identified from the PATHS OVER clause. If the precise combination of edge and vertex relations has not been seen before, a new subgraph is initialised in the graph store to represent it. Next, accumulator functions are generated to collect the data required by the SELECT clause. Any accumulators specified in the query are created according to the parameters passed in the query. In addition to these, three accumulators are always implicitly created to represent the START, LENGTH and END attributes of the output relation. Finally, accumulators may also be defined within the WHERE clause, so that conditions can be checked at each step even if the result is not included in the final relation. This WHERE condition is evaluated at each vertex along a possible path.

The implicitly-created LENGTH accumulator is initialised with a known rate-of change, since LENGTH will increase by one with each hop. This allows queries with a bound on DEPTH in the WHERE clause to run exhaustively within this bound, since Grapht knows that a shorter LENGTH can never arise further along the path. Although this is a desirable behaviour, there is currently no way for query writers to specify that other accumulated variables have a known rate of change, although future versions of the query processor may provide functionality for this.

The prioritiser and uniqueness constraints for getPaths are very simply extracted from the TRAVERSE BY clause. Similarly, a limit to the number of paths to retrieve is read directly from the LIMIT clause of the query.

gSQL can also describe complex clauses which join together results from both the row and graph handlers. In these cases, a relational join can be performed as between normal relational tables. This is possible since the result of a "PATHS OVER" query (i.e. the output of a call to getPaths).

As mentioned in Section 3.4, the Grapht prototype loads path relations as temporary tables with PostgreSQL, to allow the RDBMS to perform the join natively.

Chapter 4

Evaluation

In this chapter, I provide experimental evidence for the claims made in earlier chapters. I will begin by describing my experimental setup, and outline the methods used for the subsequent tests. Three sections follow, each examining a different performance aspect. The first of these sections aims to replicate and extend the results of Yoneki et al[11] in optimal a prefetching strategy for Grapht. The next section examines the performance of certain queries across the three database engines under consideration: PostgreSQL, Grapht and Neo4J. This allows me both to confirm the premise motivating the development of Grapht – that there exist large performance gaps in both directions between PostgreSQL and Neo4J – and to show that the addition of an intermediate hybrid query processor can help to bridge this gap and improve average overall performance. Finally, the last section includes a qualitative examination of the expressiveness of the three query systems, both in terms in terms of the declarative query language presented, and in terms the direct API offered where one is available.

4.1 Test Framework

Before undertaking any tests, an experimental hypothesis was chosen to direct the direction of my investigations. These hypotheses are outlined in the appropriate sections. To obtain a measure for query performance, a Scala benchmarking program was written, which recorded the start and end time of operations and used these to calculate the average execution time for each operation. Each query was performed ten times, with the fastest and slowest results discarded as outliers. Unless otherwise stated, 100 queries of approximately the same size were randomly chosen for each test, to prevent the particularities of any single query from impacting the overall results. For PostgreSQL and Neo4J, a cache-warming query was performed before each test by iterating through every vertex in the graph. This was not performed for Grapht, since such a cache-warming query would cause the prefetcher to load much of the graph into memory, making the presence of the prefetcher redundant by the time the query is performed. This disparity is tolerable, since we nonetheless will see a conservative estimate for Grapht's performance with respect to our performance hypotheses.

The same dataset was pre-loaded into both PostgreSQL and Neo4J, consisting of a DIMACS dataset representing the U.S. road network[8]. This graph contained 264,346 vertices and 733,846 edges. Both edges and vertices were then augmented with a "`payload`" attribute calculated as the MD5 hash of the entity ID. This was done to ensure that the database contained more than simple topological information, as is likely to be the case for many practical applications of graph databases. In PostgreSQL, the graph was defined by two relations, as shown in Table **??**. In Neo4J, the graph was defined according to the standard graph format, with an index created on vertex IDs, and attributes added to vertices and edges as entity *properties*.

All experiments were carried out using Ubuntu 16.04 running on a 3.4 GHz, 64-bit 4-core Intel Core i7 processor (2nd generation), with 8 GB of RAM. JVM v1.8 was used, with the maximum and minimum heap size set to 1024MB by default, to avoid wasting time trying to acquire more mem-

ory from the system. The JVM has global pauses for garbage collection, so a manual garbage collection was requested before each test, to try and ensure that any pauses were at fault of the current test itself. Note that in figures presented in the following sections, a logarithmic scale has been used for clarity, since there are often performance differences of several orders of magnitude between the systems.

4.2 Prefetcher Performance

The first task was to select a prefetching strategy to use in Grapht. For this evaluation, I examined the prefetchers described in section 3.3. In evaluating prefetcher performance, the initial expectation was to find that the peak performance would come from prefetching with large blocks rather than using a lookahead strategy – as reported by Yoneki et al. In addition, where the original CTE prefetcher suffered as lookahead depth increased, I expected to find that my iterative lookahead prefetcher would perform better by performing cheaper queries at large depths.

To test the effects of changing prefetching strategies, I measured the execution time for an A* search using each prefetcher. I consider here separately five different query lengths, in order to see whether prefetching strategies were particularly good or bad when examining larger portions of the graph. Paths consisting of 20, 40, 60, 80 and 100 hops were each considered. Twenty queries were selected for each path length by randomly selecting start and end points within a graph neighbourhood, and performing a shortest-path calculation to select vertices for which the shortest path had the desired length. The size of the subgraph explored by the A* algorithm for each of these paths varied largely by query, but was always 80-100 times the length of the path, such that subgraphs containing up to 12,000 vertices were examined.

Figure 4.1 shows an analysis of the effect of changing the depth of the CTE lookahead prefetcher. As expected, we see that for all path lengths performance initially improves as we increase path length, until a lookahead depth

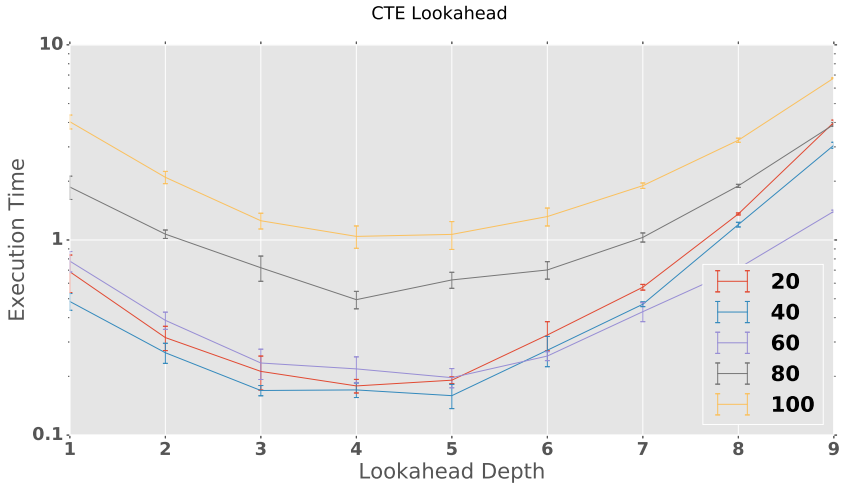


Figure 4.1: Average Execution time for A* search of different path lengths using a CTE lookahead prefetcher

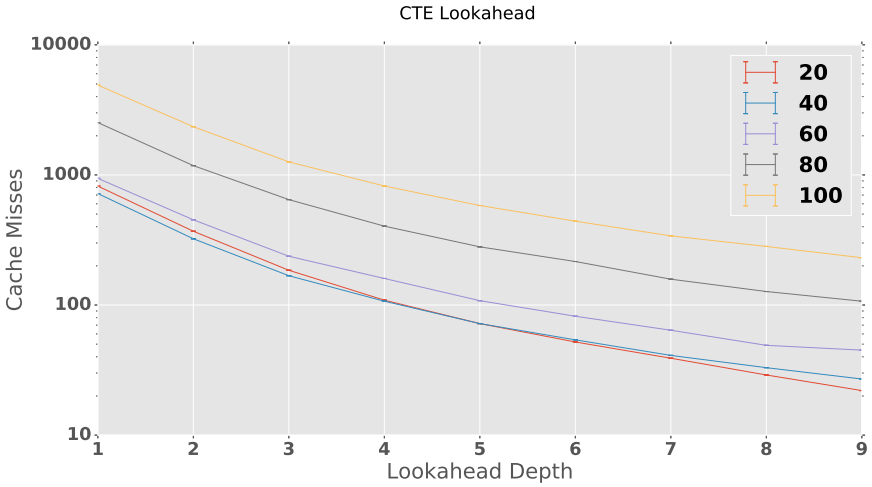


Figure 4.2: Graph miss rate for A* search of different path lengths using a CTE lookahead prefetcher

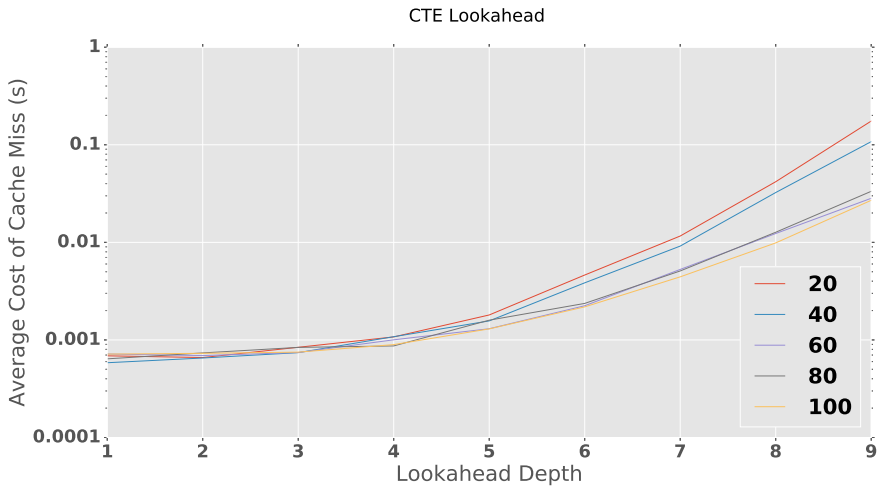


Figure 4.3: Average cost of store miss for A* search of different path lengths using a CTE lookahead prefetcher

of around 4 to 5. At this point, execution time begins to increase again. We can gain a better causal understanding of this behaviour by examining the number of vertices needing to be fetched from the database for each query (the store *miss rate*). We see from Figure 4.2 that the miss rate decreases exponentially as lookahead depth increases. This is expected since more vertices are fetched with each miss. The overall execution time thus initially decreases with depth. In contrast, the average cost of each store miss increases more than exponentially (Figure 4.3). This cost quickly overcomes the benefit from the reduced number of graph misses, and becomes the dominating factor as lookahead depth increases.

For the iterative lookahead strategy, we see that the average cost of each graph miss (Figure 4.6) is initially higher than for the CTE strategy. This is due to the fact that several queries are performed at each iteration instead of a single more complex query. However the average cost using the iterative approach grows far more slowly with lookahead depth, allowing lookahead depths around five times larger than the CTE approach to be used before performance starts to degrade. This larger lookahead depth in turn causes a much lower store miss rate. Indeed, in all of the examined queries, the lookahead depth was allowed to exceed the path size, meaning that the store

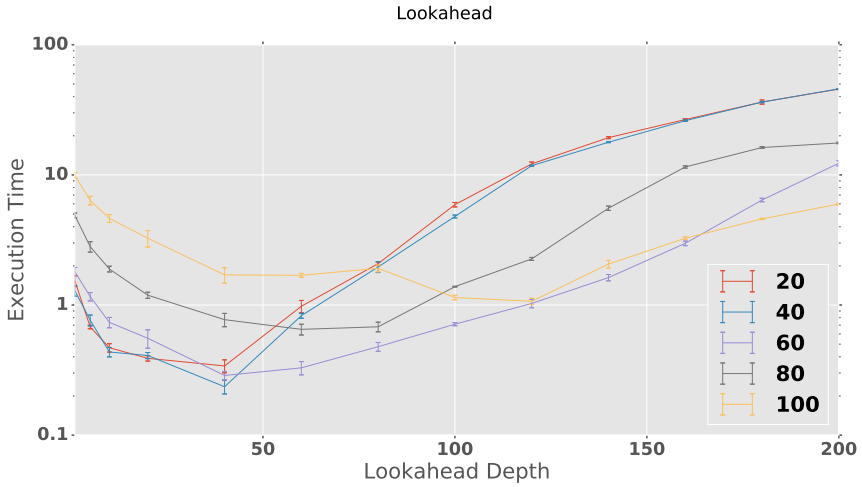


Figure 4.4: Average Execution time for A* search of different path lengths using an iterative lookahead prefetcher

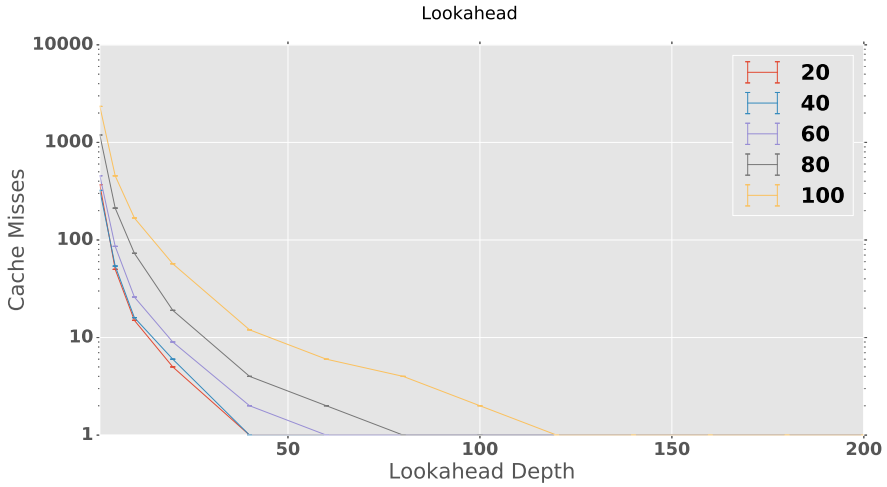


Figure 4.5: Graph miss rate for A* search of different path lengths using an iterative lookahead prefetcher

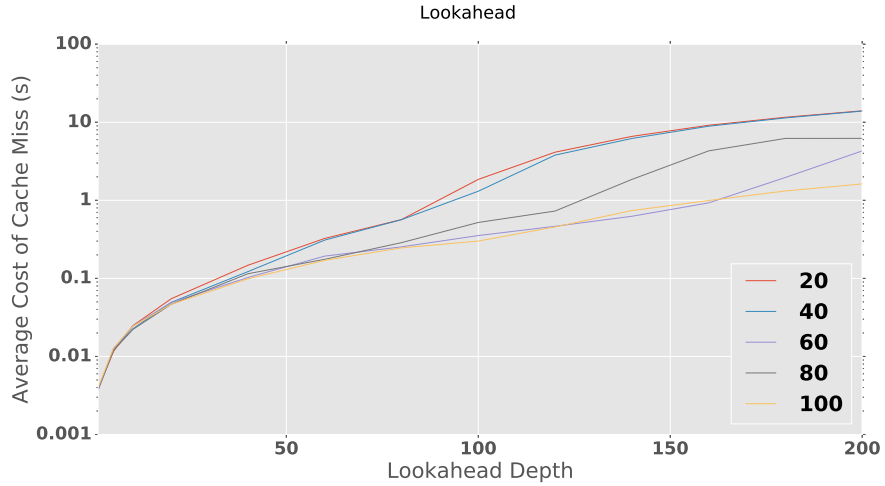


Figure 4.6: Average cost of store miss for A* search of different path lengths using an iterative lookahead prefetcher

miss rate effectively settled at 1 (when only the source vertex misses). The effect of these two factors on overall query time is that the total time taken fetching vertices from the database decreases steadily as the store miss rate decreases. As soon as the graph miss rate reaches 1, however, the overall time taken increases with lookahead depth. This happens because additional iterations only fetch vertices which will not be used, since they did not need fetching at lower lookahead depths. Although the peak performance of this approach is similar to the CTE lookahead strategy for A* search, the overall performance characteristics are quite different. It seems likely that the vastly increased lookahead depth would improve performance for other types of queries where the graph miss rate is never allowed to reach 1.

Finally, we examine the effects on performance of using the *block* prefetcher (Figure 4.7). Predictably, the overall execution time of A* reduces as the block size increases. This happens because the average cost of a graph miss remains roughly constant as block size increases (Figure 4.9), but more vertices are added to the graph store each time. This reduces the graph miss rate (Figure 4.8), which causes a corresponding decrease in execution time. However, I found that this reduction slows after a block size of around 15000, with an overall execution time of between one and ten seconds. Since both

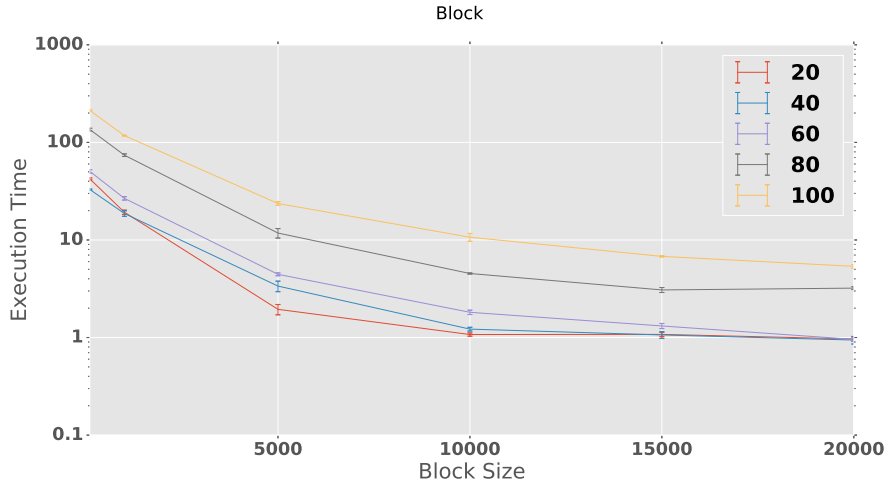


Figure 4.7: Average Execution time for A* search of different path lengths using a block prefetcher

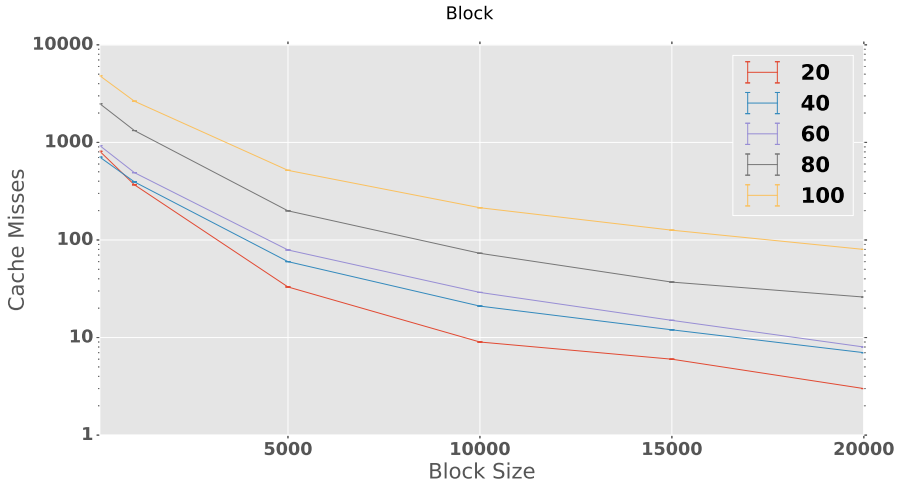


Figure 4.8: Graph miss rate for A* search of different path lengths using a block prefetcher

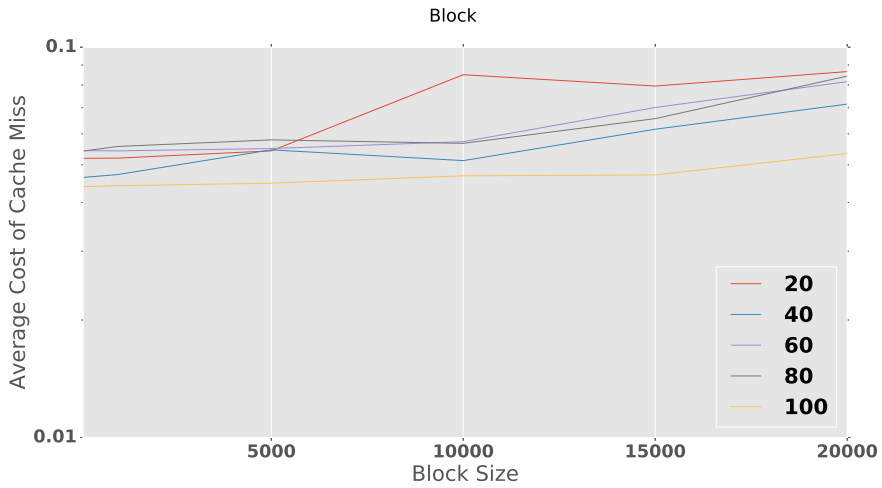


Figure 4.9: Average cost of store miss for A* search of different path lengths using a block prefetcher

lookahead prefetchers achieved significantly shorter execution times of around 0.1s, and the rate of improvement for the block prefetcher had stalled, no further experiments were run at larger block sizes.

It is interesting that this result contradicts the earlier one from Yoneki et al., who found that the block prefetcher offered better performance than a lookahead approach. Since we cannot make absolute comparisons between my test platform and theirs, we can only speculate about the cause of this disparity. It may be the case that my block prefetcher runs more slowly, which could be caused by my using a larger dataset (although the same DIMACS dataset is cited in the original Crackle report, several sizes are available and this detail is not reported). It is also possible that improvements to PostgreSQL since 2012 have made the index lookups in this dataset perform more quickly, or that indexes were misconfigured in the edge relation for the Crackle tests. These indexes are critical to achieving good lookahead performance, and indeed without them a block prefetcher performs significantly better on my setup.

For subsequent experiments, I used a CTE lookahead prefetcher with lookahead depth 5, since this offered a reliably good performance across all path

lengths.

4.3 Query Performance

Having established an appropriate prefetching strategy, the next task is to verify the hypothesis motivating Grapht's development: that there exists a significant performance difference between PostgreSQL and Neo4J for certain queries. Informally, we expect Neo4J to perform better for "graph-centric" queries, whereas we expect PostgreSQL to perform better for "row-centric" queries. A graph-centric query will tend to have the following features:

- The absence or existence of edges between vertices is of greater interest than the attributes of the edges or vertices
- Only a local part of the graph need be considered to satisfy the query

In contrast, a row-centric query will place equal weight on the attributes of entities as on the topology of the graph, and will typically need to consider vertices and edges which are not necessarily connected to one another.

Specific queries were chosen as representative of each of these categories, motivated by the mapping problem described in Chapter 1. To represent graph-centric queries, an A* search query was used. This satisfies the requirements, since the presence or absence of edges is what characterises a path between two points, and attributes of the vertices are otherwise ignored. Additionally, a solution can be found by searching through only vertices local to the start point, rather than needing to look at the entire graph. To represent row-centric queries, a string-matching query was chosen, which identifies the vertex with a payload which most closely matches some user-specified string. A match is determined using by calculating the Levenshtein distances[1] between the two strings. This is close to the kind of query required to find a location by name based on an approximate string search in a mapping appli-

[1]Informally, the Levenshtein distance is the minimum number of single-character edits required to change one word into another

cation, and is clearly row-centric since the vertex payload is of main interest, and every vertex in the graph must be considered in case it has the closest match.

4.3.1 Performance of Graph-centric Queries

To generate random queries for the A* search, a similar methodology was employed as for the prefetcher queries described in Section 4.2. Paths were chosen to be of specific lengths of 20, 40, 60, 80 or 100 hops, so that the average solution time would not be dominated by the execution time required to find particularly long paths. Degenerate cases are still possible with a fixed number of hops, however, as the heuristic guiding A* may lead the search in a suboptimal direction. To minimise the impact of this, twenty routes for each path length were examined, and the average time for all routes is presented here.

Finally, variation in performance might be due to differences in three implementations of A*. To avoid this, a single Scala implementation was prepared, with three adaptors responsible for fetching data from the relevant stores. For Neo4J and Grapht, the direct API was used rather than the declarative interface. Although it is possible to create an implementation of the A* algorithm entirely in SQL, the lack of an efficient way to create a priority queue makes performance unfairly low. Similarly, the Neo4J API provides a direct `astar` method which performs the full computation internally, but it would be unfair to compare its performance against another system which needs to transfer data from store to external application at each iteration, since Neo4J here does not suffer from the transfer latency. By providing a single A* implementation for all three databases, we narrow the focus of our test to look solely at evaluating their performance as a data store, rather than as an algorithm implementation platform.

Figure 4.10 shows the average time taken to find each path. We first notice that using Neo4J does indeed allow us to find a path significantly faster than PostgreSQL – a difference of nearly 1000x across all path lengths. This is

Figure 4.10: Average A* Execution time for paths of varying length across three database engines

unsurprising since A* is an archetypal example of a very local, topology-focussed algorithm, around which we expect a graph-centric database like Neo4J to be optimised.

As hoped, Grapht bridges the performance gap between the two systems. Although Neo4J's optimisations still grant it a performance advantage of around 35x, Grapht completes search queries almost 30x faster than PostgreSQL on average. This is a particularly pleasing result given that this performance improvement has been given "for free:" no changes have been made to the source data at all. By simply redirecting queries to pass through Grapht instead of directly to PostgreSQL, A* performance can be significantly improved.

4.3.2 Performance of Row-Centric Queries

To generate Levenshtein distance queries, a random number was generated, and an MD5 string was created from it. This string was used as the user-supplied search string. Figure 4.11 shows the time taken for each of the database systems to identify the vertex with the smallest Levenshtein distance from the user-supplied string. Here, we see the opposite results to the A* query. PostgreSQL completes the query significantly faster than Neo4J, by about 6x. This is not such a large performance gap as the 1000x seen in the previous subsection, and it seems likely that the cost of the query is largely dominated by the cost of calculating the Levenshtein distances. This cost will be comparable for the two engines, with the performance difference instead coming from PostgreSQL's row-centric optimisations.

Since Grapht builds on top of a standard relational database, it is able to simply forward the query on to PostgreSQL and achieve exactly the same performance. By doing this, we have improved significantly on Neo4J's worst-case performance.

Figure 4.11: Average Levenshtein-search execution time for 100 queries across three database engines

4.3.3 Performance of Hybrid Queries

Finally, we wish to consider not simply the performance of graph and row-centric queries in isolation, but also their performance when they are combined as in our motivating example. To create this kind of query, we use first a Levenshtein distance query to select a target vertex, and then perform an A* search from some pre-selected source vertex (as if our user was searching for a route to some destination by name). An appropriate source vertex was selected before tests were run in order to ensure that the shortest paths were of a comparable length.

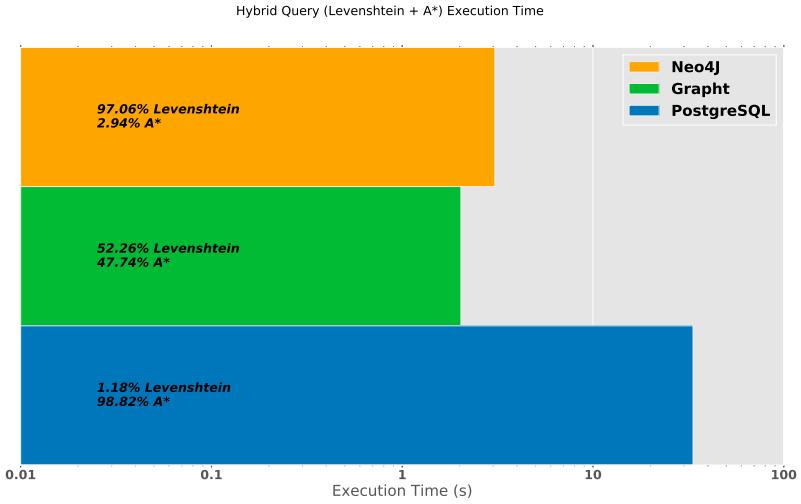


Figure 4.12: Average Execution time for 100 queries consisting of a minimum Levenshtein-distance search and an A* path search across three database engines. The percentage of time spent performing each of the two parts of the query is annotated for each engine.

Figure 4.12 shows the average execution time for these hybrid queries. In this scenario, Neo4J once again maintains a lead over PostgreSQL, thanks to it's 1000x performance difference for the A* component of the search. More interesting, however, is the fact that Grapht, by improving upon the worst-case performances of both other systems, has managed to outperform both commercial systems.

In reality, it would be impossible to select a database system based on these

benchmarks, without first determining exactly the type of queries which will most often be encountered in production. If a workload is identified which is dominated by graph-centric queries, it is likely that Neo4J would be the best choice, as it will not suffer enough from occasional row-centric queries to outweigh the benefits of the graph-centric optimisations. For workloads which are evenly balanced between graph and row-centric queries however, a hybrid system is able to outperform both specialised alternatives. Indeed, even if the workload is mostly row-centric, it may wise to choose Grapht over PostgreSQL alone, since there is no performance loss for row-centric queries, and gains are possible for occasional graph-centric queries. This suggests that maintaining an in-memory graph layer may be a worthwhile improvement to make more closely within PostgreSQL in future, rather than needing to exist as an optional add-on.

4.4 Usability of Query Systems

In the previous section, I performed a quantitative analysis of the performance of three database systems for graph-centric, row-centric and hybrid queries. Performance is only one aspect contributing to the success of a system, however. Cost of deployment, or ease of use during development are other crucial factors. If new custom queries are written frequently or on an ad-hoc basis, then slow-running but quick-to-write queries may obtain an answer to an information need more quickly overall.

There are five interfaces to consider in this evaluation. Neo4J offers two interfaces: a lightweight declarative query language called *Cypher*, and a Java API more appropriate for complex graph algorithms. PostgreSQL only offers one interface through SQL. Finally, Grapht takes the same approach as Neo4J in offering both a programmatic API and a declarative query language based on SQL called *gSQL*.

When considering the cost of deployment of a new database system, the time taken to learn to use the new system should also be examined. It is clearly

desirable from an ease-of-learning perspective to only need a single interface. Introducing a second interface is likely to create confusion about which one to use in different contexts, and at the very least increases the amount of learning required to effectively use the system to its full potential. One might first explore why it is that PostgreSQL can afford to take provide a single interface, while the more graph-oriented approaches must provide a lower-level API alongside a declarative query language. The answer lies in considering the relative search spaces of the two problems. Various optimisations and heuristics are possible to reduce the cost of traversing a tabular relation, but in the worst case it is always possible to retrieve a result in linear time by simply enumerating each row. The same is not true when considering paths through a graph. The number of possible paths grows exponentially with each additional hop in the path, meaning that for most problems it is not practical to simply enumerate all possible paths when searching for a single one. Instead, an intelligent approach is required to direct traversal through the graph.

Cypher, having been in development for some time, has the ability to exploit certain statistics and heuristics in order to slightly cut down the search space, but does not avoid the problem in any real way. To find the shortest path between two points using Cypher, for example, one must still enumerate all possible paths between the points, and rank them by increasing length. To avoid this, a large number of built-in functions are provided (such as `shortestpath`, which can be used to avoid this particular situation). However these built-in functions do not truly address the shortcoming of the language, which cannot be used to express custom traversal strategies through large graphs. When using Grapht's or Neo4j's traversal API directly, the user implements their own traversal strategy, removing the guesswork required for a declarative interface. gSQL sacrifices some of the declarative nature of the query language by allowing users to specify a traversal strategy with a `TRAVERSE BY` clause. This allows solutions to be found for a query more quickly, though at the cost of slightly more complicated query language.

A second factor limiting the expressiveness of a query language is the ability

to create and manipulate data structures. Almost all algorithms manipulate as part of their operation some central data structure. It is often difficult to define these structures using a declarative query language, which typically tries to abstract away from this kind of implementation detail. With some creativity, it is often possible to manage a workaround in SQL, by using temporary tables to encode the data structures required. A priority queue can be implemented in this way (as required for a full-SQL implementation of A*), though it will rarely perform as well as a solution using binary heaps. This is not possible at all in Cypher, which can perform little more than pattern-matching on paths through the graph. The path searches possible through gSQL are directed using the `TRAVERSE BY` clause, which internally uses a priority queue to select upcoming expansions. Much as it is possible to implement other data structures using a SQL table, it is also possible to make creative use of Grapht's priority queue to implement some other structures. For example, a "last-in, first-out" stack can be created by using the insertion index as priority. This is not quite so flexible as SQL, but again allows slightly more complicated algorithms to be implemented entirely in gSQL than are possible in a language like Cypher.

Although some queries are totally inexpressible in a particular query language, the ease of development for expressible queries should also be considered. It is difficult to fully comment on the ease of use of the different query languages without a formal usability study. However I would expect to find that neither the Grapht nor Neo4J APIs would score highly in this kind of study. Although these can be used to express any conceivable query using the full power of any JVM language, needing to prepare and compile code in order to express a simple query can be a is a huge complication, especially when much of the code will be concerned with routine tasks like establishing a database connection or deserialising data. Although Java is a very well-known programming language, using APIs in this way also requires that users be familiar with Java as well as the database API itself. For newcomers, this means overcoming two learning barriers rather than just one. Instead, I would expect that Cypher obtain a high usability score for

novices, since its pattern-matching queries are expressed in a fairly intuitive visual language. On the other hand, SQL is much more likely to have been encountered before by experienced developers, which would presumably lead to a higher score for this audience. gSQL, as a relatively small extension on top of SQL, would likely be found easier to use by experienced developers than by novices.

Chapter 5

Conclusion

TODO

Bibliography

[1] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[2] Peter Pin-Shan Chen. "The Entity-relationship Model&Mdash;Toward a Unified View of Data". In: *ACM Trans. Database Syst.* 1.1 (Mar. 1976), pp. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440. URL: http://doi.acm.org/10.1145/320434.320440.

[3] Jim Gray et al. "The transaction concept: Virtues and limitations". In: *VLDB*. Vol. 81. 1981, pp. 144–154.

[4] Serge Abiteboul and Nicole Bidoit. "Non First Normal Form Relations to Represent Hierarchically Organized Data". In: *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. PODS '84. Waterloo, Ontario, Canada: ACM, 1984, pp. 191–200. ISBN: 0-89791-128-8. DOI: 10.1145/588011.588038. URL: http://doi.acm.org/10.1145/588011.588038.

[5] H. S. Kunii. "DBMS with Graph Data Model for Knowledge Handling". In: *Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow*. ACM '87. Dallas, Texas, USA: IEEE Computer Society Press, 1987, pp. 138–142. ISBN: 0-8186-0811-0. URL: http://dl.acm.org/citation.cfm?id=42040.42071.

[6] Joachim Biskup, Uwe Rasch, and Holger Stiefeling. “An extension of SQL for querying graph relations”. In: *Computer Languages* 15.2 (1990), pp. 65–82.

[7] W. Kim. “Object-Oriented Databases: Definition and Research Directions”. In: *IEEE Trans. on Knowl. and Data Eng.* 2.3 (Sept. 1990), pp. 327–341. ISSN: 1041-4347. DOI: 10.1109/69.60796. URL: http://dx.doi.org/10.1109/69.60796.

[8] Camil Demetrescu, Andrew Goldberg, and David Johnson. “9th DIMACS implementation challenge–Shortest Paths”. In: *American Mathematical Society* (2006).

[9] Grzegorz Malewicz et al. “Pregel: a system for large-scale graph processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[10] Amit Singhal. “Introducing the knowledge graph: things, not strings”. In: *Official Google Blog, May* (2012).

[11] Eiko Yoneki and Amitabha Roy. *A unified graph query layer for multiple databases*. Tech. rep. UCAM-CL-TR-820. Department of Computer Science, University of Cambridge, 2012.

[12] Julian Shun and Guy E Blelloch. “Ligra: a lightweight graph processing framework for shared memory”. In: *ACM SIGPLAN Notices*. Vol. 48. 8. ACM. 2013, pp. 135–146.

[13] Robert Campbell McColl et al. “A performance evaluation of open source graph databases”. In: *Proceedings of the first workshop on Parallel programming for analytics applications*. ACM. 2014, pp. 11–18.

[14] Wen Sun et al. “SQLGraph: An Efficient Relational-Based Property Graph Store”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1887–1901. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2723732. URL: http://doi.acm.org/10.1145/2723372.2723732.

[15] R Pointer. “Introducing FlockDB, Twitter, Inc.” In: *Available: https://blog.twitter.com/2010/int flockdb* ().