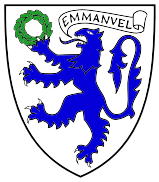
UNIVERSITY OF
CAMBRIDGE

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 21, 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: 6165

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

1 Introduction **1**
1.1 Research Goals 2

2 Background **5**
2.1 Graph Databases 5
2.2 Relational Databases 5
2.3 Related Work 6

3 Implementation **7**
3.1 System Overview 7
3.2 Row Handler 9
3.3 Graph Handler 9
3.3.1 Lookahead Prefetcher 10
3.3.2 Block Prefetcher 11
3.3.3 Iterative Lookahead Prefetcher 11
3.4 Grapht API 13
3.5 Query Processor 17
3.5.1 Improving on the Graph Manipulation Kernel 19
3.5.2 Executing gSQL 21

4 Evaluation **23**
4.1 Test Framework 23
4.2 Prefetcher Performance 23
4.3 Performance of Graph-Centric Queries 24
4.4 Performance of Row-Centric Queries 24
4.5 Performance of Hybrid Queries 24
4.6 Expressiveness of Query System 24

5 Conclusion **25**

Bibliography **25**

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 RDMS and Graph Database model, but the core division is that where an RDBMS is optimised for aggregated data, a graph database is optimised for highly connected data. In practice, however, although it is certainly the case that some datasets are more highly connected than others, many of today's business operations cannot be classified according to that metric alone.

Social networks are often used as an important example of a highly connected dataset today. However the reality is that, although relationships between friends are ideally modelled as a graph, this is not the case across the board. Performing queries across all users – such as discovering users who live within

particular geographic bounds – is much less efficient when every single edge between users needs to be considered during a graph traversal. This heterogeneity, 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.

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 Neo4J for graph-centric problems and as well as PostgreSQL for row-centric problems. Instead, the aims for this project were to improve on the worst-case performance of these systems, while still minimising the loss in best-case performance. By doing this, we can provide a system which provides a higher average-case performance in the situation where graph-centric queries and row-centric queries are both used. More concretely, there were four goals for this project. Two are fairly trivial improvements over existing systems, while the other two arise precisely because of the hybrid nature of the system. Note that for this research project, we limit our attention to data-retrieval queries only.

1: Increase performance of graph queries compared to a relational engine

Relational database engines perform poorly for graph-centric queries such as path-finding. A hybrid engine should perform better than this, even if it cannot achieve performance on par with purely graph-centric engines.

2: Increase performance of row queries compared to a graph engine

Conversely, graph engines perform poorly for row-centric queries such as aggregation over all nodes, and a hybrid engine should aim to perform better than this.

3: Increase performance of hybrid queries compared to both engines

Some queries may have multiple components to them, some of which would best be handled by a graph engine, while others best by a relational system. For example, we may wish to identify two users who are geographically closest to one another, and find a path between them through the social graph. Finding geographical neighbours involves comparisons between all nodes, not just ones which are local to one another in the social graph. This would thus be well handled by a row-centric system, and less well by a graph-centric one. The path-finding component, however, would perform much better under the graph system. Under either system, some component of the query would necessarily perform badly and cause a performance bottleneck. A hybrid system ought to not be affected so badly by either bottleneck, and may thus outperform both systems for this kind of query.

4: Expose a coherent interface for the hybrid system

Relational databases have a well-established query interface in the form of *SQL*. Queries in SQL aim to create and select particular rows from the database, and are thus inherently row-centric, 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 considering all nodes 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.

TODO: Mention that I'm only interested here in read-query ability, and that updates or insertions are not covered (In future work we can say that these shouoldn't be too difficult to add some sort of cache coherency model)

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 row-based systems. Chapter 3 discusses the implementation of a research prototype aiming to satisfy all four 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 point of reference to measure Grapht against, 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 Graph Databases

TODO: Invention

TODO: separate from graph processing (e.g. Pregel, based on BSP) [c.f. OLAP and OLTP]

TODO: Comment that problem is common enough for people to have developed in-house solution (FB open Graph, Google Knowledge Graph, Twitter FlockDB)

TODO: Benefit of graph DB is traversals are localised and do not have to take into account unrelated data Obviously this can be a disadvantage

TODO: Intro to Neo4j

2.2 Relational Databases

TODO: RDBMS has been around for a while - more stable, provides better ACID guarantees Generally people are more invested in an older system, and reluctant to change!

TODO: Intro to PostgreSQL

2.3 Related Work

TODO: see main.bib

TODO: Mention ORMs which have a similar aim to make RDBMS relations feel more natural (but don't generally improve performance in the same way Grapht does)

Chapter 3

Implementation

In aiming to produce a hybrid system somehow equally well suited to both classes of problem, 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 prefetched cache.

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 cache prefetcher and the query processor.

3.1 System Overview

As mentioned earlier, Grapht extends a relational data store – a graph query layer is "grafted" onto the underlying store. It is itself made up of two parts: a row-centric and a graph-centric handler. Both components communicate directly (and separately) with the underlying store via standard SQL queries.

This allows any SQL-compatible data-store 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, such that it would be impossible to efficiently represent the full diversity using a single query language. For this reason, the query processor is designed the be used as a possible alternative to using the Grapht API directly. (TODO: Diagram of main components)

The Grapht API has a relatively small interface. Method invocations to the API are mostly handled exclusively by one of 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, relying instead on decades of optimisation made in relational databases to efficiently retrieve data by passing queries directly on to the underlying store.

Within the graph handler, data is laid out in memory to optimise graph-centric access patterns. The handler contains a cache, 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 cache, the vertex data is obtained from the relational database. At the same time, a prefetcher is used to populate the cache with adjacent vertices, so that subsequent requests from the query processor can hope to be fulfilled directly from the cache.

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[1]. The extension is a strict superset of SQL, allowing pure-SQL queries to be passed on directly to the underlying store. Pure-graph queries end up being handled exclusively by the graph handler, and hybrid queries are broken apart and 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 store, and the resulting relations are passed on. 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.

3.3 Graph Handler

The graph-centric handler is slightly more involved than the row-handler in that it contains a cache, in which vertices are looked up before resorting the the underlying store. Internally, it 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 to the target vertex directly to avoid indirection through an index in the same way. 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 an index to the target vertex, which may not yet be loaded into the cache.

When a vertex is requested through the Grapht API, the cache 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 data store 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 query processor, 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 cache, and removing them when the cache exceeds the memory allowed by the current Java virtual machine (JVM). When the size of the cache does exceed the allowed size, vertices are removed from the index according to a least-recently used strategy. That is to say, each time a vertex is requested, the access is logged, when a vertex must be eliminated from the index, the vertex with the least recent access time is chosen. 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 nodes. Thus these vertices should be kept in the cache, 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. [2], but I have here replicated and extended these results with a third strategy.

3.3.1 Lookahead Prefetcher

The first strategy examined by Yoneki simply performed a limited depth-first traversal of the graph, starting from the requested node. 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 vertex properties 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 property 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 cache 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 many requests are made from the data store, 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 still smaller than the join cost of the CTE for large lookahead depths, and in fact this iterative lookahead approach allowed lookaheads of almost ten times the optimal depth of the original lookahead prefetcher without any significant loss of performance. Additionally, A* search algorithms ran faster when using the iterative lookahead prefetcher, presumably because the prefetcher offered guaranteed graph locality, where the block prefetcher could only base decisions on a presumed correlation.

As a result of these findings, the iterative lookahead prefetcher was used as the default Grapht prefetcher. An optimal prefetch depth was not so clearly identifiable, and it is likely that this optimality will vary based on factors such as cache 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).

3.4 Grapht API

The Grapht API allows client applications to query the underlying data store, 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 query, one must first initialise the graph cache with a particular source. This is done by using the `createGraph(vertices, vertexKey, edges, 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`, `sourceKey` and `targetKey` are used to determine which fields of the chosen relations correspond to vertex, edge source and edge target identifiers respectively.

Once the graph is initialised, two simple methods target the graph-centric handler directly, retrieving 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 vertex this edge leads to. As mentioned earlier, vertices are not guaranteed to be present in the graph cache 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 underlying store. On the other hand, edges will always be present cache 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. Between these five methods, it is possible

to traverse the entire graph efficiently, starting and ending at any arbitrary vertex, and collecting any desired data along the way.

Almost any algorithm can be efficiently expressed using this interface for graph traversal and retrieval of data. For convenience, a few further methods are provided, which use the methods described above for themselves. The most powerful of these is the `getPaths` method, which is used to discover paths branching out from some 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.

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, where the priority for new vertices is zero by default (in which case the priority queue devolves into a normal first-in, first-out queue and the search proceeds as a breadth-first search). The priority of a vertex is otherwise determined by a user-provided *prioritiser* function.

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 property 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 certain depth by using a `count` accumulator in conjunction with an appropriate evaluator. An opportunity to cut the search space down is also provided to the user, who can optionally specify

a direction or rate of change for each accumulator. For example, a `count` accumulator is guaranteed to increase by one with each step through the graph, while a `sum` accumulator can be assumed to at least increase each step if the property it is accumulating is always positive. If an evaluator is provided which rejects values greater than some given constant, then we do not need to continue expanding paths once that constant has been reached with the `sum` accumulator.

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 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 cache. 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 field in the relation. This allows paths through the graph to be manipulated just like any relation in the underlying store, 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 my 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 underlying store, so that joins could literally be performed as between normal relations. The current Grapht prototype is not tightly coupled enough to the underlying store 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. The precise impact of this prototype limitation

is discussed further in Chapter 4.

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 discard path information if it is not of interest, and instead only consider data belonging to nodes in some way reachable from the source node. For example, we may wish to retrieve a list of all junctions reachable within 100km of some given source junction. This can be achieved by using a `sum` accumulator as described above and a `last` accumulator, which simply retrieves a certain property from the last vertex along a path. An evaluator can then be attached to the `sum` accumulator, checking whether the path to reach this vertex is less than 100km. Since the sum will be non-decreasing, an exhaustive search will terminate as paths cease to be explored when 100km are passed. In this way, we can quickly retrieve a relation containing just the details of 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[1] (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 all 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 some the vertex with ID 1 would be written in Biskup's graph manipulation kernel:

```
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 properties of the edge relation can then be aggregated along the path to provide new data to be included in the output path relation. 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 included implicitly as start and end properties within the edge relation. It is possible to include vertex properties 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) may visit every single other node 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, or 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` property 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 this declaration of accumulation is a function which will be used 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 property to accumulate at each step and a separator string, returning the concatenated value of all named properties 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 nodes 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 MIN(length)
```

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. `length` is a default property of all paths, and so at each opportunity to discover a new vertex, Grapht will select one according to the minimum path length so far (i.e. according to a breadth-first traversal).

One limitation has been introduced to gSQL queries which was not present in GMK. This is 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 node) but these strategies have not been implemented for this initial prototype, hence a start node 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 source of data specified in the query is looked at. 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 should be optimised for.

In the case of complex clauses which join together results from disparate data sources, 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`) 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 data relations has not been seen before, a new subgraph is initialised in the cache 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 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 `LENGTH` accumulator which is implicitly created is intialised with a known

rate-of change. This allows queries which have a bounded depth in the `WHERE` clause to run exhaustively within the appropriate bounds, since Grapht knows that a shorter `LENGTH` will never arise further along the path. Although this is a desirable property, 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.

Chapter 4

Evaluation

4.1 Test Framework

4.2 Prefetcher Performance

In the following subsections, tests were performed to evaluate different prefetching strategies. In all cases, data was collected about the prefetcher while responding to an A* traversal across a DIMACS dataset representing the US road network[**dimacs**]. For repeatability, six queries were fixed and used for every prefetcher. A more thorough description of the benchmarking environment can be found in Chapter 4.

4.3 Performance of Graph-Centric Queries

4.4 Performance of Row-Centric Queries

4.5 Performance of Hybrid Queries

4.6 Expressiveness of Query System

Joins between graph and relational components go here, rather than under performance of hybrid queries, I think.

Chapter 5

Conclusion

TODO

Bibliography

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

[2] 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.