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 19, 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: 2973

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 Graph Databases . . . 2
- 1.2 Relational Databases . . . 2
- 1.3 Related Work . . . 2
- 1.4 Research Goals . . . 2

2 Implementation **5**
- 2.1 System Overview . . . 5
- 2.2 Query Processor . . . 6
  - 2.2.1 Improving on the Graph Manipulation Kernel . . . 8
  - 2.2.2 Executing gSQL . . . 10
- 2.3 Prefetchers . . . 11
  - 2.3.1 Lookahead Prefetcher . . . 11
  - 2.3.2 Block Prefetcher . . . 11
  - 2.3.3 Evaluation . . . 11

3 Evaluation **13**
- 3.1 Test Framework . . . 13
- 3.2 Performance of Graph-Centric Queries . . . 13
- 3.3 Performance of Row-Centric Queries . . . 13
- 3.4 Performance of Hybrid Queries . . . 13
- 3.5 Expressiveness of Query System . . . 13

4 Conclusion **15**

Bibliography **15**

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.

In this 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. Finally, I will outline the research goals for my system, Grapht.

1.1 Graph Databases

1.2 Relational Databases

1.3 Related Work

1.4 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.

Chapter 2 discusses the implementation of a research prototype aiming to satisfy all four of these goals: *Grapht*. The success of this prototype is experimentally evaluated in Chapter 3, 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 4 summarises the findings made, and discusses possible directions for future research.

Chapter 2

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.

2.1 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: the query processor and the prefetcher. Both components communicate directly (and separately) with the underlying store via standard SQL queries. This allows any SQL-compatible data-store to be store via standard SQL queries. This allows any SQL-compatible data-store to be used fairly indiscriminately.

The query processor receives queries from a client application and breaks them apart into row-centric subqueries targeting the relational database directly, and graph-centric subqueries targetting the graph cache. 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 are handled exclusively by the graph cache, and hybrid queries are broken apart and dispatched appropriately.

The graph cache presents a simple API for the processor to use. Internally, data is laid out to optimise graph-centric access patterns. The cache has a limited size by design, so that Grapht can be used on systems even with small amounts of memory available. When the query processor requests a vertex 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.

2.2 Query Processor

Biskup's Graph Manipulation Kernel[1] (GMK) unites relational and graph models by defining paths over graphs as a relation 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 a graph using a syntax very similar to SQL. For example, a "friend-of-friend"-style query in Biskup's graph manipulation kernel to find vertices two hops away from some the vertex with ID 1 would be written:

```
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.

2.2.1 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 depth-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, as it gives the Grapht implementation a known place to start searching for possible paths. 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, such that a start node must always be provided.

2.2.2 Executing gSQL

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 for the query is looked at. If data is being queried from one of the relational tables, then that data is fetched from the relational store directly. If data is being queried from a "`PATHS OVER`" clause, then the query will instead be resolved through the Grapht cache. 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, since the result of a `PATHS OVER` query is ultimately still a relation. Ideally, this output relation of paths would be indistinguishable from a normal relation as far as the underlying store is concerned. For the purposes of this prototype, however, PostgreSQL is totally unaware of the path relation which Grapht has constructed. Until integration within PostgreSQL can be more tightly made, a workaround is to load the output relation into PostgreSQL as a temporary in-memory table. 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 3.

Once the query has been identified as targeting the Grapht cache, 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.

The `WHERE` condition is evaluated at each vertex along a possible path. To improve performance here, the evaluator can be made aware of the rate or direction of change of each variable along the path. For example, it is known that along any path the `LENGTH` attribute will increase by one with each hop. Similarly, if the user knows that no edges have a negative `cost` attribute, then sum of these costs will be nondecreasing along the path. This knowledge is used to cut down the search space when it is known that further expanding a path cannot yield a valid solution. Currently, this ability to specify variables' directions of change is not exposed to the gSQL client. Instead, the `LENGTH` accumulator is the only one initialised with rate-of-change knowledge. Future versions of Grapht could expose this ability, giving the power to clients to tune their queries to improve performance.

TODO: paragraph on prioritiser then we're done here

2.3 Prefetchers

2.3.1 Lookahead Prefetcher

2.3.2 Block Prefetcher

2.3.3 Evaluation

Chapter 3

Evaluation

3.1 Test Framework

3.2 Performance of Graph-Centric Queries

3.3 Performance of Row-Centric Queries

3.4 Performance of Hybrid Queries

3.5 Expressiveness of Query System

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

Chapter 4

Conclusion

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.