Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Håkon Strandlie

Routing in MySQL

A memory-aware approach

Master's thesis in Computer Science
Supervisor: Norvald Ryeng
June 2021

NTNU
Norwegian University of
Science and Technology

Håkon Strandlie

Routing in MySQL

A memory-aware approach

Master's thesis in Computer Science
Supervisor: Norvald Ryeng
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

This thesis looks at implementations of routing / shortest path calculation in databases and argues that while there exists an implementation for PostgreSQL, its imperative approach is suboptimal in a database system. To demonstrate an alternative, the thesis implements a prototype of a declarative routing module in MySQL and examines some of the consequences of a declarative interface. The prototype is experimentally evaluated and found to use large amounts of I/O, with large time consumption as a consequence. A more effective buffering strategy is suggested based on the results from the evaluation.

Acknowledgement

I am very grateful to Norvald Ryeng for being the best kind of advisor, giving insights, answering questions, and always being available for inspiring discussion and helpful feedback throughout the work with this thesis. I would also like to thank Adam Wulkiewicz and Torje Digernes at Oracle, for helping me with learning and understanding MySQL, Boost and C++. Without you, this thesis would not have been the same.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

Questions like *"Where am I?"* and *"How do I get to where I want to be?"* are asked every day. More and more often, these questions are answered in computer systems using geospatial data, and as existing data is improved and new data is collected, support for operations on these data is in increasing demand from a range of systems. Several database systems, with MySQL as an excellent example, now supports storing and processing geospatial data, and MySQL 8.0 made geospatial data a first-class citizen in the system[1].

To build on this excellent support, this thesis implements a working prototype of a routing module in MySQL, and the prototype is evaluated using real data from OpenStreetMap. Many routing systems have previously been implemented and are well known to end-users through services such as Google Maps and GPS guidance systems in vehicles. These systems often have routing implemented as an application-level service, where the shortest path is calculated on a server or a client, based on geospatial data returned from a database. To save the application layer from having to perform this task, it is useful to have the ability to perform routing in the database system, since this will reduce the amount of data transferred from the database. pgRouting in PostgreSQL is an example of an existing implementation of routing in a database system. The SQL interface to pgRouting requires the user to specify which algorithm to use when calculating the shortest path, i.e. it takes an imperative approach. This is a significant mismatch with what the user of a SQL-based database system would expect since SQL is a declarative query language. Additionally, it places the responsibility of memory management on the user, since the memory usage of a given algorithm is deterministic from the size of the graph. This means the user needs to be aware of the memory usage of a routing query because the implementation has no other option but to abort the query if memory usage is too high.

This thesis examines the following research questions:

RQ1: How can we create an SQL interface to a routing module that allows the database system to optimize the execution similarly to how it optimizes a normal query?

RQ2: What are some of the factors that impact the performance of a system with an interface as in **RQ1** when the system is memory limited?

Answering **RQ1** leads to question about whether the interface should be imperative or declarative, which places different requirements on the user, the dataset, and the system, depending on what approach is taken, as discussed in Section 2.2.4.

For **RQ2** it is important to consider that datasets containing the entire world or the solar system are not inconceivable, so routing systems where the system optimizes the query must be able to handle the case where datasets are so large that they do not fit into memory. As a prototype of this approach, the thesis also implements the aggregation function `ROUTE`, which presents a declarative interface to routing functionality. Existing implementations and their limitations are examined, as well as possible algorithms and some optimizations, and the consequences of such a declarative approach are explored, such as a necessary hands-on approach to memory management, and an effective cache/buffer replacement policy. The implementation introduces `RVector`, an STL container compliant data structure for storing the vertices in the graph. `RVector` observes a memory limit, maintains a memory buffer, and moves data to disk when the memory limit is exceeded. It attempts to use the available resources as efficiently as possible, and then degrade gracefully when the resources are exhausted.

To achieve the result of this thesis, significant work was performed with integrating custom buffer functionality with the existing implementation of MySQL, which is a large and mature system, and with the Boost Graph Library. Getting a correct and working implementation of this required substantial investments of time and effort. Fortunately, the goal was reached and a working routing module is presented in the thesis, together with performance results, and directions for future improvements.

Chapter 2 examines the details of the routing problem, together with details of a graph representation of the problem, and the relationship between routing and geometry data. It also looks at background information relevant for the implementation, such as algorithms and some optimizations on these, cache replacement policies, and spatial access methods. The distinction between imperative and declarative interfaces, and the consequences of a declarative interface is visited, and details about MySQL, GIS, and the current state of routing in MySQL, as well as existing implementations in PostgreSQL, Oracle, and Boost Graph Library are also examined. Chapter 3 describes the experimental evaluation that was performed on the implementation, including details about the datasets and details about the implementation of `RVector`. Chapter 4 discusses the results, including the advantages, disadvantages, and practical usability of the current implementation, and possible improvements based on the results from the test.

Chapter 2

Background

This chapter contains background information that is important to a solution to the routing problem. Section 2.1 defines the problem, then Section 2.2 examines relevant theory and concepts, before Section 2.3 looks at some existing implementations.

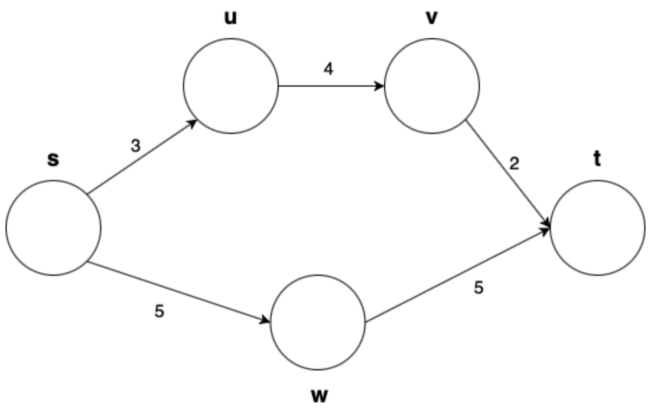
In this section, the graph notation from [2] is used, in which a graph is given as $G = (V, E)$ and we access the set of vertices in the graph using $G.V$, and the set of edges by $G.E$. We also find adjacent vertices (neighbors) of a vertex $v \in G.V$ with $G.Adj[v]$.

2.1 The problem

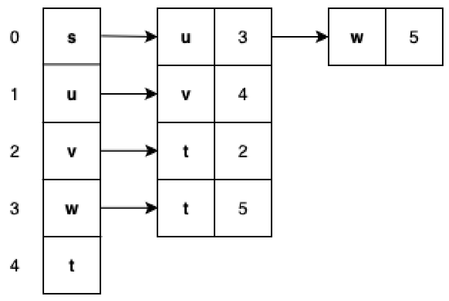
Routing, or **shortest-path selection**, is the task of selecting a path from n sources to m targets where $n, m \in [1, N]$. It is a well known problem in the literature, and it is used to solve several real-world problems, such as Internet packet switching and the Traveling Salesperson Problem[3]. With the emergence of geographic data and geo-tagging on the internet[4] routing with geographic data is in increasing demand.

2.1.1 Problem representation as a graph

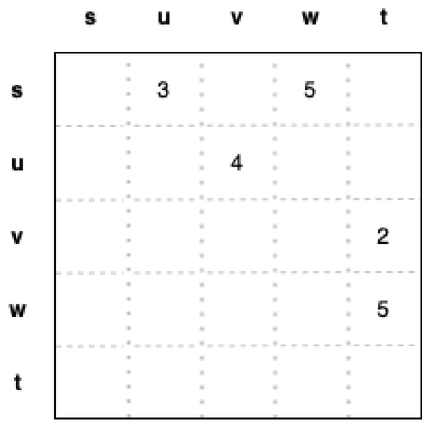
The routing problem must be transferred into a manageable form to be reasoned about and solved, and this is usually accomplished by mapping the problem domain into a graph. A graph representation maps a problem domain to a graph $G = (V, E)$ where G is *directed* if the edges $e \in E$ are ordered[2], and an edge is ordered if the edge $e_1 = (v_1, v_2)$ is *not equal* to the edge $e_2 = (v_2, v_1)$ and v_1 is *not equal* ot v_2. Otherwise, the graph is undirected. Each of these edges can have a weight associated with it, provided by a **weight function** $w : E \longrightarrow \mathbb{R}$. This function typically represents **distance**, **time** or **price** in the problem domain. More infor-



(a) Example of a directed graph with 5 nodes, and 5 edges with edge weights.



(b) Adjacency list memory representation of the graph in Figure 2.1a



(c) Adjacency matrix representation of the graph in Figure 2.1a. Blank spaces represent no edge between corresponding nodes.

Figure 2.1: Example of a graph, and corresponding memory representations for the graph. Figures adapted from [2]

mally, this mapping can be to take a dataset and create a data structure as shown in Figure 2.1a, which shows an example of a *directed* graph with edge weights and its associated possible memory representations. Here, each edge could represent a road and each vertex could represent intersections between these roads.

Although other representations than a graph exist, as discussed in Section 2.2.1, representing the problem as a graph is usually the preferred solution, since graphs are well studied in the literature. This means that many algorithms, data structures, and optimization techniques exist for applications in many fields[5], and many of these are often used in the industry which means that they are well tested.

Given that a graph representation is used, the problems that can be solved can be categorized into four categories:

Single-Source Shortest Paths: Given a graph $G = (V, E)$ and a **source** vertex $s \in G.V$, find a shortest path in the graph from s to each other vertex $v \in G.V$

Single-Destination Shortest Paths: Given a graph $G = (V, E)$ and a **target** vertex $t \in G.V$, find a shortest path in the graph from each other vertex $v \in G.V$ to t. By reversing the direction of each edge $e \in G.E$, the problem is equivalent to Single-Source Shortest Path.

Single-Pair Shortest Path: Given a graph $G = (V, E)$ and two vertices $u, v \in G.V$, find a shortest path in the graph from u to v

All-Pairs Shortest Paths: Given a graph $G = (V, E)$, find a shortest path in the graph for all pairs of $u, v \in G.V$ where $u \neq v$

For this thesis, **Single-Source Shortest Path** and especially **Single-Pair Shortest Path** are most relevant, since they are the problems usually solved by a routing system. **All-Pairs Shortest Paths** can be used for pre-processing, for example when calculating reach, as described in Section 2.2.1.

2.1.2 Graph representation in memory

There are two standard ways to represent a graph in memory[2]. Representing the graph as an **Adjacency List** is usually preferred over representing the graph as an **Adjacency Matrix** since most graphs encountered in applications are sparse and **Adjacency List** provides a way to store these efficiently. As an example, see Figure 2.1, where an example graph with 5 vertices and 5 edges is shown. In Figure 2.1b the memory representation of an **Adjacency List** can be seen, which shows an array of 5 adjacency lists - one list for each vertex present in the graph. Similarly, in Figure 2.1c the memory representation of **Adjacency Matrix** can be seen, which has one column and one row for each vertex, such that an edge between two vertices is indicated by the presence of a number in the cell in the intersection of the row for the source and the column for the target. Note that the blank spaces in Figure 2.1c represent that no edge with source row and target $column$ exists. As a consequence of this structure, checking whether there is an edge that connects two given vertices, or adding and removing such edges, can be performed in $O(1)$ time. The disadvantage is that it will have a memory cost of $O(|V|^2)$ no matter how few edges the graph contains, and for this reason the memory cost of $O(|E|+|V|)$ of *Adjacency List* is usually preferred for sparse graphs.

2.1.3 Geometry

In MySQL a **geometry** is defined as *a point or an aggregate of points representing anything in the world that has a location*[6]. The OGC Simple Feature Access standard[7] defines a `POINT` datatype. MySQL complies with this standard, and implements a `POINT` datatype which is defined as a representation of "a single location in coordinate space", and the `LINESTRING` datatype is defined as a sequence of points or a "Curve with linear interpolation between Points" to create a geometry with a continuous curve. Other geometric datatypes also exist, but for our purposes the `POINT` and `LINESTRING` are most relevant.

One way to store the data describing a graph is by using these two data types. Since the data for a routing graph will often consist of roads, which will intersect with other roads at some points, `LINESTRING`s can be used to store a road with curves and bends, and these `LINESTRING`s can run between `POINTS` which store intersections between roads. From this, the distance of a path can be found by summing the lengths of all the `LINESTRING`s which constitute this path. All geometric datatypes in MySQL inherit from the `GEOMETRY` root class[8] and thus have coordinates and length information that can be calculated from the coordinates.

On the other hand, we do not need this geometric information directly to perform routing when using a graph representation, and we can take an approach where the geometry is decoupled from the routing operation. The only requirements for building a graph are to have some way to identify vertices, i.e. a vertex ID, and to have some metric for the length of edges. In the datasets used in this thesis (Section 3.1.1) the distance of edges was supplied as a separate field. This distance is calculated as a straight-line distance using the coordinates of vertices in the `LINESTRING`[9] (or `Way`, as it is called in the tool we used to export data from OpenStreetMap[10]), and consequently, it can be seen that the geometric information is used in the toolchain. But at the same time, it is worth noting that the routing operation itself does not require any geometric information.

The SQL/MM standard[11] takes the approach where it, as seen in Code Listing 2.1, assumes a column that stores edges as geometries, and source and target vertices are specified as `POINT`s, i.e. geometries. In other words, an SQL/MM-compliant implementation cannot be decoupled from the geometries during the calculation of routes. This could require making geometric operations more often at query time to determine lengths of edges which could result in substantial overhead for a large graph, while a decoupled approach calculates this as a pre-processing step.

2.2 Theory and concepts

This section will describe some relevant theory and concepts for the thesis. First, Section 2.2.1 will take a look at different methods used when routing, which is another way to categorize routing algorithms, different than the categorization presented in Section 2.1.1. Then, Section 2.2.2 will examine some algorithms in detail, together with some optimization techniques for reducing the cost of these algorithms in Section 2.2.3. Section 2.2.4 and 2.2.5 will look at the differences between imperative and declarative routing, and examine some of the consequences of a declarative routing system, before Section 2.2.6 and 2.2.7 look at Cache Replacement Policies and Spatial Access Methods.

2.2.1 Routing Methods

Section 2.1.1 categorized graph algorithms based on the number of sources and targets a shortest path was calculated for, i.e. **Single-Pair Shortest Path** and **All-Pairs Shortest Path**. This section will take a look at the underlying method and *idea* used when calculating the shortest path. The methods in this section mainly solve the **Single-Pair Shortest Path** problem, and except for round-based routing, they assume a graph representation. Round-based routing is included as an example of another method.

Relaxation-based routing

The origin of the word *relaxation* is in [12] given as the following:

> The notion of "relaxation" comes from an analogy between the estimate of the shortest path and the length of a helical tension spring, which is not designed for compression. Initially, the cost of the shortest path is an overestimate, likened to a stretched-out spring. As shorter paths are found, the estimated cost is lowered, and the spring is relaxed. Eventually, the shortest path, if one exists, is found and the spring has been relaxed to its resting length.

In other words, relaxation takes initial overestimates for the distance along paths, and iteratively improves them until an optimal solution is reached. This is the technique used in well-known algorithms such as Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm. This method produces optimal results, but their space and time complexities can be prohibitive for large graphs.

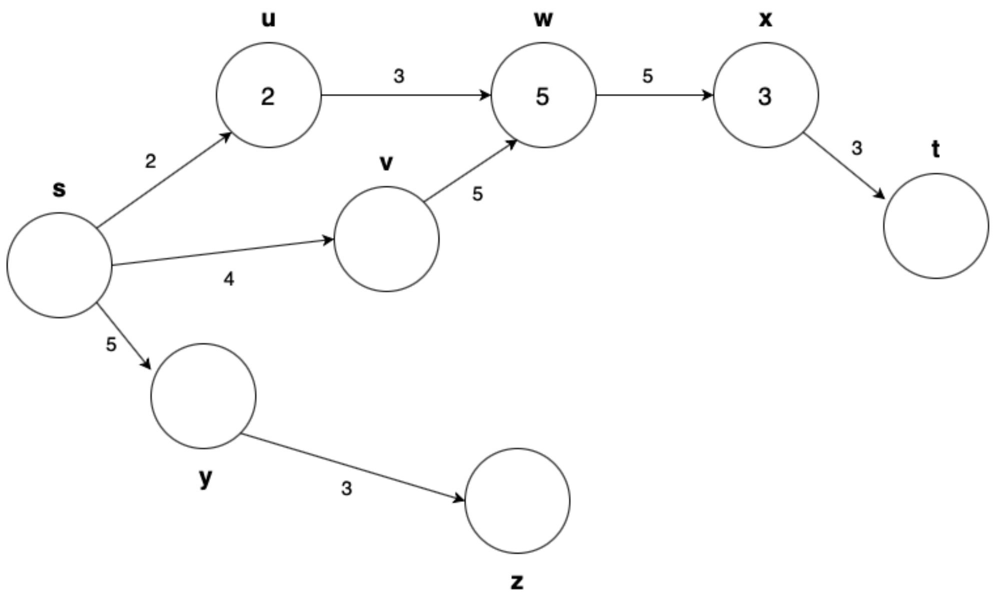


Figure 2.2: An example of the reach of vertices. For clarity, the *reach* of nodes **s**, **v y**, **z** and **t** are not shown.

Reach-based routing

In relaxation-based algorithms, an attribute called *reach*[13] can be added to vertices in the graph. It is mainly used when routing in road networks, and it is defined as $r(v,P) = min\{m(s,v,P), m(v,t,P)\}$ where v is a vertex on a shortest path P, $m(u,v,P)$ is some *reach* metric, and s, t are source and target vertices. Informally, given a shortest path P the value $r(v,P)$ denotes how far one can travel on P from v, and this value will be high if v is close to the middle of P, and P is a long shortest path. Thus, it signifies the importance of v in the graph, and it indicates whether a vertex should be considered during relaxation. In practice, this means that large parts of the search space are pruned, as less important vertices are ignored.

In Figure 2.2 an example can be seen of the calculation of reach for u, w and x where P is the shortest path from s to t. Here, the sum of the edge weights from u to v is used as the reach metric $m(u,v,P)$, and *reach* for w is calculated as $r(w,P) = min\{m(s,w,P), m(w,t,P)\} = min\{5,8\} = 5$. It can also be seen that if u is removed from the graph, the shortest path from s to t goes through v, and the reach of w is increased to 8 since it is now in the center of a shortest path which is now longer. This means that w can now *reach* farther in both directions, which provides a better understanding of the meaning of the word.

When calculating shortest paths with *reach*, some pre-processing of the graph is required since, to calculate *reach* for a vertex v, we need a shortest paths P. One way to do this is to calculate **All-Pairs Shortest Paths** using e.g. the Floyd-Warshall algorithm[2], but the paper[13] argues that in total, the approach avoids extensive pre-processing, while computed paths are "provably optimal" and that it delivers speed comparable to existing industry approaches. It is also worth noting that while the original paper uses Dijkstra's algorithm, *reach* works well with **A*** because the pruning mechanism of *reach* works similarly to the pruning mechanism in **A***[14].

ALT

ALT is an acronym for algorithms that use the **A*** algorithm, **L**andmarks and the **T**riangle inequality[15]. Landmarks are vertices that are selected by some random or domain-specific criterion and elevated into a more significant status. As a pre-processing step, the shortest path between all vertices in the graph and these landmarks is pre-computed and stored, and this information together with the triangle inequality provides lower bounds on distances between vertices. In other words, we can then quickly look up a distance between two vertices, and we know that the true distance is no less than this distance. This quickly and effectively prunes a large number of vertices from the search space without having to relax its corresponding edges, and very fast search of a graph can be accomplished.

Round-based routing

Round-based routing is an example of an alternative method not based on edge relaxation. It is a method that does not use an underlying graph representation, and it is developed for use on public transport networks, on which it exploits properties that are difficult to handle efficiently by relaxation-based methods[16]. Buses and trains operate on pre-defined lines, and this can be used to create an algorithm that explores routes from source to target with an increasing amount of transfers between lines. This means that it iteratively finds routes with k transfers, where $k \in [0, 1, 2, ..., n]$, i.e. it initially searches for a direct route (0 transfers), then a route with 1 transfer, 2 transfers, and so on. When no further improvements were found to a previously calculated route, the algorithm can be terminated. The method does not rely on pre-processing, which means that it works in scenarios where public transportation is constantly ahead or behind schedule, and it computes Pareto-optimal journeys.

2.2.2 Algorithms

This section examines some algorithms. The main algorithm is Dijkstra's algorithm which is the algorithm that is implemented in the prototype, but it also takes a look at Bidirectional Dijkstra, A*, and Relational Bidirectional Set-Dijkstra respectively, which are variants of Dijkstra's algorithm optimized for specific use cases.

In addition to this, the RAPTOR algorithm is described, which is an example of round-based routing.

Dijkstra's algorithm

E. W. Dijkstra developed an algorithm based on relaxation[17] which solves the Single-Source Shortest Path problem. It assumes a graph $G = (V, E)$ which can have both directed and undirected edges with and without edge weights, but if it has edge weights these must be non-negative. Each vertex $v \in G.V$ has the following two attributes:

- $v.d$: Distance from source $s \in G.V$ to v.
- $v.\pi$: The predecessor (node before) v when traversing the shortest path in the graph from the source s to v

Algorithm 1 INITIALIZE SINGLE SOURCE(G, s) *Algorithm from [2]*

```
for each vertex $v \in G.V$ do
    $v.d = \infty$
    $v.\pi = NIL$
end for
$s.d = 0$
```

Algorithm 2 RELAX(u, v, w) *Algorithm from [2]*

```
if $v.d > u.d + w(u, v)$ then
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
end if
```

Algorithm 3 DIJKSTRA(G, w, s) *Algorithm from [2]*

```
INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q$ != $\emptyset$ do
    $u$ = EXTRACT-MIN($Q$)
    $S = S \cup u$
    for each vertex $v \in G.Adj[u]$ do
        RELAX(u, v, w)
    end for
end while
```

The algorithm is given in Algorithm 3 which is adapted from [2], but the reader is also encouraged to watch a video simulation of the algorithm, such as [18]. From this, it can be seen that algorithms based on relaxation such as Dijkstra's algorithm begins with estimates for the distances between the source and the other vertices which are initially overestimated. In Dijkstra's algorithm, this initialization is done in Algorithm 1, where $v.d$ is set to infinity, except for the source which is set to 0, and all predecessors are set to NIL.

Next in Algorithm 3, the set S is initialized to empty, and Q which is a min-priority queue[2] that is ordered using the distance $v.d$ of $v \in V$, is initialized with all the vertices in the graph. S will throughout the runtime of the algorithm contain all the finalized vertices, which are the vertices where the length of the shortest path from the source have been determined.

Now, the main loop of Algorithm 3 is reached, where the vertex u with the minimum distance $u.d$ is extracted from Q. Then, each neighbor of u is relaxed (Algorithm 2) and u is inserted into S and thus finalized. When Q is empty the algorithm terminates.

It is worth noting that Dijkstra's algorithm produces the optimal shortest path and that most known shortest-path algorithms are based on the idea of edge relaxation[2], which is the basic idea first seen in Dijkstra's algorithm.

Bidirectional Dijkstra's algorithm

This algorithm performs simultaneous (but not parallel) *forward* and *backward* searches[19]. The *forward* search starts at the given source vertex, while the *backward* search starts at the given target vertex, and this technique will typically reduce the number of total visited vertices in a search. Formally, the *forward* search is performed on the normal graph $G = (V, E)$, while the *backward* search is performed on the graph $\bar{G} = (V, \bar{E})$ where $\bar{E} = (u, v) \mid (v, u) \in G.E$, i.e. the edges are reversed. Both the *forward* and the *backward* graph will have its own set for the finalized vertices (S_f and S_b) and its own priority queue (Q_f and Q_b). The algorithm terminates when the vertex $v \in G.V$ (note that $G.V$ is common to the two graphs) is found in $S_f \cup S_b$, i.e. the two searches have reached the same vertex.

One useful fact of bidirectional Dijkstra's algorithm is that it does not need to make additional assumptions compared to the regular Dijkstra's algorithm.

A*

A*[20] is an extension of Dijkstra's algorithm which changes the ordering of the priority queue Q of Dijkstra's algorithm to reflect not only the distance from the source vertex s but also reflect a heuristic which expresses the desirability of a vertex, similar to reach-based routing in Section 2.2.1. This new ordering directs the search of a target node towards the goal and effectively prunes a large part of the search space by ignoring vertices that seem to lead away from the goal when selecting the next vertex to relax. The reader is encouraged to watch [21] for a visualization of the difference in visitation pattern of Dijkstra's algorithm and A*, and consequently the difference in performance for the two algorithms resulting from the lower number of edges that are relaxed.

Looking at A* in more detail, it can be seen that Q is ordered by the ordering function $f(v) = v.d + h(v)$ where $v.d$ is the distance from the source vertex for a vertex $v \in G.V$, and $h(v)$ is the heuristic function which estimates the cheapest path from v to the target. An often used $h(v)$ is the shortest possible distance (straight line in a Cartesian coordinate system) from v to the target. A* is guaranteed to return a path with the least possible cost from source to target when $h(v)$ never overestimates the actual cost from v to the target. In this case we call $h(v)$ *admissible*, and this is clearly true for the shortest possible distance discussed above. It is also worth noting that when $h(v) = 0$, A* is equal to Dijkstra's algorithm.

A* makes the same assumptions as Dijkstra's algorithm and additionally needs to assume an *admissable* heuristic function.

Relational bidirectional Set Dijkstra

An interesting approach is called Bidirectional Set Dijkstra[22]. It uses new features of recent SQL Standards and performs an efficient version of bidirectional Dijkstra's algorithm with normal tables in a relational database, SQL operators, and window functions. This approach also handles the case where the graph does not fit into memory, and this is achieved by storing both the graph and all relevant runtime information in normal tables, and fetching this information into memory by SQL statements as the algorithm progresses.

The naíve implementation of this approach results in an excessive need for joins, and thus gives poor performance. Window functions and merge statements, new features of the SQL standard at the time, were used to improve this performance. Additionally, the approach evaluates sets of vertices at a time, instead of one single vertex at a time, since this takes better advantage of scheduling in modern database management systems.

Since this method is just an implementation of Dijkstra's algorithm with SQL operators, it has equal assumptions to Dijkstra's algorithm.

RAPTOR

Round b**A**sed **P**ublic **T**ransit **O**ptimized **R**outer (**RAPTOR**) is an example of an algorithm which is not based on relaxation. This is a round-based algorithm, which exploits the inherent structure present in public transportation. The paper[16] specifically defines the problem as the **Arrival Problem**, in which a source stop s_s, a target stop s_t and a time of departure t_d is used to calculate a journey which departs s_s no earlier than t_d, and where the arrival at s_t is as early as possible.

In place for a graph, RAPTOR introduces the concept of a timetable. The timetable contains all available information for a public transport domain (e.g. **Transport of London**), and the timetable also maps departures of public transport to stops and points in time. A journey is calculated from an entry stop to an exit stop and works in rounds as described in Section 2.2.1.

2.2.3 Optimizations

Reducing the cost of routing is a well-studied problem, and researchers have come up with various approaches depending on their focus. Since none of these address the exact problem of this thesis, this subsection is a brief overview of related results.

The most common focus in the literature is to reduce the time cost of routing, and the authors of [19] discuss several techniques for speeding up Dijkstra's algorithm. Three well-known techniques are to change the execution of the algorithm such that it is performed from both source and target simultaneously (Bidirectional Dijkstra), to add a heuristic to the ordering of the priority queue, and make it a goal-oriented search (A*), or using landmarks.

Another approach is to exploit a domain-specific hierarchy in the problem. This technique is a part of a broader set of pre-processing techniques[23] that performs some processing of the graph earlier than query time and stores this information in a way that keeps the routing algorithm from having to traverse the entire graph, and thus reduce the cost of routing. Since routing on road networks is a common task, and roads can be classified from dusty local roads to interstate highways, this is a commonly used domain-specific hierarchy. It is called **Highway Hierarchies**. In this case, highways are deemed most important and are for this reason selected in the shortest path before less important roads, with only a slight modification of Dijkstra's algorithm. An example of **Highway Hierarchies** is the special case of **Contraction Hierarchies**[23], in which a process called *contraction* calculates and adds shortcuts to the graph. This results in the expansion of fewer vertices during the execution of the search for the shortest path, and an example figure based on the top part of Figure 2.1a can be seen in Figure 2.3. Here, a shortcut is added from s to t, and it can be seen that by using the shortcut (solid line) the path has the same cost, but fewer vertices have to be expanded in between than if using the old path (dashed line), which results in less work and lower cost. When performing this *contraction*, the vertex selected for *contraction* is selected based on a prioritization of the vertices, in which the most important factor is how many edges would be removed (or added) to the graph if this vertex was contracted.

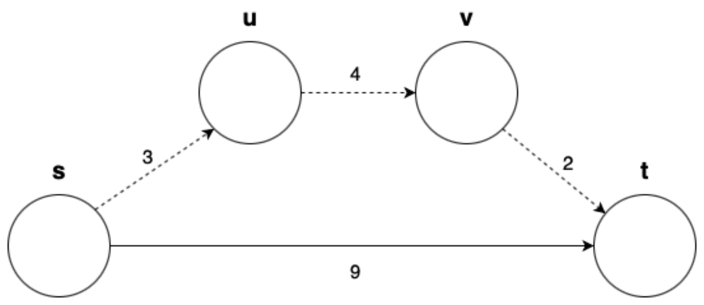


Figure 2.3: Example of contraction when adding a shortcut from *s* to *t*. Original path is shown with dashed line, shortcut is shown with solid line.

In [13] it is argued that this approach cannot guarantee the optimality of the returned result, and the same paper also proposes **reach-based routing** which is discussed in Section 2.2.1.

It is much less common to see memory cost as the focus, even though a consequence of some of the speed-up techniques in [19] are prohibitively large memory cost. This is somewhat understandable since it is often less cost and effort to add more memory to a system than it is to add more computational power.

It is also worth noting that memory cost is often called memory *requirement* in the literature, which indicates an approach where the algorithm "requires what it requires". One example where memory cost *is* considered is Dial's implementation of Dijkstra's algorithm[24] where the assumption of a low number of distinct edge weights leads to a memory optimization by placing vertices into buckets based on their distance from the source. The approach will not be detailed here since the required assumption about the data cannot be made in the implementation.

2.2.4 Imperative vs. declarative

SQL is a mostly *declarative* language, i.e. a language where a user describes **what** the wanted result is. This is often contrasted with *procedural* / *imperative* languages, where the user describes **how** the wanted result should be obtained. C++ is an example of a language that is imperative by default, even though it has recently introduced functional lambda functions which makes it more declarative.

As mentioned in Section 2.3.2 the SQL interface of pgRouting makes the user choose what algorithm to use, i.e. **how** the result should be obtained, and this makes the approach more *imperative* than *declarative*. This is a significant mismatch with the declarative nature of SQL, and it places the important responsibility of choosing the appropriate algorithm on the user such that the routing executes efficiently and correctly. Some users have this knowledge, similarly to how some users know how to most efficiently execute a join of several tables. But it can be argued that most users do not have this knowledge, and would prefer the database system to make this choice based on available data. For these reasons, a declarative syntax to trigger the computation of the shortest path would be preferable. This is also the suggested approach in the SQL/MM standard[11] which defines the two routines `ST_ShortestUndPath` and `ST_ShortestDirPath` for exposing functionality for shortest path computation in SQL.

Code listing 2.1: The syntax for calling the routine for directed graphs in SQL/MM

```
SELECT * FROM ST_ShortestDirPath(
                    paths_table_name           variable length character string,
                    path_id_column_name        variable length character string,
                    geometry_column_name       variable length character string,
                    path_start_column_name     variable length character string,
                    edge_weight_column_name    variable length character string,
                    start_point                ST_Point,
                    end_point                  ST_Point)
```

The syntax for `ST_ShortestDirPath` is shown in Listing 2.1 as an example. Here we see that the user specifies the names of tables and columns which contain the relevant data, in addition to the start and endpoints. No choice of algorithm is left to the user. The syntax for `ST_ShortestUndPath` is similar, and is found in [11].

2.2.5 Declarative routing

Since SQL provides a declarative user interface, as described in Section 2.2.4, a declarative way to invoke the computation of the shortest path would be preferable and ideally, it would also be compliant with the SQL/MM standard.

Making this choice has some implications. Firstly, it means that the choice of algorithm has to be decided within the routing module. Fortunately, this type of choice is common in a DBMS, since the optimizer always attempts to choose an efficient and cost-effective plan for every query made[25], while maintaining a semantically equivalent result. There are several relevant criteria, for example, the existence of negative cycles in the graph which would eliminate Dijkstra's algorithm for the benefit of Bellman-Ford's algorithm, and if an all-pairs shortest path computation is to be performed it would benefit from choosing the Floyd-Warshall algorithm[2].

Another implication is that memory usage becomes a determining factor. When the user chooses the algorithm, the DBMS cannot control how much memory is used, since this is deterministic from the algorithm chosen and the size of the graph. Consequently the DBMSs only choice is to abort the query if excessive memory usage is detected, and thus the responsibility for choosing the right algorithm is on the user. When the DBMS chooses the algorithm it must instead use available statistics about the graph and the runtime environment to optimize the choice of algorithm, similar to the way a non-routing query is optimized, since the user of the declarative interface will expect the query to finish in a reasonable time without considering technicalities such as available memory. Therefore the DBMS would have to solve any challenges, including exceeding memory limits, and this can be seen as a shift from a query having a memory *requirement* as discussed in Section 2.2.3, to the query having a memory *limit*, which then becomes another criterion for the choice of algorithm.

As mentioned in Section 2.3.1, MySQL runs on installations ranging from small single servers such as a Raspberry Pi to larger clusters working in distributed settings. The variation in available memory for queries in this range of installations, and the possible variation in the number of simultaneous users, makes it challenging to make assumptions about the available memory in the installations MySQL runs on. As a consequence of all this, the routing module must be able to handle the entire range, and display a gradual degradation of the quality of service but also make sure a result is returned eventually and within a reasonable time.

2.2.6 Cache Replacement Policies

Given a dataset D of size n blocks, and a cache $C \in D$ of size m blocks where $m < n$, the cache is only useful if it provides faster access to the data than does the main storage medium. In [26] design issues relating to cache design is categorized into the following categories:

- Cache size
- Block size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Number of Cache Levels

For our purposes, the most relevant parts are the **Mapping Function** and the **Replacement Algorithm**. The **Mapping Function** decides what location out of the m blocks in the cache a new block of data will occupy when it is read into the cache. This is related to the **Replacement Algorithm** because when a new block is read in another block will have to be replaced unless the cache is not yet filled. A flexible **Mapping Function** provides the **Replacement Algorithm** with more scope to design an algorithm that maximizes the number of hits per miss, which we want as large as possible. A reasonably effective **Replacement Algorithm** is the **Least Recently Used (LRU)** algorithm[26]. When a new element is to be inserted into the cache, a decision has to be made about which element should be removed. The LRU algorithm then finds the element that has been in the cache the longest without being referenced. Other, less effective algorithms are **First-in, First-out (FIFO)** and **Clock**.

2.2.7 Spatial Access Methods

In databases, the performance of queries can usually be increased by introducing an index. In datasets where data is mainly accessed by a single key (or ID) performance can e.g. be increased by using a B-tree[27]. When data is not easily identified or accessed by a single key or multiple keys with a total ordering, which is the case with geographic data that is often accessed using 2 or more coordinates, the B-tree is not efficient anymore.

To efficiently index spatial data, Spatial Access Methods (SAMs)[27] were created using a different construction principle than B-trees, and most of the SAMs follow one of the following two approaches:

- **Space-driven structures:** The space in which the objects to index are placed, is partitioned into equally sized, rectangular cells, independently of the distribution of data objects. The objects are then mapped to these cells using some geometric mapping function.
- **Data-driven structures:** The set of objects, not the space, is partitioned into cells of different sizes such that the partitioning is adapted to the distribution of the objects in the space.

The purpose of a spatial index is to efficiently assign spatial objects to a disk page. In both space-driven and data-driven structures, this happens by assigning each cell to a disk page - the difference is how nodes get assigned to these cells. The R-tree[27] is a data-driven spatial index that can handle large volumes of spatial data. It is based on two simple concepts:

- **Containment:** Nodes in the index are organized in a hierarchy of rectangles, where a lower level rectangle is *contained* within a higher level rectangle
- **Directory Rectangle:** The minimal bounding box which contains the rectangles of any child nodes of a rectangle

When reading from the index, the tree is traversed from the root and any Directory Rectangles which contain the query is further examined until a root node is found. A useful consequence of the tree structure is that access is logarithmic in the number of nodes.

When inserting into the index, the tree is searched for the smallest Directory Rectangle that contains the Minimum Bounding Rectangle of the object to be inserted. When the leaf node for such a Directory Rectangle is found, the object is either inserted if the leaf node exists, or a new leaf node is created if it does not.

More details on reading and inserting to an R-tree are found in [27].

2.3 Existing implementations

This section examines previous work which this thesis builds upon, including previous work in GIS in MySQL, and the implementations of pgRouting and the Oracle Routing Engine which are implementations of routing modules in other database systems than MySQL.

2.3.1 MySQL and GIS

As discussed in [28], MySQL is the most common choice for a relational database in modern web applications. It is used in both small installations such as a single server, and large, distributed clusters, and everything in between.

At the time of this writing, the latest released version of MySQL is 8.0.25. It has good support for GIS data, which is data that is related to a geographic position[28]. In other words, MySQL can be used for storing and querying information **and** the geographic data this information is related to, as well as perform operations on this data. Data about geometries such as `POINT`s, `LINESTRING`s and `POLYGON`s can be stored[29], and operations such as functions that create geometries in various formats and operations that describe relations between two geometries ("Do these geometries intersect" or "What is the distance between these two objects"?) can be performed on this data [30]. The implementation conforms to the OGC OpenGIS Implementation Standard for Geographic information[7], and it is based on Boost Geometry which is an efficient, generic, and peer-reviewed C++ library for geometric operations.

In the 8.0.23 release of MySQL[31] two functions called `ST_HausdorffDistance()` and `ST_FrechetDistance` are added to MySQL. These are functions that reflect how similar the shapes of geometries are, and one use case is to examine the similarity of two routes from A to B. Additionally, it can be used when comparing a planned route and an actual route, to see if the planned route was followed.

In the 8.0.24 release of MySQL[32], two other functions called `ST_LineInterpolatePoint()` and `ST_LineInterpolatePoints()` are added to MySQL. These are functions which take a `LINESTRING` and `fractional_distance` arguments and return one or several geographic `POINT(s)` along the route of the `LINESTRING` with the `fractional_distance` between them, not necessarily equal to the points that made the original `LINESTRING`. Since routing is usually performed on a collection of roads and intersections, as described in Section 2.1.3, this functionality can be used to split a `LINESTRING` into smaller points with an appropriate distance to create points that can function as intersections. Generally, these functions can be useful as pre-processing steps of the dataset before building the graph.

Currently, MySQL 8.0 cannot be used to answer routing queries such as "give me the shortest path from A to B". Supporting these types of queries would be a logical extension of the currently supported geographic queries, and would enable users with existing GIS data to compute shortest paths, especially since aggregation function such as `SUM` or `AVERAGE` already exist, and Single-Source routing can be seen as an aggregation of rows of a table describing a graph. Additionally, as discussed above, Oracle is very recently adding functionality that is related to and useful when executing routing operations to MySQL, and thus routing functionality is a natural fit in the future direction of MySQL.

2.3.2 pgRouting

PostgreSQL, a Database Management System (DBMS) that is one of the open-source competitors to MySQL, has a project called pgRouting[33] which is an extension to PostgreSQL that provides "routing and other network analysis functionality"[34]. pgRouting assumes a graph representation of the problem and implements most of the known algorithms for computing shortest paths in graphs. The graph representation for the routing problem is discussed in more detail in Section 2.1.1.

To compute a shortest path, an SQL query is sent to a PostgreSQL server that has the pgRouting extension enabled. The SQL query is of the form shown in Code Listing 2.2. The first parameter to `pgr_dijkstra` is a text string that must contain a SQL sub-query that `pgr_dijkstra` executes to fetch the rows from a table that describes the edges of a graph. From these rows, a graph is built as an internal data structure on which the shortest path is performed.

Code listing 2.2: Syntax for performing Dijkstra's algorithm in pgRouting

```
SELECT * FROM pgr_dijkstra(edges_sql, start_vertex, end_vertex)
```

Note that the user here is required to specify which algorithm to use for the routing. Dijkstra's algorithm is only one of many possible, well-known choices, and examples of other possible choices in pgRouting is Bellman-Ford's algorithm, or to perform a topological sort.

As mentioned in [33], one limitation of pgRouting is that it needs to process and add all edges that are returned from the `edges_sql` SQL query to the graph, to compute a path through the resulting graph. Moreover, pgRouting has no mechanism for controlling memory usage when computing a path, which consequently can lead to a situation where a large graph can make the server executing the algorithm run out of memory.

The format of the tables used in pgRouting are found in Appendix A.7.

2.3.3 Oracle

The Oracle Relational Database Management System (Oracle RDBMS) has an interesting implementation of routing, called the Oracle Routing Engine. The functionality is implemented as a web service in the Oracle RDBMS[35]. This means that it is outside of the database management service, and is not available by SQL, but it is available by XML/HTTP requests as a web service. Single-Source Shortest Path routing is supported, and it can be performed to one or several targets, and it can have several intermediary targets along the way. The problem with several intermediary targets can be solved as an instance of the Traveling Salesperson Problem (TSP)[3] where the web service tries to optimize the order of the targets, but since TSP is NP-complete the user is also given a choice to specify an order of the targets, which results in a possibly sub-optimal solution. Apart from this optimization choice which asks the user to specify **how** the result should be obtained, the user is only asked to specify the source, target, and intermediate targets, if any, and the web service thus provides a mainly declarative interface to routing, since the user is not given options on what algorithms to use, or asked to provide any further detail. The Routine Engine also takes memory limitation into account, since it only loads the subsets of the graph (called partitions in [35]) that are relevant into memory, and this partitioning is done as pre-processing when the graph is loaded into the database. This implementation is interesting since it applies the ideas discussed above of a declarative interface, as well as an active approach to memory management.

2.3.4 Boost Graph Library

Boost is a collection of C++ libraries that has a common goal of being portable between different systems while maintaining exceptional quality and good interoperability with the C++ Standard Library[36]. Boost Graph Library (BGL) is one such library, and it is implemented by graph algorithm experts. Furthermore, all Boost libraries go through peer-review, and consequently, BGL is generally thought of as near-optimal and error-free.

BGL has since the beginning had a strong emphasis on generic programming. For example, this means that it is not tied to specific data structures or algorithms, and it displays great flexibility in extending the available data structures and algorithms. Consequently, developers can tailor their use of BGL to take advantage of data structures that fit the needs of their application, or implement their own if needed.

Different types of graphs have different requirements, and BGL is adaptable to many of these requirements.[36] One example is the directionality of edges, and BGL supports **undirected**, **directed** and **bidirectional** edges. Informally and from a programmers perspective, in a **directed** graph the programmer has access to two lists of edges for each vertex $v \in G.V$, known as the *in_edge_list* and the *out_edge_list* which provides the edges that can be traversed to reach *v* and the edges that can be traversed from *v* to other vertices, respectively. In the **undirected** graph every edge from *v* is an out-edge and thus only the *out_edge_list* is available. The **bidirectional** graph is very similar to the undirected graph, except that it takes up twice the amount of space since each edge always appears in both *in_edge_list* and *out_edge_list*.

Users can also specify what type of memory representation to use for the graph, from the two representations discussed in Section 2.1.2. Given that an **Adjacency List** representation is chosen, which is the most common, BGL lets the user choose what data structures to use to store the *vertices* and *out-edges* (the list of edges out from a vertex). Additionally, all edges are stored in a separate data structure which is referenced by *out-edges* of vertices, and the type of this data structure is also chosen by the user. Details about the different choices are found in [37].

Chapter 3

Evaluation

This section will evaluate a possible implementation of routing in MySQL which observes a memory limit. In the case where memory consumption reaches a limit, this implementation maintains a memory buffer which is subsequently swapped to and from disk as needed.

Section 3.1 provides details about the evaluation that was performed, such as the datasets and how they were imported, details about the implementation, the hardware used for the tests, how the measurements were acquired, and specifically what the test was. Section 3.2 gives the results and discusses what memory and time consumption was observed, and what impact the pruning of vertices had. It also makes some observations about the impact of large I/O volumes in the current implementation.

3.1 Experimental setup

To implement routing in MySQL a built-in function was implemented, map extracts were acquired from OpenStreetMap, map data was loaded into MySQL tables, and the function was used to perform routing while statistics were collected.

3.1.1 Datasets

The implementation uses three datasets, which are extracts from OpenStreetMap's data in Norway. The datasets are of three sizes: **small**, **medium** and **large**. Each smaller dataset is a complete subset of the larger subset(s), and the contents of the datasets are as following:

Small: *Møllenberg*. A residential block in Trondheim city centre, Norway. See Figure 3.1

Medium: *Trøndelag*. The southern part of Trøndelag, Norway. See figure 3.2

Large: *Norway*. The entire graph of Norway. No figure is supplied.

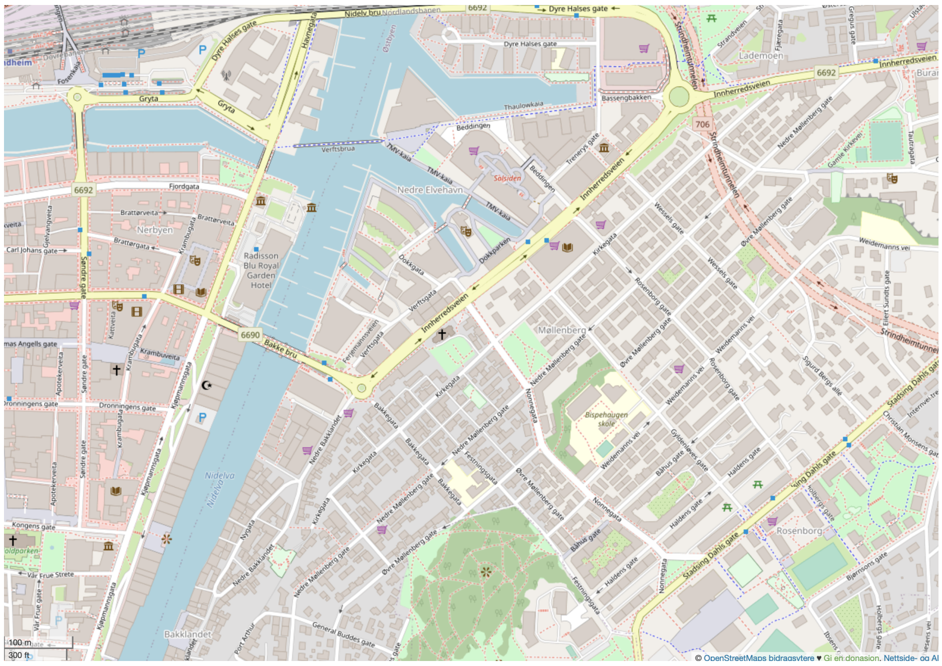


Figure 3.1: Map corresponding to the small dataset.
Bounding box: (long, lat) (63.42860, 10.39780), (63.43710, 10.42570)
© OpenStreetMap contributors

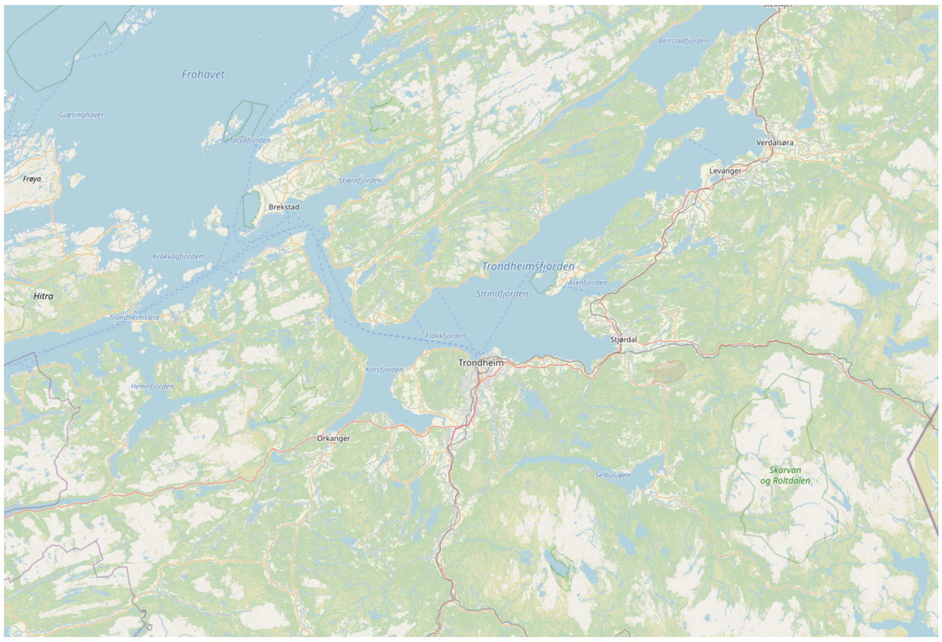


Figure 3.2: Map corresponding to the medium dataset.
Bounding box: (long, lat) (62.92400, 9.03630), (64.01570, 11.62900)
© OpenStreetMap contributors

The source of the datasets is OpenStreetMap[38] (OSM). OSM is an openly accessible map database distributed under the Open Database Licence[39] and thus the data is considered Open Data and can be used for any purposes without charge. The geographic data is collected by a community, and any registered member can make changes to the data. The veracity of such community-driven data can be questioned, and there are examples of incorrect maps and vandalism in the project, but for the most part, the data is available and correct and fits our purposes nicely. As an additional precaution, the data used for development was from the local area in Trondheim and could thus be verified for correctness, and no discrepancies were ever encountered.

There is currently no tool to import data from OpenStreetMap into MySQL. A tool called `osm2pgrouting`[10] exists for PostgreSQL, and this was used to transform data to an appropriate format and load it into MySQL. Data was stored as two tables called `ways` and `vertices`, containing *edges* and *vertices* respectively. Details about this process, and a more detailed description of the dataset is found in Appendix A.2.

3.1.2 Implementation

Implementing a working routing module in an existing database system such as MySQL is a time-consuming task. MySQL is already a large and mature system, and to get the routing module correctly integrated, several other existing modules such as the query parser and existing aggregation functions must be well understood. Furthermore, the routing functionality must be implemented using Boost Graph Library, and buffer replacement in `RVector` which includes proper handling of I/O using file handlers and (de)serialization using Boost Serialization, must be correctly performed. Integrating these was a significant challenge, especially since the author did not have any prior experience with any of these modules, and reaching the goal of a working prototype of a routing module with successful memory management is an important achievement in this thesis.

This section explains the reasoning behind some of the choices that were made in the implementation and describe the current functionality to the degree necessary to understand the test results. Further details are found in Appendix A.3.

Choices

There are many ways to implement functions in MySQL. An aggregation function (e.g. SUM or AVERAGE) takes rows according to some predicate and calculates a single, aggregate value for these rows. Routing in a graph can be seen as taking rows of edges, creating a graph from these, and calculating a shortest path for these edges, and thus an aggregation function provides a suitable environment for this operation. A disadvantage of this approach is that the shortest path is returned as a string value, which is difficult to process further unless it is in a standardized format.

Another relevant type of function would be a table function, where the returned result is a table. This would allow the calculated path to be returned as a table of rows, where each row could be the ID of a vertex in the shortest path and would allow easier further processing. This would allow the result to be further used in a query, i.e. in joins or selections. This is also the approach defined in the SQL/MM standard[11]. Table functions are more difficult to implement in MySQL, and time constraints prevented the implementation from using a table function. Furthermore, the correctness and prototype quality are not impacted by having the less optimal return value.

Consequently, routing was implemented as an aggregation function in MySQL. This implementation assumes all edges are undirected. The information to correctly handle one- and two-way streets as directed edges is present in the dataset, but only 0,6% of streets in the **large** dataset were known to be one-way only. Additionally, as discussed in Section 2.2.2, a consequence of storing directed edges is an increase in space requirement. Since most roads are bidirectional, the required space would be close to the case for a bidirectional graph, i.e. almost 2x the space requirement for a **undirected** graph. Modeling a **directed** system as an **undirected** graph is a potential cause for incorrect routes, but because of the low amount of directed edges in the dataset, the risk was judged as low, especially in a prototype. Also note that since we have chosen a declarative approach this would permit choosing appropriate graph representations depending on the dataset, although this was not implemented here.

There are many possible algorithms for finding the shortest path, and the declarative approach chosen here would allow many to be implemented. As noted in Section 2.2.2, most shortest-path algorithms are based on Dijkstra's algorithm, and this makes including it a natural starting point when implementing a prototype for a routing module. Many optimizations of Dijkstra's algorithm exists, and the alternative implementation of Dial[40] initially looks promising for decreasing memory usage. Unfortunately, the savings are predicated on having a low number of (integer) edge weights, and since most of the edge weights in the dataset are float-valued and distinct as seen in Table 3.1, in addition to the algorithm not being implemented in Boost Graph Library, the basic implementation of Dijkstra's algorithm in BGL was chosen. Again, the declarative approach would allow this approach to be chosen if the graph was found to have a low number of distinct edge weights.

As seen in Table 3.2 vertices in the graph created from OSM-data are not close to being connected to every other vertex. For example, we see that on average every vertex in the **medium** dataset is connected to 1.6 other vertices. For all the

Dataset	Edge weights	Distinct edge weights	Distinct percentage
Small	2064	2059	99,76%
Medium	159 580	159 503	99,95%
Large	1 198 126	1 197 547	99,95%

Table 3.1: Number of edge weights, distinct edge weights and the associated percentage for the three datasets

Dataset	Vertex count	Avg. number of edges from vertex
Small	1544	1.44
Medium	132 471	1.58
Large	1 147 572	1.27

Table 3.2: Number of vertices, and average number of other vertices a vertex is connected to in the datasets

datasets the average number of edges out from a vertex is lower than 2, and for the **small** and **large** datasets, it is closer to 1. This means that the graph is (*very*) sparse, and as discussed in Section 2.1.2, the **Adjacency List** representation is most efficient in that case.

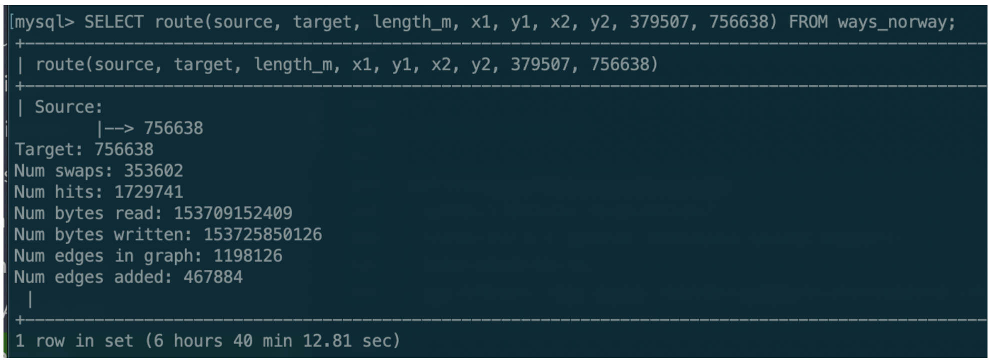
The full definition of the Boost **Adjacency List** is shown in Code Listing 3.1. This is a little cryptic, but it configures the graph as the following:

- **vecS:** Use a `std::vector` to store the **OutEdgeList**, i.e. the list of edges out from each vertex.
- **rVectorS:** Use `RVector` (the custom container developed in this project) to store the **VertexList**, i.e. the global list of all vertices in the graph
- **undirectedS:** The graph is undirected. This choice is justified in Section 3.1.2.
- **no_property:** No **VertexProperties** are defined, i.e. no extra information about each vertex is stored other than their ID.
- **property<edge_weight_t, double>:** Store the weight of each edge as an **EdgeProperty**, i.e. as extra information about the edge
- **no_property:** No **GraphProperties** are defined, i.e. no extra information about the graph is stored.
- **vecS:** Use a `std::vector` to store the **EdgeList**, i.e. the global list of all the edges to which **OutEdgeList** refers.

The implementation has been checked and verified for memory leaks using ASAN.

Code listing 3.1: Definition of Adjacency List used in the implementation

```
adjacency_list<
      vecS, rVectorS, undirectedS, no_property,
      property<edge_weight_t, double>, no_property, vecS>
```



```
[mysql> SELECT route(source, target, length_m, x1, y1, x2, y2, 379507, 756638) FROM ways_norway;
+-----------------------------------------------------------------------------------------------
| route(source, target, length_m, x1, y1, x2, y2, 379507, 756638)
+-----------------------------------------------------------------------------------------------
| Source:
        |--> 756638
Target: 756638
Num swaps: 353602
Num hits: 1729741
Num bytes read: 153709152409
Num bytes written: 153725850126
Num edges in graph: 1198126
Num edges added: 467884
  |
+-----------------------------------------------------------------------------------------------
1 row in set (6 hours 40 min 12.81 sec)
```

Figure 3.3: An example query and returned result. No shortest path was found for this query.

Syntax

To execute the routing function, the word `ROUTE` was added to the grammar of the MySQL parser. It matches queries of the form shown in Listing 3.2. The first 7 arguments to the `ROUTE` function are column names in the `ways` table, containing relevant information. The last 2 arguments are constants describing the source and target to route between in the graph.

Code listing 3.2: Example query for the ROUTE function

```sql
SELECT ROUTE(source_ids_column_name,
             target_ids_column_name,
             cost_column_name,
             source_longitude_column_name,
             source_latitude_column_name,
             target_longitude_column_name
             target_latitude_column_name,
             source_vertex_id,
             target_vertex_id)
  FROM edge_table_name
```

An example query is shown in Figure 3.3. This shows a query where no shortest path was found because results with shortest paths tend to have paths with many vertices, which do not fit in the format of this thesis. In the figure, the returned statistics are shown, as well as the elapsed time.

From the information passed to the `ROUTE` function, a graph is built from the resulting data. To perform the actual routing, Dijkstra's algorithm was performed using Boost Graph Library, as described in Section 2.3.4.

Memory limit

The implementation takes the data in the *ways* table, together with the *source* and *target* constants passed to the `ROUTE` function and creates a data structure on which it can perform the routing using Boost Graph Library. Because BGL is generic the programmer has a choice of what data structures to use when storing the edges and vertices in the `adjacency_list` representation, and these can also be custom data structures supplied by the programmer. Since all supported data structures in BGL are in-memory, a custom data structure called `RVector` is created, and it is used to store all vertices in the graph and observes the maximum allowed memory footprint for the routing function. It is worth noting that the graph is created by adding edges between vertices by calling the function outlined in Code Listing 3.3, which omits some details about the `add_edge` function for clarity. When adding these edges BGL will at any time make sure the number of elements in the data structure containing the vertices is equal to the largest value for `source` or `target` encountered so far. For example this means if we have a graph consisting of the edges `(1, 2)`, `(2, 3)` and `(3, 1000)` the data structure would have 1001 elements where only index 1, 2, 3 and 1000 would ever be used.

Code listing 3.3: A simplified outline of the function used to build the graph edge by edge. It returns the newly added edge

```
Edge add_edge(unsigned long source, unsigned long target, double weight);
```

The value of the memory limit, which is further discussed in Section 4.3.1, is by default set to 16 777 216 bytes, or approximately 17MB, on the system used for testing. Details about the test hardware are found in Section 3.1.3

`RVector` is implemented as a container adaptor[41], which is a container class that encapsulates an underlying container to provide additional functionality without having to re-implement functionality that already exists in another container type. `RVector` encapsulates a `std::vector` as the underlying container, and since the interface mimics the interface of `std::vector` the two can be used interchangeably. When the size of `RVector` is estimated to become larger than the maximum allowed memory footprint for the routing function it will split the data into several sub-buffers of smaller size and start to move data between a memory buffer and files on disk as needed. We call this operation *swapping* the memory buffer, and it happens when an index is requested that is not currently kept in the memory buffer (buffer *miss*). It consists of writing out the current contents of the memory buffer to a file and then reading the contents of the file containing the requested index. Since `RVector` presents the interface of `std::vector` any index can be randomly accessed at any time. To minimize time spent waiting on I/O, file handles were opened once and kept open until the end of the query. The maximum allowed memory footprint is given by the minimum of the two system variables `tmp_table_size` and `max_heap_table_size`, which existed in MySQL from before this implementation[42].

When the size does not exceed the limit, as shown in Figure 3.4a, all vertices are held in the memory buffer and routing is performed as if the vertices were stored in a `std::vector`. When the size *does* exceed the limit, as shown in Figure 3.4b, `RVector` will start to write and load data to and from disk files to keep the in-memory buffer within the limit.

To keep the `RVector` as generic as possible, the mapping from in-memory objects to files on disk was made using minimal domain knowledge. As discussed in Section 4.3.1, this may have been a constraint that unnecessarily impaired performance. Two procedures were created, shown in Code Listing 3.4 and 3.5. These procedures are fairly simple, and both calculate the maximum number of elements of type `T` that can fit in memory given a memory limit and the size of `T`. From this, the file index and element index are calculated using division and modulus respectively. As an example, assume an adjustment factor of 1 for now. Given a memory limit of 6, $n = 7$ and `sizeof(T)`$= 2$ the file index would be $fileIndex = 7/(6 * 1/2) = 2$ since we are performing integer division which is equal to the `floor`-function.

The adjustment factor is the only bit of domain knowledge introduced in this calculation. It effectively adjusts the size of `T` to make it closer to its real size. The reason for this inaccuracy in size and possible improvements are discussed in Section 4.3.1.

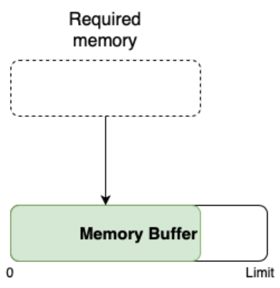
Code listing 3.4: The procedure for calculating which on-disk file an element is placed in, based on the index of the element in RVector

```
size_type getFileIndex(size_type n) const {
  const size_type max_vec_size = ram_limit_ * ADJUSTMENT_FACTOR / sizeof(T);
  return n / max_vec_size;
}
```

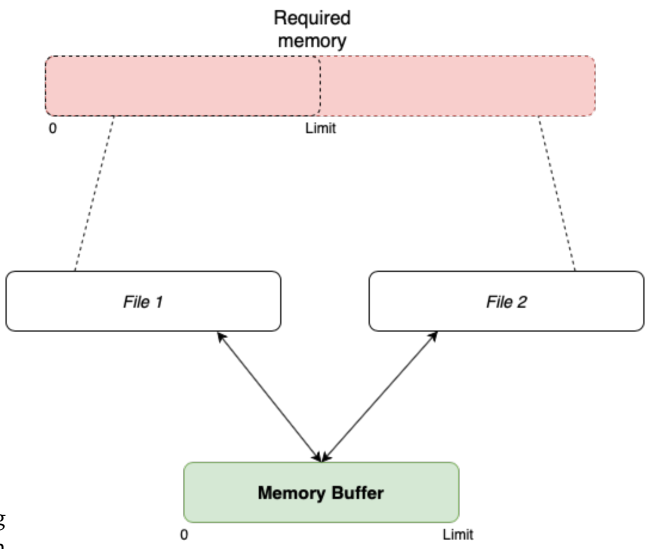
Code listing 3.5: The procedure for calculating what index in a given on-disk file an element will have, based on the elements index in RVector

```
size_type getElementIndex(size_type n) const {
  const size_type max_vec_size = ram_limit_ * ADJUSTMENT_FACTOR / sizeof(T);
  return n % max_vec_size;
}
```

To reduce the number of vertices in the graph, the implementation performs pruning based on the geographic position of vertices, as shown in Figure 3.5. The geographic midpoint between the source and target is calculated using `boost::geometry::distance`, and edges with a distance of more than the source-target distance from the midpoint are discarded. Since the efficient selection of vertices is not the focus of this project, the distance was set experimentally without much further consideration, but some edge cases where this might lead to sub-optimal results are discussed in Section 4.3.3.



(a) `RVector` when routing can be performed within memory limit



(b) `RVector` when routing will exceed memory limit

Figure 3.4: Figures showing RVector

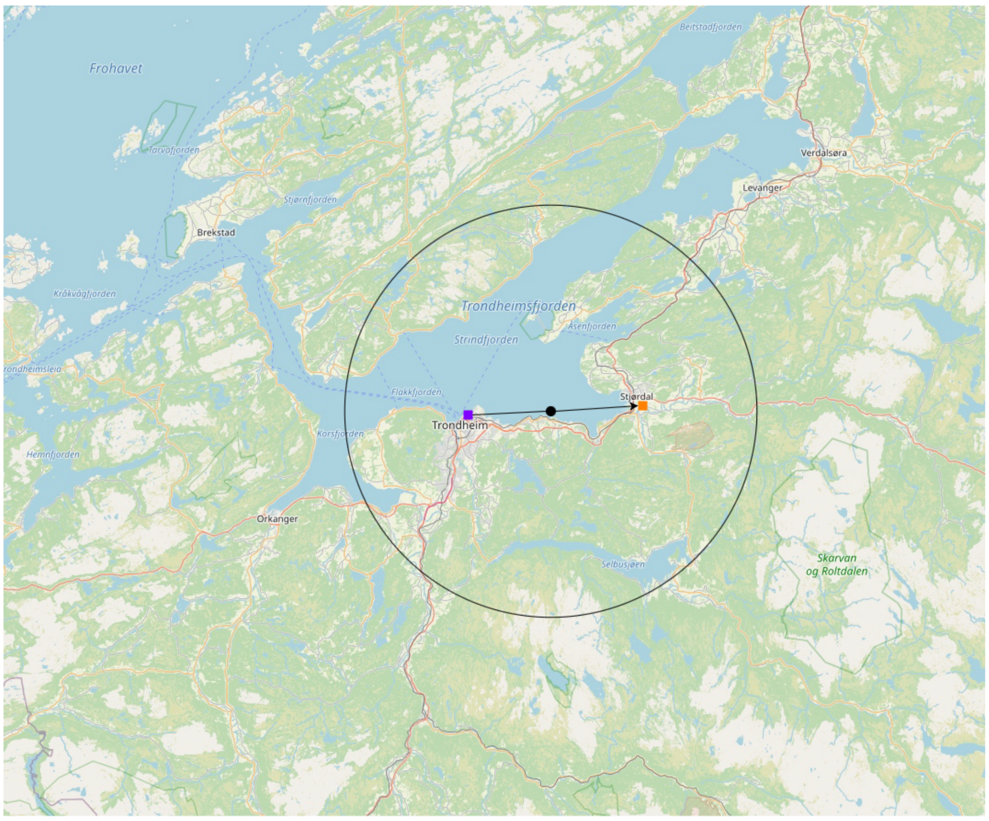


Figure 3.5: Map showing an example of vertex pruning when routing from the purple square to the orange square. Edges outside the black circle are discarded. © OpenStreetMap contributors

The code is available from https://github.com/strandlie/mysql-server/tree/routing and is explained in detail in Appendix A.3.

3.1.3 Hardware

The tests in this section were performed on a 2016 MacBook Pro laptop. It has a 2,7 GHz quad-core Intel i7 CPU and 16GB of RAM, which was sufficient for most operations. One exception is the handling and loading of the **large** dataset into MySQL, in which it ran out of memory. This processing was done on a larger server based in AWS. The MacBook Pro is a common consumer laptop used by many developers, but it is not a typical computer for running a production MySQL server. This is a limitation of this test, and further experimentation running on a wider range of more typical servers would be interesting further work.

3.1.4 Measurements

The evaluation measures memory usage, time consumption, the number of times the memory buffer was replaced, and I/O in bytes read and written. To measure memory usage MySQL has integrated instrumentation, and this data is stored in a performance table when a custom allocator is supplied to STL containers such as `std::list` or `std::vector`. Memory usage is then sampled at a given interval, and it measures only the memory used by `RVector`. Two sampling intervals were used, every 0,1 seconds for memory limited queries and every 0,001 seconds for non-memory limited queries. This distinction was made because the memory-limited queries ran for a long time, and the log files became very large without adding any more information. Further details about this are found in Appendix A.4.

Data on time consumption was also collected from MySQL directly since it reports elapsed time when completing a query.

The metrics other than memory and time was collected using counters in the implementation. Since the I/O was handled by the implementation, the number of bytes read and written from disk was readily available using the `tellp()` and `tellg()` functions of `std::ofstream` and `std::ifstream` respectively. The number of *swaps* was also increased whenever a buffer *miss* was encountered.

3.1.5 Evaluations

The evaluation setup used the three datasets described in Section 3.1.1. From each of these datasets, 5 pairs of vertices were randomly selected using the query in Code Listing 3.6. For each of these 15 vertex pairs, 4 test runs of the code were performed and evaluated. For some of the pairs, no shortest path existed and the query returned an empty result, but most did have a path and the shortest path was returned.

Code listing 3.6: The procedure for selecting a random vertex-pair in a dataset

```
SELECT id FROM vertices_<dataset> ORDER BY RAND() LIMIT 2;
```

For the small and medium datasets, and for each of the 5 randomly selected vertex pairs in each dataset, the query is run 4 times where the only difference is that the memory limit is set to a fraction of the default memory limit. For each run, it is set to:

- **1:** 6/6 of standard - 16 777 216 bytes
- **2:** 4/6 of standard - 11 184 810 bytes
- **3:** 2/6 of standard - 5 592 064 bytes
- **4:** 1/6 of standard - 2 796 203 bytes

The MySQL server is shut down and restarted between each run of the test to reset any statistics and give each query a similar starting point.

Because of time constraints, and because the implementation performed large amounts of I/O and thus was very slow for larger graphs, the selection of vertex pairs from the large dataset was not random for vertex pair 4 and 5. The query in Code Listing 3.6 was used, but it was run several times until vertex pairs that pruned larger amounts of edges were found, such that the evaluation would complete in time. This is a weakness in the experimental evaluation, but since we have one vertex pair with a low amount of pruning some useful insights were still found.

Further details on how the evaluation was performed on a code level are found in Appendix A.4.

3.2 Results

Memory and time consumption results are given in Table 3.3, 3.4 and 3.5, in addition to Figure 3.6 and 3.7. In the tables, memory and time usage is presented as relative amounts (in percent) to the baseline, to clearly show the trends in how memory and time consumption varies together since the absolute values have a large number of digits and can be difficult to comprehend. The baseline is a test run without any memory restriction.

I/O results are given in Table 3.6 and 3.7. Since the baseline is a test run without any memory restriction, it does not have any I/O, and the I/O results are consequently given as absolute values.

Within each dataset, the vertex pairs are numerated from 1 to 5. For reproduction purposes, the ids from OpenStreetMap are found in Appendix A.6.

For space, the columns of the tables were given brief names. The full meaning of these are:

- **Pair:** The identifier of the vertex pair used in a test run
- **Pruned:** The percentage of edges pruned from the policy described in Section 3.1.2.
- **Limit:** A fraction of the standard memory limit, as described in Section 3.1.5
- **Limited:** If the memory usage of the test run was *actually* limited by the limit
- **Memory:** How much memory the test run consumed, relative to a test run when the query ran without a memory restriction. Can be thought of as *memory savings*
- **Time:** How much time the test run consumed, relative to a test run when the query ran without a memory restriction. Can be thought of as *slowdown*
- **Hit ratio:** The percentage of requests to `RVector` which found its content in the buffer, relative to all requests to `RVector` throughout the test run
- **Num. files:** The number of disk files created for the test run
- **Max file size:** The size of the largest disk file created in the test run
- **Total I/O:** The sum of the number of bytes read and written during the test run

3.2.1 Summary of the results

From the results, it is evident that the implementation is successful in limiting the amount of memory consumed, but when memory is limited and I/O is performed it is slowed down significantly. The *least* amount of slowdown when the query is memory limited (the best performance) is seen for vertex pair 3 in the medium dataset with the 2 / 6 memory limit, in which the time consumption is 84 000 % higher than when the query is not memory limited. This query performs 4,6 GB of I/O, which is the lowest amount. The *highest* amount of slowdown (the worst performance) is seen for vertex pair 1 in the large dataset with the standard memory limit, in which the time consumption is 8 913 867 % higher than without any memory limitation. This query performs 3,7 TB of I/O, which is the highest amount.

From this, it can be seen that performance is mainly impacted by the amount of I/O performed. High amounts of I/O lead to poor performance, and low amounts lead to better performance. One might expect that a higher hit ratio in the buffer would also lead to better performance, and this is the case for vertex pairs in the medium dataset where more than 40 % of edges are pruned away, i.e. vertex pair 1, 3, and 5. But it is not the case for the vertex pairs with the least amount of pruned edges in the medium dataset (vertex pairs 2 and 4), and all vertex pairs in the large dataset. Thus, the results show that when a graph has many edges, such as in the large dataset or the medium dataset with little pruning, it is beneficial to have more files of a smaller size, and thus reduce the overall amount of I/O, and the benefit of this shadows the benefit of a better hit ratio. Where the graph contains fewer edges, the hit ratio is the determining factor of the performance.

We also see that the number of files was equal for a given memory limit across vertex pairs, which is as expected since the assignment to files is decided with the memory limit as the only varying factor, as discussed in Section 3.1.2.

3.2.2 Memory consumption

From the results, we see that the memory usage for a given memory limit is equal for all vertex pairs (i.e in Table 3.4 all runs with the limit 4/6 has a memory usage of 56% relative to an unrestricted run). This means that the implementation is successful in limiting the actual memory usage. It is also interesting to note that the number of files increases as the memory limit is decreased. Since the size of the available buffer is smaller, and the entire file is read into the buffer when an element is referenced from it, decreasing file sizes indicate that the memory limit is respected.

3.2.3 Time consumption

For the small dataset in Table 3.3 we see that the test runs complete without any slowdowns because they are not memory limited. For the medium dataset, it does not look so good. The test runs that run with standard and 4/6 memory limit complete without slowdowns, but the test runs with 2/6 and 1/6 memory limit show slowdowns between 84 000% in the best case, and 6 038 158% in the worst case. For the large dataset, we see similar figures with slowdowns between 328 407 % and 8 913 867 %. These increases in time consumption could be justified if the memory consumption was decreasing at a similar rate, but as seen in table 3.4 and 3.5 this is not the case. The best time-to-memory-change ratio is found for vertex pair 5 in the large dataset.

3.2.4 I/O

Looking at the I/O results in Table 3.6 and 3.7, it can also be seen that time slowdown is strongly correlated with the amount of I/O performed. For all vertex pairs in the large dataset, more time consumed is followed by larger amounts of I/O. The medium dataset has a similar situation, except for vertex pair 2, in which the situation is reversed.

Generally, the implementation shows very large amounts of I/O, and reducing this to a minimal amount seems to be an important step towards improving performance.

3.2.5 Impact of pruning

The impact of pruning is best shown in the results for the medium and large datasets. In the medium dataset, comparing vertex pairs 3 and 4 in which 88 % and 0 % of edges were pruned respectively, it can be seen that the differences are significant. When it is possible to prune many edges, it leads to a decrease in consumed time by at least an order of magnitude, in addition to an increased hit ratio, while consuming the same amount of memory.

It is worth noting that if the implementation did not perform pruning, all vertex pairs in the medium dataset would have approximately the same performance as vertex pair 4 in which no edges are pruned. In other words, even the unsophisticated pruning in this implementation is effective.

3.2.6 Memory limit

It can be observed from Figure 3.8 that the implementation does impose a memory limit, but this memory limit is higher than the number calculated by the implementation. In other words, the implementation consumes more memory than it "should". Figure 3.8 is the taken from vertex pair 1 in the medium datasets, but the same behavior was observed for all vertex pairs in all datasets.

Pair	**Pruned**	**Mem. limit**	**Limited**	**Memory**	**Time**	**Hit ratio**
1	15%	Standard	No	100 %	100 %	100 %
1	15%	4 / 6	No	100 %	100 %	100 %
1	15%	2 / 6	No	100 %	100 %	100 %
1	15%	1 / 6	No	100 %	100 %	100 %
2	23 %	Standard	No	100 %	100 %	100 %
2	23 %	4 / 6	No	100 %	100 %	100 %
2	23 %	2 / 6	No	100 %	100 %	100 %
2	23 %	1 / 6	No	100 %	100 %	100 %
3	85 %	Standard	No	100 %	100 %	100 %
3	85 %	4 / 6	No	100 %	100 %	100 %
3	85 %	2 / 6	No	100 %	100 %	100 %
3	85 %	1 / 6	No	100 %	100 %	100 %
4	66 %	Standard	No	100 %	100 %	100 %
4	66 %	4 / 6	No	100 %	100 %	100 %
4	66 %	2 / 6	No	100 %	100 %	100 %
4	66 %	1 / 6	No	100 %	100 %	100 %
5	27 %	Standard	No	100 %	100 %	100 %
5	27 %	4 / 6	No	100 %	100 %	100 %
5	27 %	2 / 6	No	100 %	100 %	100 %
5	27 %	1 / 6	No	100 %	100 %	100 %

Table 3.3: Summary of the time and memory results from test runs on the small dataset

Pair	Pruned	Limit	Limited	Memory	Time	Hit ratio
1	44 %	Standard	No	100 %	100 %	100 %
1	44 %	4 / 6	No	100 %	100 %	100 %
1	44 %	2 / 6	Yes	56 %	1 739 800 %	92 %
1	44 %	1 / 6	Yes	28 %	2 290 000 %	82 %
2	18 %	Standard	No	100 %	100 %	100 %
2	18 %	4 / 6	No	100 %	100 %	100 %
2	18 %	2 / 6	Yes	56 %	5 895 700 %	89 %
2	18 %	1 / 6	Yes	28 %	4 524 800 %	81 %
3	88 %	Standard	No	100 %	100 %	100 %
3	88 %	4 / 6	No	100 %	100 %	100 %
3	88 %	2 / 6	Yes	56 %	84 000 %	98 %
3	88 %	1 / 6	Yes	28 %	162 900 %	92 %
4	0 %	Standard	No	100 %	100 %	100 %
4	0 %	4 / 6	No	100 %	100 %	100 %
4	0 %	2 / 6	Yes	56 %	6 038 158 %	87 %
4	0 %	1 / 6	Yes	28 %	5 917 105 %	79 %
5	63 %	Standard	No	100 %	100 %	100 %
5	63 %	4 / 6	No	100 %	100 %	100 %
5	63 %	2 / 6	Yes	56 %	695 806 %	96 %
5	63 %	1 / 6	Yes	28 %	1 342 903 %	86 %

Table 3.4: Summary of the time and memory results from test runs on the medium dataset

Pair	**Pruned**	**Mem. limit**	**Limited**	**Memory**	**Time**	**Hit ratio**
1	12 %	Standard	Yes	17 %	8 913 867 %	89 %
1	12 %	4 / 6	Yes	14 %	6 354 609 %	88 %
1	12 %	2 / 6	Yes	7 %	4 131 563 %	84 %
1	12 %	1 / 6	Yes	3 %	3 087 773 %	76 %
2	61 %	Standard	Yes	17 %	2 766 197 %	93 %
2	61 %	4 / 6	Yes	17 %	2 114 742 %	92 %
2	61 %	2 / 6	Yes	7 %	1 483 380 %	89 %
2	61 %	1 / 6	Yes	3 %	1 127 371 %	83 %
3	46 %	Standard	Yes	17 %	4 477 892 %	92 %
3	46 %	4 / 6	Yes	14 %	3 346 814 %	91 %
3	46 %	2 / 6	Yes	7 %	2 296 471 %	87 %
3	46 %	1 / 6	Yes	3 %	1 671 078 %	81 %
4	70 %	Standard	Yes	17 %	1 256 827 %	95 %
4	70 %	4 / 6	Yes	14 %	1 135 301 %	94 %
4	70 %	2 / 6	Yes	7 %	785 422 %	92 %
4	70 %	1 / 6	Yes	3 %	624 177 %	86 %
5	93 %	Standard	Yes	17 %	1 173 077 %	94 %
5	93 %	4 / 6	Yes	14 %	898 626 %	93 %
5	93 %	2 / 6	Yes	7 %	523 901 %	93 %
5	93 %	1 / 6	Yes	3 %	328 407 %	91 %

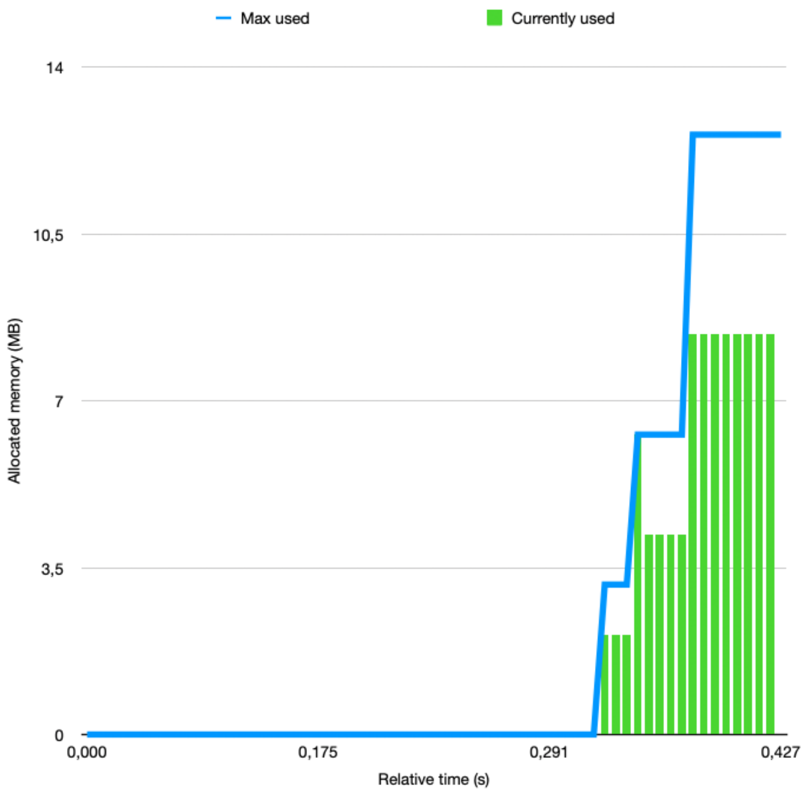
Table 3.5: Summary of the time and memory results from test runs on the large dataset

Pair	Pruned	Mem. limit	Limited	Num files	Max file size	Total I/O
1	44 %	Standard	No			
1	44 %	4 / 6	No			
1	44 %	2 / 6	Yes	2	2,2 MB	111,0 GB
1	44 %	1 / 6	Yes	4	1,2 MB	146,7 GB
2	18 %	Standard	No			
2	18 %	4 / 6	No			
2	18 %	2 / 6	Yes	2	2,6 MB	219,2 GB
2	18 %	1 / 6	Yes	4	1,4 MB	242,1 GB
3	88 %	Standard	No			
3	88 %	4 / 6	No			
3	88 %	2 / 6	Yes	2	1,4 MB	4,6 GB
3	88 %	1 / 6	Yes	4	0,73 MB	9,1 GB
4	0 %	Standard	No			
4	0 %	4 / 6	No			
4	0 %	2 / 6	Yes	2	3,0 MB	361,9 GB
4	0 %	1 / 6	Yes	4	1,5 MB	362,7 GB
5	63 %	Standard	No			
5	63 %	4 / 6	No			
5	63 %	2 / 6	Yes	2	1,5 MB	26,6 GB
5	63 %	1 / 6	Yes	4	0,97 MB	59,5 GB

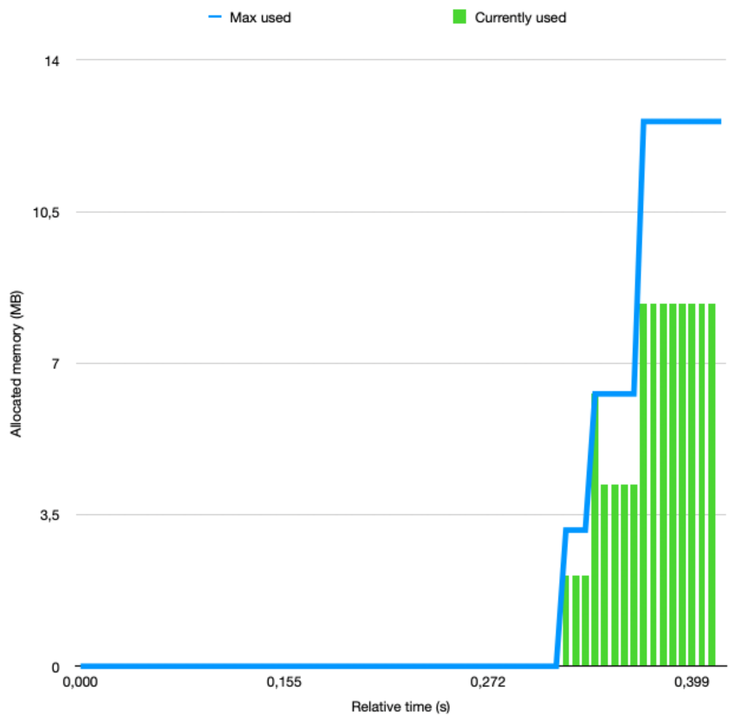
Table 3.6: Summary of the I/O results from test runs on the medium dataset

Pair	Pruned	Mem. limit	Limited	Num files	Max file size	Total I/O
1	12%	Standard	Yes	5	8,0 MB	3,7 TB
1	12%	4 / 6	Yes	7	5,5 MB	2,8 TB
1	12%	2 / 6	Yes	14	2,8 MB	1,8 TB
1	12%	1 / 6	Yes	27	1,4 MB	1,2 TB
2	61%	Standard	Yes	5	4,3 MB	820,7 GB
2	61%	4 / 6	Yes	7	3,0 MB	638,2 GB
2	61%	2 / 6	Yes	14	2,1 MB	422,2 GB
2	61%	1 / 6	Yes	27	0,85 MB	307,4 GB
3	46%	Standard	Yes	5	5,4 MB	1,3 TB
3	46%	4 / 6	Yes	7	3,6 MB	1,0 TB
3	46%	2 / 6	Yes	14	1,7 MB	687,9 GB
3	46%	1 / 6	Yes	27	1,0 MB	492,8 GB
4	70%	Standard	Yes	5	3,9 MB	442,4 GB
4	70%	4 / 6	Yes	7	2,5 MB	363,9 GB
4	70%	2 / 6	Yes	14	1,5 MB	245,8 GB
4	70%	1 / 6	Yes	27	0,75 MB	197,3 GB
5	93%	Standard	Yes	5	1,6 MB	224 GB
5	93%	4 / 6	Yes	7	1,2 MB	168,0 GB
5	93%	2 / 6	Yes	14	0,61 MB	100,7 GB
5	93%	1 / 6	Yes	27	0,40 MB	61,8 GB

Table 3.7: Summary of the I/O results from test runs on the large dataset

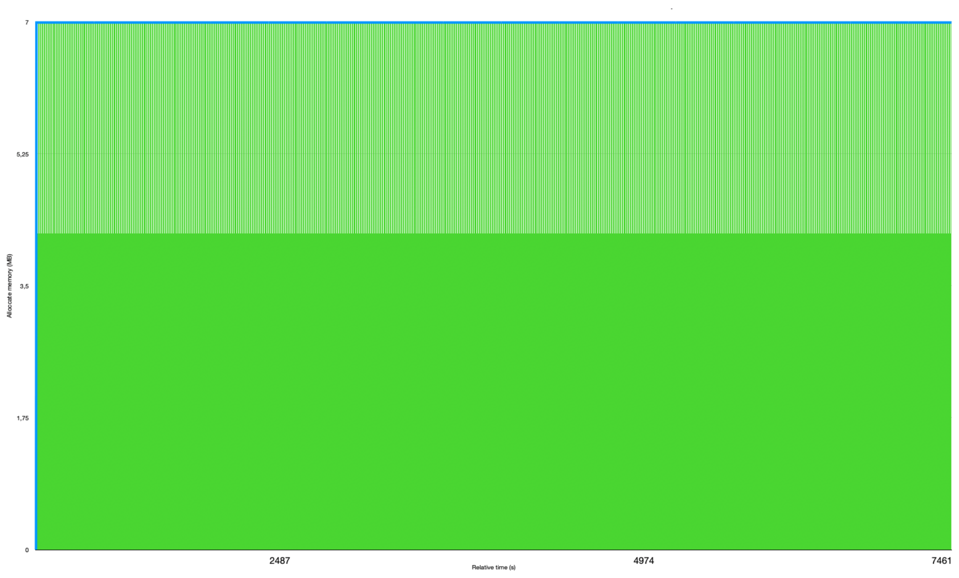


(a) Standard memory limit

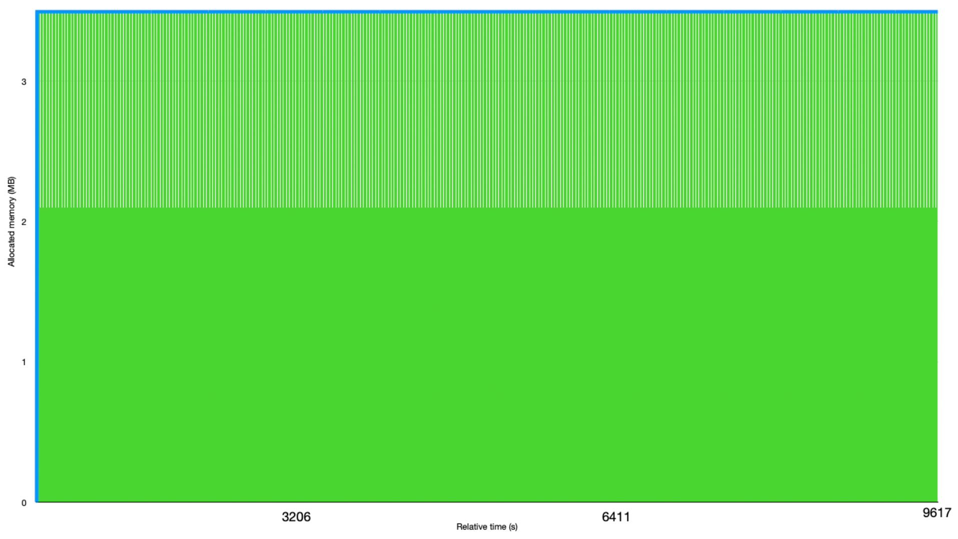


(b) Memory limit of 4 / 6 of standard

Figure 3.6: Memory usage (green bars) and max memory usage (blue line) at any point in the query for vertex pair 1 in the **medium** dataset (1)

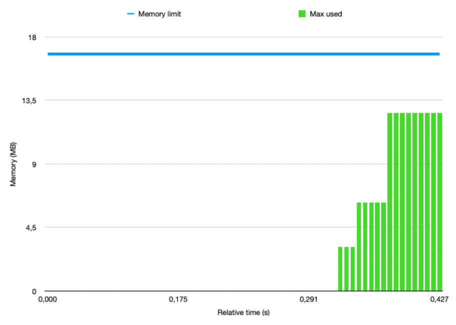


(a) Memory limit of 2 / 6 of standard

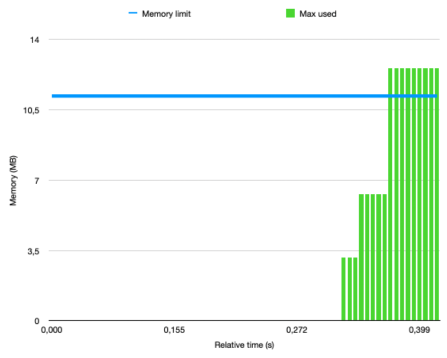


(b) Memory limit of 1 / 6 of standard

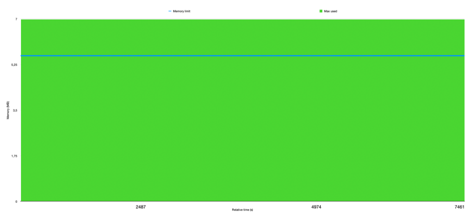
Figure 3.7: Memory usage (green bars) and max memory usage (blue line) at any point in the query for vertex pair 1 in the **medium** dataset (2)



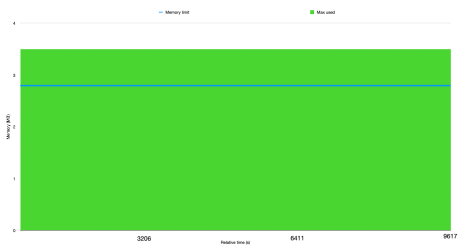
(a) Max memory: 12,58 MB **Limit:** 16,78 MB



(b) Max memory: 12,58 MB **Limit:** 11,18 MB



(c) Max memory: 6,99 MB **Limit:** 5,59 MB



(d) Max memory: 3,49 MB **Limit:** 2,80 MB

Figure 3.8: Graphs showing maximum memory consumption (green bars) through a query exceeding the memory limit (blue line) set for that query, for vertex pair 1 in the **medium** dataset

Chapter 4

Discussion

This section provides discussion on the datasets chosen, found in Section 4.1, and presents some thoughts on the results in Section 4.2, such as the practical usability of the current implementation, the importance of the hit ratio in the buffer, and the fact that better performance was observed with lower memory limits in the larger datasets. Then, in Section 4.3 details about the implementation of `RVector`, such as the buffering strategy and the memory limit is discussed, before some discussion on the current interface and the impact of pruning is presented in Section 4.3.2 and 4.3.3.

4.1 Datasets

As described in Section 3.1.5, we selected 5 vertex pairs from the **small**, **medium** and **large** datasets. This setup was chosen because it represents the experience of several types of users. Some users have very small datasets, some have very large datasets, while most have datasets of a size somewhere in between, and we wanted to test the experience for all of these. It can be argued that users of very small datasets will have no use for this since the small datasets most likely will fit within the memory limit. Nevertheless, it can also be argued that users with small datasets also have less powerful computers, and thus this should be a valuable test setup. Using this setup lets us test three sets of 5 queries of approximately the same size, instead of 15 queries of unknown size. This is especially important for managing time spent on running tests.

Additionally, it is worth noting that if we selected 15 vertex pairs from the largest dataset instead, and accidentally got the same vertex pairs as in the current setup, the implementation would mostly behave the same as now because of pruning, except that the vertices would be spread across more files. This would happen because vertices are indexed and placed into files with a local ID, and the keyspace for these local IDs is larger in the larger dataset since there are more vertices. Details about this local ID are found in Section 4.3.1.

4.2 Results

This section discusses some important aspects of the results obtained in Chapter 3.

4.2.1 Practial usability

As seen in Table 3.4, we see slowdowns of orders of magnitude when the query is limited by the memory limit. This limits the practical usability of the implementation since a query that finishes in less than a second when it is not memory limited can take hours or days to finish when it is limited.

4.2.2 Better performance with lower memory limit

For vertex pair 2 and 4 in the **medium** dataset, and all vertex pairs in the **large** dataset (Table 3.4 and 3.5) we observe an increase in performance (less time consumed) when the memory limit is decreased, when the opposite would be expected since the hit ratio is also lowered. The reason for this likely is that when the memory limit is decreased for vertex pairs with many edges, less data must be read and written to disk each time, which again leads to less time waiting for I/O, compared to when reading and writing larger files. This indicates that replacing the entire buffer each time we get a buffer *miss* might be decreasing the performance significantly due to excessive I/O, and more files of a smaller size is a better strategy.

These results also seem to indicate that when moving from small to large datasets with varying degrees of pruning, the amount of I/O, which is a result of the size of the disk files, becomes a more important factor in the time consumption, than the hit ratio. This can be seen from the results of the medium dataset in Table 3.4 and 3.6, where vertex pair 1, 3, and 5 experiences higher time consumption with a lower memory limit, while vertex pair 2 and 4 experiences the opposite. Since vertex pairs 2 and 4 have the least amount of pruned edges, and all vertex pairs in the large dataset in Table 3.5 and 3.7 experience lower time consumption with a lower memory limit, some limit seems to exist in the medium dataset with the current implementation where file size becomes the determining factor.

It is also worth noting that the implementation still has very poor performance with lower memory limits, even if it is better.

4.2.3 Importance of hit ratio

From Table 3.4 we also see the importance of the hit ratio. Hit ratio is defined as the proportion of requests that find their data present in the buffer, relative to all requests. For the **small** dataset, where all test runs fit into memory, we get a 100 % hit ratio and excellent performance. For the **medium** and **large** datasets, we

Vertex pair	Average time-slowdown	Average hit-ratio
1	5 621 953 %	84,25 %
2	1 872 923 %	89,25 %
3	2 948 064 %	87,75 %
4		
5	731 003 %	92,75 %

Table 4.1: Average time-slowdown and average hit-ratio for vertex pairs in the large datasets

observe a decrease in hit ratio when the memory limit is decreased, as expected. As discussed above, the effect of this is not observed as clearly as expected, particularly within test runs with the same vertex pair with a lowering memory limit. When comparing between vertex pairs, however, for example by calculating the average time consumption and average hit ratio, as seen in Table 4.1, we see that the average time consumption is lowest for the highest average hit-ratio, and vice versa. From this, we conclude that optimizing the hit ratio is still important.

4.3 Implementation

4.3.1 RVector

Using RVector for the entire graph

As discussed in Section 2.3.4 and 3.1.2, when using Boost Graph Library with the graph represented as an *Adjacency List*, the programmer can choose what data structures to use for storing the lists of edges and vertices independently (called EdgeList and VertexList from now, respectively). In this thesis, the `RVector` is only used for the VertexList, while it could also be used for the EdgeList. To simplify the evaluation and get clear results the project only introduces `RVector` for the VertexList, and thus keeps the number of "moving parts" as low as possible. Also using `RVector` for the EdgeList would have complicated the implementation, mainly in the serialization logic where the contents of the RVector is written to disk since `serialize`-functions are customized for many types used within the *Adjacency List* data structure. Given that the author had very limited experience with C++, Boost Libraries and MySQL server code this seems a reasonable choice in the available time frame.

On the other hand, not using `RVector` for the EdgeList does come with a substantial disadvantage. The memory usage of the routing module is not entirely under control, since the EdgeList can in general have any size, which can make the routing module exceed its memory limit. For this reason, it would be useful to look at using `RVector` for the EdgeList as well, even if the dataset used in this project is sparse as discussed in Section 3.1.2.

Genericity of RVector

`RVector` is implemented as a template class in C++. The ability to store a sequence of any element, similarly to `std::vector`, has obvious advantages and this was the rationale for doing it that way. As the implementation progressed, however, the issue of serialization arose. When writing the contents of `RVector` to disk, the elements and all their member variables need to have a procedure defined for serialization. This can either be defined in the class (i.e. by Boost Graph Library) or the implementation can define custom serialization (called non-intrusive serialization). BGL does define a serialization procedure for a full graph, but since our goal here is to serialize a partial graph, a large part of the work with this implementation became to define non-intrusive serialization logic for internal types used in the **Adjacency List** implementation. The advantage of this is that the implementation has great control over the serialization process. The disadvantage is that `RVector` is not *really* generic anymore, since using it with elements of new types requires new non-intrusive serialization methods to be implemented. An alternative solution would be if serialization support was implemented directly in BGL for **Adjacency List**, and serialization procedures were required for using elements with `RVector`. This would be a natural requirement since the purpose of `RVector` is to be able to spill over elements to disk when it gets too large.

Details about the implementation of serialization are found in Appendix A.3.2.

Buffering strategy

The buffering strategy (i.e. the cache replacement policy) in the current implementation is functional, but it has several disadvantages. Although the current implementation is limited by excessive amounts of I/O and not by hit-ratio to the same degree, this section argues that improving the buffering strategy would both increase the hit-ratio and decrease the amount of I/O.

Using the concepts from Section 2.2.6 and the implementation details from Section 3.1.2, we see that the two procedures `getFileIndex(n)` and `getElementIndex(n)` are effectively the **Mapping Function** of the buffer. These functions make a very inflexible **Mapping Function** since elements are deterministically assigned to files and positions within these files based on their index, and for this reason, all test runs for a given dataset with a given memory limit will have the same amount of files on disk. For example, any test run on any vertex pair in the large dataset with the 2/6 memory limit will always create 14 files on disk. As a consequence, the **Replacement Algorithm** is almost non-existent. It consists of checking the file index of an element given its index, and if this file index is not equal to the index of the file currently in memory, replace the contents of the buffer, which is currently the entire contents of the current file, with the entire contents of the new file.

The main advantage of this policy is its simplicity. It is very easy to follow and implement correctly. Also, this strategy performs very well for queries that are not limited by the memory limit since the buffer does not have to be "warmed" with elements at the beginning of the query, but performs like any other `std::vector`. This can be seen in the results from the **small** dataset in Table 3.3. Lastly, if "closeness" in element index in `RVector` entailed closeness in access patterns, i.e. if it were true that when index 10 was accessed it was more likely that indices 5 through 15 would be accessed next, this buffering strategy would be much more effective. Unfortunately, this is not always the case, because when the OpenStreetMap-data are loaded into MySQL tables using the procedure detailed in Appendix A.2, vertices are given a local ID in the database in addition to their global ID given by OSM. This local ID is used as an index into `RVector` when fetching vertices and their corresponding edge lists as part of edge relaxation in Dijkstra's algorithm, and since the number of vertices BGL creates is equal to the largest vertex ID encountered, as described in Section 3.1.2, this contraction of the keyspace saves significant space compared to using the global OSM id as an index and makes sure every index in `RVector` is a real vertex in the graph. The downside of this approach is that, as seen in Table 4.2 there is not a one-to-one correspondence between local and global IDs. The global IDs make jumps when the local IDs do not and they consequently diverge. Thus any "closeness" or distance that might be expressed through the indices will be over- or underrepresented which will decrease the efficiency of the principle of locality, which is the main motivation for using buffers. Thus, using the local index to assign vertices to disk files leads to poor performance since the principle of locality is not utilized efficiently.

Unfortunately, time constraints prohibited implementing an improved solution, but the following paragraphs will present a promising idea for improvement. Mainly we need to make our **Mapping Function** more flexible, since assigning file position based on the local index does not give satisfactory performance. The current implementation uses the geographic coordinates of the vertices for pruning the graph but then discards the coordinates. It is possible in BGL to store properties of a vertex[43], and by storing these coordinates as properties of the vertices the implementation could store and retrieve vertices from disk based on their geographic position, and essentially create a spatial index for the vertices. Fortunately, the database literature and industry have developed several spatial indices which could be useful here. The R-tree, as described in Section 2.2.7, could be very useful here. The R-tree maps geographic locations to disk pages, but transferring the

Local ID	OSM ID
20	78 272
21	78 273
22	78 274
23	78 275
24	78 277
25	78 278
26	78 299
27	78 309
28	78 311
29	78 312
30	78 315

Table 4.2: Examples of pairs of local-global IDs in the dataset

concept to files and thus mapping geographic locations to disk files would enable storing neighboring vertices in the same file, regardless of their local ID, and consequently lead to fewer misses in the buffer. The disadvantage of this approach is that it would require the implementation to store some information about which files vertices are stored in after insertion, while the current implementation just calculates this for each index without storing any information. Regardless, this would be an acceptable trade-off since it would increase the hit ratio significantly.

Here we have assumed that, similarly to the current implementation, the entire file is read into memory on a buffer *miss*. The advantage of this is the non-complex implementation of not having to keep track of and replace individual vertices in the buffer. On the other hand, this approach can lead to large data volumes being read and written if often accessed vertices are stored in different files, which leads to much time spent waiting on I/O, as demonstrated by the results. As discussed in Section 2.2.6, better performance can be achieved by replacing individual vertices in the buffer as they are referenced. Using the **LRU** policy on a vertex level, or even the **FIFO** policy would further increase performance based on the current implementation, particularly if combined with the R-Tree when fetching the vertex from disk. This would be an improvement since a single buffer *miss* would not evict all vertices from the buffer when the buffer can possibly contain vertices that will be referenced in the near future. Additionally, it is worth noting that as Dijkstra's algorithm works through the vertices it does so in a sweeping manner, as seen in [18], such that a buffer that tries to keep recently references vertices could work well. Such an implementation would have to take care to not create files of too large size to avoid the performance issues of this thesis, but since an RTree is data-driven it will adapt to the distribution of vertices and create files of approximately equal size.

Does the memory limit work?

As discussed in Section 3.1.2, the implementation introduces a constant adjustment factor when calculating the file index and element index of a given vertex. This adjustment factor essentially makes the size of the type of the element *T* which is stored in `RVector` closer to its true size. The reason why `sizeof(T)`, as seen in Code Listing 3.4 and 3.5, is inaccurate is that the `sizeof`-operator is C++ does not calculate a "deep size", i.e. the collective size of the elements in any member containers is not calculated into the size. As an example described in Section 3.1.2, each vertex has a list of varying sizes of all the edges it is connected to. Thus the actual memory footprint of a vertex is impacted by the number of vertices, as well as other factors. The implementation used an adjustment factor of 2, meaning that the assumption is that the actual memory footprint is twice as large as `sizeof(T)`. As seen in Figure 3.8, the actual memory footprint exceeds the limit by between 13% and 25 %, which implies that the adjustment factor of 2 does not work with the dataset.

This challenge is quite tied to the current implementation since the adjustment factor is needed every time the position of a vertex is to be calculated. On the other hand, the maximum size of the memory buffer would have to be enforced in any implementation, so it would be useful to have better solutions to this problem. One possibility is to set the adjustment factor at the beginning of the query, based on statistics or sampling of the dataset to obtain an adjustment factor proportional to the actual memory footprint of the vertices present for the query. At this time, the current, actual memory usage of the query is not available to `RVector`, but given the availability of this, the adjustment factor could be set quite easily based on data such as the average number of edges for each vertex, as shown in Table 3.2.

As mentioned in Section 3.1.2, the memory limit is set to the minimum of the two system variables `tmp_table_size` and `max_heap_table_size`. This choice was made because this value is available as the variable `ram_limitation` in MySQL server code, and thus using it as a limit would impose a limit limit on the routing module which is equal to the limit imposed on another, existing MySQL module which also has a possibly large memory usage. These system variables are not intended for the routing module, and can consequently be changed without care for it. Additionally, a user may want to increase the memory limit for heap tables and temporary tables, without wanting to increase the memory for the routing module, something that is impossible in the current implementation. On the other hand, it provides a realistic memory limitation since it is already present in MySQL.

4.3.2 Interface

The interface to the routing functionality as currently implemented in MySQL differs from the interface defined in SQL/MM standard, as seen in Code Listing 2.1. The current interface can be seen in Code Listing 3.2. The reason for this is partly the choice to implement the function as an aggregation function, as discussed in Section 3.1.2. This type of function takes rows from a table as input, where the table name is given in the `FROM edge_table_name` of Code Listing 3.2, while SQL/MM defines that the table name should be given as a parameter to the function. Other than this, the interfaces differ mostly because SQL/MM relies on geometric data to define an edge, and as such only takes the geometry (*geometry_column_name*) and the start of the edge(*path_start_column_name*) as arguments. The implementation explicitly takes arguments for the source (*source_id_column_name*) and target id (*target_id_column_name*).

The reason for this is mainly because of the prototype nature of this implementation, and also because the source and target ids are the data that is immediately available in the dataset. Additionally, BGL creates an edge from source and target id, i.e. from the data available, thus using the geometry input from the function would require a layer of indirection. On the other hand, the geometry-information *is* also available in the dataset and it would be possible to implement this interface, given that we didn't implement it as an aggregation function, but the simple solution was chosen in this prototype.

4.3.3 Pruning

The implementation prunes edges that are outside a given radius from the midpoint between the source and the target. This helps reduce the memory footprint of the implementation, by eliminating edges that are most likely irrelevant to the shortest path. As mentioned in Section 3.1.2, the focus of this project is not to efficiently select vertices, and consequently, the radius from the midpoint was set experimentally by the rationale that the shortest path between a source and a target will very likely not travel more than 50% of the distance to the target in the wrong direction.

One example where this fails, which can be classified as an edge-case but nevertheless is interesting, is the following. If you are traveling, as shown in Figure 4.1, from the purple square in Trondheim to the orange square in Vanvikan on the other side of the fjord and cannot use a boat, the implementation will erroneously give the answer that no route exists. This happens because the implementation prunes all edges outside the red circle, which is the defined radius from the midpoint, while we need all edges within the green circle to compute the optimal shortest path. Note that if a ferry/boat is an option, which it usually is, the red circle does contain the shortest path, so this is an edge case. As a result, the implementation loses the optimality of Dijkstra's algorithm due to edge pruning in

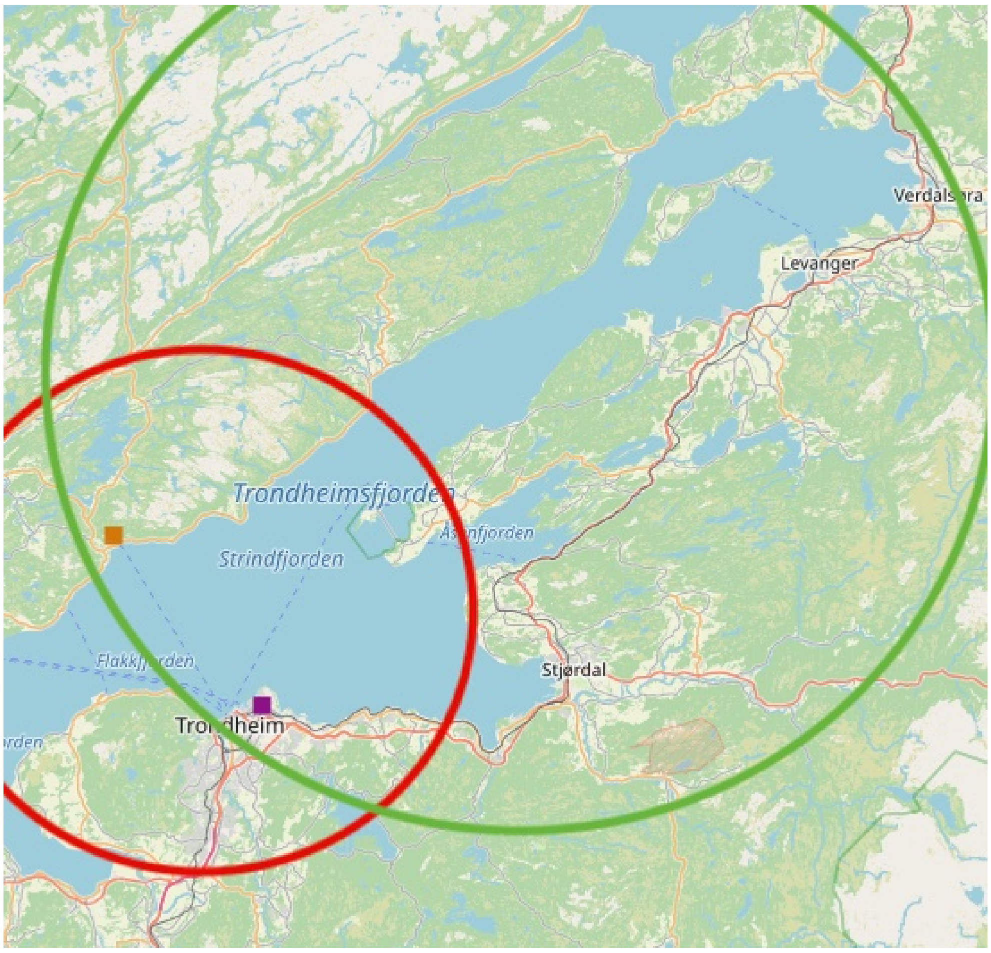


Figure 4.1: Map showing an example of the actual vertex set with the current implementation (red circle) and the required vertex set for a correct shortest path (green circle) when traveling from the purple square to the orange square. © OpenStreetMap contributors

this case. It is not difficult to imagine other similar cases which can lead to similar results. Thus, we have to ask the question if it is necessary to have pruning. In the current implementation, it is necessary, because when the requested vertices are scattered almost randomly across files the I/O volumes quickly become large when files are read and written into memory often. On the other hand, if we had a more efficient buffering strategy, as discussed in Section 4.3.1, which is better at keeping the relevant vertices in memory, the number of vertices stored on disk would not be an issue, at least in the context of this thesis, and with the low cost of storage today it would probably not be an issue in many other applications either. This is especially true if one considers the disk space the cost to guarantee optimality from Dijkstra's algorithm.

4.3.4 Algorithms

This thesis only implements Dijkstra's algorithm. As discussed in Section 2.2.4 a central part of the implementation is its declarative nature, i.e. the possibility to implement other algorithms and let the module choose which algorithm to use based on the context. It would have been interesting to see the performance of the implementation with other algorithms, but given the current performance, it would not have changed the conclusion. With an improved buffering strategy, the current implementation is a useful platform to build on with other algorithms, but the current implementation would not have benefited from more algorithms due to its performance issues.

4.4 Validation

It is worth noting that we have not validated most of the shortest paths calculated here against an external source. At the beginning of the project, we calculated shortest paths in the local environment of Trondheim, where we could validate paths against our local knowledge. The result from this was very promising since the calculated paths matched well with our knowledge. Also since the Boost Graph Library implementation of Dijkstra's algorithm is used, we can be fairly certain that it performs correctly, as discussed in Section 2.3.4. Nevertheless, validation of results would be valuable, especially if the solution was getting ready for production.

Chapter 5

Conclusion

This thesis examines some of the relevant questions when implementing routing in a database system. It looks at how we can create an SQL interface to a routing module that allows the database system to optimize the execution similarly to how it optimizes a regular query and argues that such a routing module should have a declarative interface, which transfers much of the responsibility for choosing algorithms and managing memory usage to the database system. An examination of existing approaches, with **pgRouting** and the **Oracle Routing Engine** can be found in Section 2.3.2 and 2.3.3, and an argument for the declarative interface can be found in Section 2.2.4. A discussion of the interface in the current implementation is found in Section 4.3.2.

The thesis examines two research questions:

RQ1: How can we create an SQL interface to a routing module that allows the database system to optimize the execution similarly to how it optimizes a normal query?

RQ2: What are some of the factors that impact the performance of a system with an interface as in **RQ1** when the system is memory limited?

A routing prototype was implemented in MySQL to answer **RQ1**. It is implemented as an aggregation function in MySQL and uses Boost Graph Library to perform Dijkstra's algorithm, and a custom data structure called `RVector` to handle memory management of vertices in the graph and enforce a memory limit. Details about the implementation are found in Section 3.1.2, and some choices and insights from the evaluation are discussed in Section 4.3. The routing function presents a declarative interface, and Section 2.2.4 and 2.2.5 examines some consequences of this choice, and argues that it is the preferable choice in an SQL database system.

To answer **RQ2**, the implementation was experimentally evaluated, and the results from this evaluation are found in Section 3.2. The implementation works and calculates correct routes while observing and respecting a memory limit. From the evaluation, it is found that the amount of I/O performed by the implementation is an important factor in the performance, as seen in Section 3.2. The amount of I/O performed correlates strongly with the time slowdown of a query. Additionally, for queries where the amount of I/O is lower and the system is not so limited by waiting on I/O, the hit-ratio in the buffer is found to be an important factor. Section 4.3.1 discusses the weaknesses of the current buffering strategy and argues that the hit ratio can be improved while the amount of I/O can be decreased. Additionally, it is found in Section 4.3.3 that pruning of unnecessary edges is an important factor for the performance of the system and that effective pruning is a strategy for reducing the memory footprint of the system, even with the unsophisticated pruning that is performed in the current implementation.

5.1 Future work

In the future, it would be interesting to see how storage cost and performance are impacted when the graph uses directed edges instead of undirected edges, as it does now. This would improve the quality of shortest paths that are calculated, but since the amount of directed edges in the dataset is low it requires some more insight.

The tests in this thesis were only run on one computer. It would be useful to see how it performs on other computers, especially since the point of the prototype is to make any type of computer able to run the routing module.

Right now, the implementation prunes edges based on a simple distance from the midpoint between the source and target. It would be very interesting to see how more intelligent pruning would impact performance, such as taking the mode of transportation (walking, bike, car) into consideration and pruning based on how far it is possible to get with the current mode. Another option would be to identify the average distance shortest paths move in the wrong direction before moving in the right direction as a pre-processing step and using this to prune all unusable edges.

To simplify the implementation, `RVector` was only used for **VertexList** and not for **EdgeList**. A natural extension of this thesis would be to also use `RVector` for **EdgeList**, and thus further control the memory usage of the entire application, and also observe the performance in this case.

It would also be interesting to examine the impact of using geometries to store the edges and to define the source and target vertices when compared to using the vertex ID and edge length as done in this thesis.

Bibliography

[1] N. Ryeng, “Spatial support in mysql,” FOSS4G, 2019, [Online]. Available: https://www.slideshare.net/NorvaldRyeng/spatial-support-in-mysql.

[2] T. Cormen, C. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, Massachusets, US: The MIT Press, 2009, 3rd edition.

[3] G. Dantzig, R. Fulkerson, and S. Johnson, “Solution of a large scale traveling salesman problem,” *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, 1954.

[4] G. Cong, C. S. Jensen, and D. Wu, “Efficient retrieval of the top-k most relevant spatial web objects,” *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 337–348, Aug. 2009, ISSN: 2150-8097. DOI: 10.14778/1687627.1687666. [Online]. Available: https://doi.org/10.14778/1687627.1687666.

[5] D. Delling, P. Sanders, D. Schultes, and D. Wagner, “Engineering route planning algorithms,” in *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 117–139, ISBN: 978-3-642-02094-0. DOI: 10.1007/978-3-642-02094-0_7. [Online]. Available: https://doi.org/10.1007/978-3-642-02094-0_7.

[6] Oracle Corporation. (May 3, 2021). “Spatial data types,” [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/spatial-types.html.

[7] “OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture,” Open Geospatial Consortium Inc., Standard, 2011.

[8] Oracle Corporation. (Jun. 8, 2021). “11.4.2.2 geometry class,” [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/gis-class-geometry.html.

[9] pgRouting Community. (Jun. 8, 2021). “Implementation of node class in osm2pgrouting,” [Online]. Available: https://github.com/pgRouting/osm2pgrouting/blob/main/src/osm_elements/Node.cpp.

[10] pgRouting Community. (Nov. 29, 2020). “Osm2pgrouting - import osm data into pgrouting database,” [Online]. Available: `http://pgrouting.org/docs/tools/osm2pgrouting.html`.

[11] “Information technology - Database languages - SQL multimedia and application packages,” International Organization for Standardization, Geneva, CH, Standard, 2015.

[12] L. L. Larmore. (Dec. 3, 2020). “Pathing,” [Online]. Available: `http://web.cs.unlv.edu/larmore/Courses/CSC269/pathing`.

[13] R. Gutman, “Reach-based routing: A new approach to shortest path algorithms optimized for road networks.,” Jan. 2004, pp. 100–111.

[14] A. Goldberg, H. Kaplan, and R. Werneck, “Reach for a*: Efficient point-to-point shortest path algorithms,” Tech. Rep. MSR-TR-2005-132, Jan. 2006, Technical Report for CLASSiC FP7 European project, p. 41. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/reach-for-a-efficient-point-to-point-shortest-path-algorithms/`.

[15] A. Goldberg and C. Harrelson, “Computing the shortest path: A search meets graph theory,” eng, ser. SODA ’05, Society for Industrial and Applied Mathematics, 2005, ISBN: 9780898715859.

[16] D. Delling, T. Pajor, and R. F. Werneck, “Round-based public transit routing,” *Transportation science*, vol. 49, no. 3, pp. 591–604, 2015.

[17] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.

[18] Computer Science. (Dec. 8, 2020). “Graph data structure 4. dijkstra’s shortest path algorithm,” [Online]. Available: `https://www.youtube.com/watch?v=pVfj6mxhdMw`.

[19] D. Wagner and T. Willhalm, “Speed-up techniques for shortest-path computations,” in *STACS 2007*, W. Thomas and P. Weil, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 23–36, ISBN: 978-3-540-70918-3.

[20] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: `10.1109/TSSC.1968.300136`.

[21] K. Wang. (Dec. 7, 2020). “Compare a* with dijkstra algorithm,” [Online]. Available: `https://www.youtube.com/watch?v=g024lzsknDo`.

[22] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang, “Relational approach for shortest path discovery over large graphs,” *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 358–369, Dec. 2011, ISSN: 2150-8097. DOI: `10.14778/2095686.2095694`. [Online]. Available: `https://doi.org/10.14778/2095686.2095694`.

[23] D. Delling, P. Sanders, D. Schultes, and D. Wagner, “Engineering route planning algorithms,” in *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 117–139, ISBN: 978-3-642-02094-0. DOI: 10.1007/978-3-642-02094-0_7. [Online]. Available: https://doi.org/10.1007/978-3-642-02094-0_7.

[24] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, “Shortest paths algorithms: Theory and experimental evaluation,” eng, *Mathematical programming*, vol. 73, no. 2, pp. 129–174, 1996, ISSN: 0025-5610.

[25] R. Elmasri, *Fundamentals of database systems*, eng. Boston , Mass: Pearson, 2016, 7th edition, ISBN: 1-292-09761-2.

[26] W. Stallings, *Operating systems : Internals and design principles*, eng, Harlow, England, 2018.

[27] P. Rigaux, M. Scholl, and A. Voisard, “6 - spatial access methods,” eng, in *Spatial Databases*, Elsevier Inc, 2002, pp. 201–266, ISBN: 1558605886.

[28] H. Strandlie, “Geospatial routing in dbmss,” Autumn Project -NTNU, 2020.

[29] Oracle Corporation. (Jun. 8, 2021). “The opengis geometry model,” [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/opengis-geometry-model.html.

[30] Oracle Corporation. (Jun. 8, 2021). “Spatial analysis functions,” [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/spatial-analysis-functions.html.

[31] Oracle Corporation. (May 3, 2021). “Changes in mysql 8.0.23 (2021-01-18, general availability), spatial data support,” [Online]. Available: https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-23.html#mysqld-8-0-23-spatial-support.

[32] Oracle Corporation. (May 3, 2021). “Changes in mysql 8.0.24 (2021-04-20, general availability), spatial data support,” [Online]. Available: https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-24.html#mysqld-8-0-24-spatial-support.

[33] R. R. Sankepally and K. S. Rajan, “Improving path query performance in pgrouting using a map generalization approach,” *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLII-4/W8, pp. 191–198, 2018. DOI: 10.5194/isprs-archives-XLII-4-W8-191-2018. [Online]. Available: https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLII-4-W8/191/2018/.

[34] pgRouting Contributors. (Apr. 19, 2021). “Pgrouting introduction,” [Online]. Available: https://docs.pgrouting.org/latest/en/pgRouting-introduction.html.

[35] R. Kothuri, A. Godfrind, and E. Beinat, *Pro Oracle Spatial for Oracle Database 11g*. New York, US: Springer-Verlag New York, Inc, 2007.

[36] J. Siek, L. Lee, and A. Lumsdaine, *The Boost Graph Library*. Upper Saddle River, New Jersey, US: Pearson Education, Inc., 2002.

[37] J. Siek. (Apr. 26, 2021). "Choosing the edgelist and vertexlist," [Online]. Available: `https://www.boost.org/doc/libs/1_73_0/libs/graph/doc/using_adjacency_list.html#sec:choosing-graph-type`.

[38] OpenStreetMap Foundation. (Nov. 21, 2020). "About openstreetmap," [Online]. Available: `https://www.openstreetmap.org/about`.

[39] Open Knowledge Foundation. (Nov. 21, 2020). "Open data commons open database licence (odbl," [Online]. Available: `https://opendatacommons.org/licenses/odbl/`.

[40] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '94, Arlington, Virginia, USA: Society for Industrial and Applied Mathematics, 1994, pp. 516–525, ISBN: 0898713293.

[41] cplusplus.com. (May 24, 2021). "Containers," [Online]. Available: `https://www.cplusplus.com/reference/stl/`.

[42] Oracle Corporation. (May 31, 2021). "Mysql dynamic system variables," [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/dynamic-system-variables.html`.

[43] D. Gregor. (May 28, 2021). "Boost graph library - bundled properties," [Online]. Available: `https://www.boost.org/doc/libs/1_42_0/libs/graph/doc/bundles.html`.

[44] OpenStreetMap Foundation. (Nov. 29, 2020). "Export - openstreetmap," [Online]. Available: `https://www.openstreetmap.org/export`.

[45] Overpass Community. (Nov. 30, 2020). "Overpass api," [Online]. Available: `https://www.overpass-api.de`.

[46] Geofabrik GmbH. (Nov. 29, 2020). "Download openstreetmap data for norway," [Online]. Available: `https://download.geofabrik.de/europe/norway.html`.

[47] Oracle Corporation. (Sep. 11, 2020). "11.4.5 spatial reference system support," [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/spatial-reference-systems.html`.

[48] wiki.GIS.com Community. (Nov. 30, 2020). "Well-known text," [Online]. Available: `http://wiki.gis.com/wiki/index.php/Well-known_text`.

[49] The PostgreSQL Global Development Group. (Nov. 30, 2020). "Pg_dump," [Online]. Available: `https://www.postgresql.org/docs/9.3/app-pgdump.html`.

[50] R. Ramey. (Jun. 4, 2021). “Serialization - overview,” [Online]. Available: `https://www.boost.org/doc/libs/1_76_0/libs/serialization/doc/index.html`.

[51] R. Ramey. (Jun. 4, 2021), [Online]. Available: `https://www.boost.org/doc/libs/1_76_0/libs/serialization/doc/tutorial.html#serializablemembers`.

[52] R. Ramey. (Jun. 4, 2021), [Online]. Available: `https://www.boost.org/doc/libs/1_76_0/libs/serialization/doc/tutorial.html#nonintrusiveversion`.

[53] Oracle Corporation. (Dec. 1, 2020). “Memory summary tables,” [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/memory-summary-tables.html`.

[54] N. Ryeng. (Apr. 17, 2021). “Query profiling in mysql,” [Online]. Available: `https://github.com/nryeng/dbama-query-profiling`.

Appendix A

Additional Details

This appendix contains some more details that are informative, but not necessary to reach the conclusion of the thesis. This includes details about the dataset, how the dataset was imported into MySQL, as well as code-level details about the implementation and the evaluation.

A.1 Dataset export

To export data from OpenStreetMap, three different strategies were needed. The **small** dataset was exported using the export functionality[44] found on the OSM website. The **medium** dataset was exported with the Overpass API[45]. The export functionality used for the **small** dataset has a vertex limit of a maximum of 50 000 vertices in a single export, and we also experienced error messages informing of exports hitting memory limits in the Overpass API used for the **medium**, so the **large** dataset was downloaded from Geofabrik[46], and not any of the former sources.

The **small** and **medium** XML files used can be found at
https://drive.google.com/drive/folders/1Rfn33w07TDt5Fw4Kpgjy4T1GDcDUh6b3.

The **large** dataset used can be downloaded from [46]. It is tagged with the date *2020-11-26 03:31*. The **small** and **medium** datasets were exported in October 2020 and may have changed if they are downloaded from the sources and not the link given above.

A.2 Dataset handling

The next step, after exporting the data, was to load it into a pgRouting-enabled PostgreSQL database with the help of `osm2pgrouting`[10]. This import results in several tables, of which we use two: *ways* and *ways_vertices_pgr*. The first contains data about all edges found in the graph, such as the source and target of each edge, the coordinates for the source and target, and the costs of traveling the edge in each direction. The definition of the *ways* table can be seen in Table A.1. Each source and target of edges in the *ways* table is a vertex, and these are described in the *ways_vertices_pgr* table, and the definition is found in Table A.2. *ways_vertices_pgr* also contains additional information about vertices, such as the coordinates of each vertex, and in total, this is enough to build a graph.

A column named *the_geom* of type *GEOMETRY* is present is both *ways* and *ways-_vertices_pgr* which describes the edges and points of the tables when they are placed in a Spatial Reference System[47]. Migrating this column to MySQL from PostgreSQL was not straightforward, but it was solved by creating a new column which had the type *TEXT* and exporting the geometry to Well-known text[48]. Well-known text is a markup language for representing geometries in a portable way in GIS, which is supported by PostgreSQL and MySQL, and it can be used to export and import between them. This operation can be seen in Code Listing A.1.

Code listing A.1: Example of export to Well Known Text

```
ALTER TABLE ways ADD COLUMN the_geom_line TEXT;
UPDATE ways SET the_geom_line = ST_AsText(the_geom);
ALTER TABLE ways DROP COLUMN the_geom;
```

Next, we need to dump the contents of the pgRouting database in PostgreSQL into `INSERT` statements. Since these are standardized, they can be executed in MySQL and used to load the dataset into MySQL. The pg_dump tool[49] performs this job and dumps an entire PostgreSQL database into `INSERT` statements, but some manual post-processing is necessary to adapt the syntax to MySQL.

At last, the dumped and post-processed data is run in MySQL, and the corresponding *the_geom* column is created and filled with data converted from the column containing the Well-known text. See an example in Code Listing A.2.

Code listing A.2: Example of operations when importing data to MySQL

```
ALTER TABLE ways ADD COLUMN the_geom GEOMETRY;
UPDATE ways
    SET the_geom = ST_GeomFromText(the_geom_line, 4326, "axis-order=long-lat");
ALTER TABLE ways DROP COLUMN the_geom_line;
```

A.3 Implementation details

The code for the implementation can be found at https://github.com/strandlie/mysql-server/tree/routing on the branch *routing*. The code which is discussed below can be found in the commit with the hash: $aa894e0ec34e362bd3f309a98b45f554a3de7fee$.

A.3.1 Aggregation function and routing

As mentioned in Section 3.1.2 the prototype is implemented as an aggregation function. This was done by changing the `mysql-server/sql/item_sum.h/cc` files and defining a custom aggregation function which we call `Item_sum_route`. In MySQL, aggregation functions are C++ classes with member variables and functions which are called by the server to execute the function and calculate the result. There are several such functions, but the two main functions for an aggregation function are:

`ClassName::add()` Called when a new row from the table is ready to be processed.

`ClassName::val_<type_of_data>()` Called after processing all rows to produce an aggregated result. <type_of_data> can be be one of `str`, `decimal`, `int`, and `real`. The `val_str()`-method was implemented here, since a route cannot be easily represented by a numeric value.

`Item_sum_route::add()` is called for each edge in the dataset. The current implementation does some preliminary pruning at this point. This pruning is very limited in scope because the coordinates of the source and target vertex are not included as parameters to the `ROUTE` function, and no easy way was found to fetch information from other tables in the database from code in an aggregation function. Consequently, we do not know what the coordinates for the source and target vertex are, and cannot calculate a midpoint, until edges with the source and target vertex at either end have been processed by the `add()` function. This means, that until we have processed such edges, *all* edges must be added to the graph and the amount of pruning is dependent on when the source and target vertices appear in edges in the graph. Initially, we used only this strategy for pruning, which routinely pruned only 5-15% of edges and not every edge outside the radius.

`Item_sum_route::val_str()` aggregates the result by constructing an instance of the `Graph_routing`, and getting the id of the source and target which is supplied to the `ROUTE` function. All logic related to routing, together with Boost Graph Library calls are contained within the `Graph_routing` class. `Graph_routing` also makes another pass through the edges that were added in the add()-function. At this point, the coordinates for the source and target vertex are known as long as they exist as vertices in the dataset, and the midpoint can be calculated. All edges outside the defined radius from the midpoint are pruned, and we are left with all edges which have at least one point within the radius. This pruning strategy is much more effective, and it has been observed to prune up to 93% of edges.

The `Graph_routing` class encapsulates all calls to Boost Graph Library, and the call to Dijkstra's algorithm is shown in Code Listing A.3. It can be found on line 93 in `mysql-server/sql/routing/graph_routing.cc`. It simply takes the graph, source vertex, an array to store distances in (to look at when the algorithm completes, this is not used in the current implementation), and an array to store predecessors in (to look at when the algorithm completes, used to output the shortest path in the current implementation). From this, it is apparent that the implementation stores some state about what source the routing was last performed for, and stores this state until it is output at the end. It is also clear that implementing more algorithms is straightforward since an equivalent function to `executeDijkstra-(Vertex source)` can be added for other algorithms.

Code listing A.3: How the call to Dijkstra's algorithm in BGL happens

```
void Graph_router::executeDijkstra(Vertex source) {
    dijkstra_shortest_paths(
      G, source,
      b::distance_map(&distances[0]).predecessor_map(&predecessors[0]));
    currentSource = source;
}
```

At the end of `Item_sum_route::val_str()`, a result string is created from the predecessors found in the shortest path, and it is returned.

A.3.2 Serialization

The main challenge when developing this prototype was understanding and correctly implementing serialization for graph vertices in Boost. Initially, a naive approach was taken where the « operator was used directly to write out data objects to a file, but this turned out to not work for the more complex and nested vertex data type in Boost Graph Library. For this reason, Boost Serialization[50] was used. Initially, the expectation was that support for Boost Serialization should be built into Boost Graph Library, which would help implement serialization without many bugs and much work. Unfortunately, this is not the case and we had to implement serialization manually for every relevant data type.

It is not difficult to implement serialization for Boost Serialization. Mainly it can be done in one of two possible ways:

- **1:** The class of the object has defined a `serialize`-method, which describes how serialization is performed for such objects[51]
- **2:** The class of the object does *not* define a `serialize`-method, and we have to use non-intrusive serialization[52]

This thesis used alternative **2**, since we did not want to change Boost Graph Library. It is simply a function named `serialize`, as seen in Code Listing A.4, which takes an archive (can be thought off as a file reference), an object and a version number, and performs both read from and write to file using the operator in C++. The disadvantage to this approach is that it does not have access to private or protected members in the data-object seen in Code Listing A.4. This created a bug with the implementation, which finally had to be fixed by changing the visibility of a variable in the class `stored_ra_edge_iter` in `boost/graph/detail/adjacency-`

_list.hpp from protected to public. In other words, we had to change the code in an imported library, which is not ideal, but it is documented in the README of the mysql-server/sql/routing directory, and if native support for serialization is implemented in BGL, as proposed in this thesis, it would not be necessary.

Code listing A.4: Example of non-intrusive serialization

```
template <class Archive>
void serialize(Archive &ar, stored_vertex_t &data,
               __attribute__((unused)) const unsigned int version) {
  ar &data.m_out_edges;
  ar &data.m_property;
}
```

A.3.3 Parser

To make the MySQL Server call our function, the word ROUTE was added to the parser grammar, and to achieve this we changed sql/yacc.yy, which contains parser rules which assign expressions and their arguments to different, underlying functions. This can be found on lines 10485 to 10488 in mysql-server/sql/sql-_yacc.yy, and the important lines are seen in Listing A.5. ROUTE_SYM is defined in sql/lex.h as a SYM_FN and at line 1267 in sql/yacc.yy similar to existing code there. expr_list is a rule that quickly allowed us to get the parser to recognize our query, and it defines a function that takes any list of arguments. One current disadvantage of this is that the type of these arguments are not checked, which would be necessary future improvement before releasing into production.

Code listing A.5: The new lines added to the parser in yacc.yy

```
ROUTE_SYM '(' expr_list ')' opt_windowing_clause
{
    $$= NEW_PTN Item_sum_route(@$, $3, $5);
}
```

A.4 Test details

Each test run with each of the vertex pairs found by the query in Code Listing 3.6 followed a specific procedure, to make sure the results were as comparable as possible. First, only one test was run at a time. Then, for each test the following was performed:

1: Restart the MySQL server if it was previously running

2: Set the memory limit by setting the max_heap_table_size system variable

3: Start the monitoring procedure in Appendix A.5

4: Execute the query

5: Wait for the query to finish. Unfortunately, this usually takes hours or days

6: Export the data from the test into CSV files with the procedure in Code Listing A.11

The test uses three mechanisms for measuring data, which are found in the sections below.

A.4.1 Measuring memory

To measure memory, the implementation uses a custom allocator called `Routing_allocator` which gives `std::vectors` an unique key in the `performance_schema.memory_summary_by_thread_by_event_name` table[53] when allocated when the program is run. This means that MySQL keeps track of every byte that is allocated and freed, and also maximum and minimum amounts, in that vector. In the implementation we find this in `mysqlserver/sql/routing/rvector.h` on line 36, seen in Code Listing A.6.

The definition of `Routing_allocator` is shown in Code Listing A.7, which shows that the `Routing_allocator` is just a wrapper around `Malloc_allocator` in MySQL, and initializes it with `key_memory_routing`, which is a custom key equal to `"memory/sql/routing"`.

Code listing A.6: Definition of a std::vector with RoutingAllocator

```
  std::vector<T, Routing_allocator<T>> vec_;
```

Code listing A.7: Definition of Routing Allocator

```

template <typename T>
class Routing_allocator: public Malloc_allocator<T> {
public:
  Routing_allocator(): Malloc_allocator<T>(key_memory_routing) {}
};
```

To capture this memory and store it during a run of the query, the memory usage is stored with the procedure in Code Listing A.10, and exported into CSV with the procedure in Code Listing A.11.

A.4.2 Measuring time

Time consumption of the query was recorded by the internal time recorder in MySQL. An example of this time consumption is shown in Figure 3.3. No additional custom code was implemented to measure time consumption.

A.4.3 Measuring buffer performance, and I/O

To measure the number of hits and misses in the buffer, as well as the number of I/O bytes read and written, the implementation has some simple counters. This can be seen in `mysqlserver/sql/routing` on line 365, which is also shown in Code Listing A.8. Here the lines `RoutingStats::numBufferHits += 1` and `RoutingStats::numSwaps += 1` count the occurrences of a buffer hit or miss, respectively. These numbers are stored as static member variables of the `RoutingStats` class and are reported at the end of the query.

Additionally, since we are manually handling file objects (the `getInfileForFileNr(i)` and `getOutfileForFileNr(i)` just returns an open file object, or creates it if it does not exist) I/O is measured using the `tellg()` and `tellp()` methods of `std::ifstream` and `std::ofstream`. This can be seen in Code Listing A.9, and works because we are always replacing the entire buffer, which means that we can always start reading and writing from the beginning of the respective file. By calling `file.seekg(0)` and `fileseekp(0)` we are resetting reading and writing to the start of each file, and the amount of bytes written / read can be directly read from the `tellg()/tellp()` methods.

Code listing A.8: The definition of the method changeWorkingSet in RVector

```
void changeWorkingSet(size_t new_idx) {
    if (new_idx == currentFileIdxInMem) {
      RoutingStats::numBufferHits += 1;
      return;
    }
    RoutingStats::numSwaps += 1;
    fh.pushVectorWithIdx(currentFileIdxInMem, vec_);
    vec_ = std::vector<T, Routing_allocator<T>>();
    vec_ = fh.readVectorWithNumber(new_idx);
    currentFileIdxInMem = new_idx;
}
```

Code listing A.9: Methods for reading and writing to file

```
template <typename T>
void routing_file_handler<T>::readVectorWithNumber(size_t file_nr,
                                                   std::vector<
                                                     T,
                                                     Routing_allocator<T>
                                                   >& vec) {
  std::ifstream &file = getInfileForFileNr(file_nr);
  file.seekg(0);
  if(file.is_open()) {
    boost::archive::text_iarchive ia(file);
    ia >> vec;
  }
  RoutingStats::numBytesRead += file.tellg();
}

template <typename T>
void routing_file_handler<T>::pushVectorWithIdx(
    size_t file_nr, std::vector<T, Routing_allocator<T>> vec) {
```

```
  std::ofstream &file = getOutfileForFileNr(file_nr);
  file.seekp(0);
  boost::archive::text_oarchive oa(file);
  oa << vec;
  RoutingStats::numBytesWritten += file.tellp();
  file.flush();
}
```

A.5 Query Profiling

Below, the SQL procedure which is used to collect information about the memory usage of a query in MySQL can be found. It is adapted from [54] and useful documentation about usage is found in the README. It is worth noting that this procedure samples the memory usage at given intervals, which is apparent from the `DO SLEEP(0.1);` towards the middle of the procedure. This means that the column which shows the current memory usage can miss spikes and valleys in memory usage. Fortunately, this is not an issue because `performance_schema-.memory_summary_by_thread_by_event_name` also stores the maximum amount of allocated memory at any point, even if this happens "between" sampling intervals. For the results, we rely on this `HIGH_NUMBER_OF_BYTES_USED` amount when measuring memory usage.

Code listing A.10: Procedure to collection information about the memory usage of a query

```
DROP PROCEDURE IF EXISTS monitor_connection;
DELIMITER $$
CREATE PROCEDURE monitor_connection(
  IN conn_id BIGINT UNSIGNED,
  IN sleep_interval FLOAT
)
BEGIN
  DECLARE thd_id BIGINT UNSIGNED; -- P_S thread ID of connection conn_id
  DECLARE state VARCHAR(16); -- Current connection state
  DECLARE stage_ignore_before BIGINT UNSIGNED; -- Ignore history before this TS
  DECLARE stage_min_ts BIGINT UNSIGNED; -- Earliest timestamp in stage history

  -- Thread ID is used as key in P_S,
  -- so get the thread ID of the suppliced connection ID.
  SET thd_id = (SELECT THREAD_ID
                FROM performance_schema.threads
                WHERE PROCESSLIST_ID=conn_id);

  -- Get the current maximum stage history table timestamp.
  -- All new events will have a more recent timestamp.
  SET stage_ignore_before = (SELECT MAX(TIMER_END)
                             FROM performance_schema.events_stages_history_long
                             WHERE THREAD_ID=thd_id);

  # Create a table to log memory usage data.
  DROP TABLE IF EXISTS monitoring_data;
  CREATE TABLE monitoring_data AS
    SELECT
    NOW(6) AS 'TS',
    THREAD_ID,
    EVENT_NAME,
    COUNT_ALLOC,
    COUNT_FREE,
    SUM_NUMBER_OF_BYTES_ALLOC,
    SUM_NUMBER_OF_BYTES_FREE,
    LOW_COUNT_USED,
    CURRENT_COUNT_USED,
    HIGH_COUNT_USED,
    LOW_NUMBER_OF_BYTES_USED,
    CURRENT_NUMBER_OF_BYTES_USED,
    HIGH_NUMBER_OF_BYTES_USED
  FROM performance_schema.memory_summary_by_thread_by_event_name
  WHERE THREAD_ID = thd_id AND EVENT_NAME = 'memory/sql/routing';

SELECT CONCAT ('Waiting␣for␣connection␣', conn_id, '␣(thread␣', thd_id,
              ')␣to␣start␣executing␣a␣query') AS 'Status';

# Wait for query to start.
REPEAT
  SET state = (SELECT PROCESSLIST_COMMAND
               FROM performance_schema.threads
               WHERE THREAD_ID=thd_id);
UNTIL state = 'Query' END REPEAT;

SELECT 'Connection␣monitoring␣starting' AS 'Status';

# Repeat until query finishes.
REPEAT
  SET state = (SELECT PROCESSLIST_COMMAND
               FROM performance_schema.threads
               WHERE THREAD_ID=thd_id);
  INSERT INTO monitoring_data
    SELECT
      NOW(6) AS 'TS',
      THREAD_ID,
      EVENT_NAME,
      COUNT_ALLOC,
      COUNT_FREE,
      SUM_NUMBER_OF_BYTES_ALLOC,
      SUM_NUMBER_OF_BYTES_FREE,
      LOW_COUNT_USED,
      CURRENT_COUNT_USED,
      HIGH_COUNT_USED,
      LOW_NUMBER_OF_BYTES_USED,
      CURRENT_NUMBER_OF_BYTES_USED,
      HIGH_NUMBER_OF_BYTES_USED
    FROM performance_schema.memory_summary_by_thread_by_event_name
    WHERE THREAD_ID = thd_id AND EVENT_NAME = 'memory/sql/routing'
      ;
  DO SLEEP(sleep_interval);
UNTIL state = 'Sleep' END REPEAT;

SELECT 'Connection␣monitoring␣ended' AS 'Status';

# Get the minimum timestamp in query stage history.

SET stage_min_ts = (SELECT MIN(timer_start)
                    FROM performance_schema.events_stages_history_long
```

```
                      WHERE THREAD_ID=thd_id
                        AND timer_start > stage_ignore_before);
   SELECT stage_min_ts AS 'stage_min_ts',
          stage_ignore_before AS 'stage_ignore_before';
   # Timestamps in picoseconds, divide by 10^12.
   SELECT
     EVENT_NAME,
     SOURCE,
     timer_start,
     timer_end,
     (timer_start - stage_min_ts) / 1000000000000 AS start,
     (timer_end - stage_min_ts) / 1000000000000 AS end
     FROM performance_schema.events_stages_history_long
     WHERE THREAD_ID = thd_id AND timer_start > stage_ignore_before
     ORDER BY timer_start;
   # Dump it to a table, too:
   DROP TABLE IF EXISTS monitoring_stages;
   CREATE TABLE monitoring_stages AS
     SELECT
       EVENT_NAME,
       SOURCE,
       timer_start,
       timer_end,
       (timer_start - stage_min_ts) / 1000000000000 AS start,
       (timer_end - stage_min_ts) / 1000000000000 AS end
       FROM performance_schema.events_stages_history_long
       WHERE THREAD_ID = thd_id AND timer_start > stage_ignore_before
       ORDER BY timer_start
   ;
END $$
DELIMITER ;
```

Code listing A.11: Procedure to export information about the memory usage of a query into CSV

```
    SET @min_ts = (SELECT UNIX_TIMESTAMP(MIN(TS)) FROM monitoring_data);
    (SELECT 'wall_clock',
            'relative␣time',
            'number_alloced_mb',
            'number_free_mb',
            'current_used_mb',
            'high_used_mb')
    UNION
    (SELECT
       TS AS the_ts,
       UNIX_TIMESTAMP(TS) - @min_ts,
       SUM_NUMBER_OF_BYTES_ALLOC / 1000 / 1000 AS SUM_ALLOC_MB,
       SUM_NUMBER_OF_BYTES_FREE / 1000 / 1000 AS SUM_FREE_MB,
       CURRENT_NUMBER_OF_BYTES_USED / 1000 / 1000 AS CURRENT_USED_MB,
       HIGH_NUMBER_OF_BYTES_USED / 1000 / 1000 AS HIGH_USED_MB

    FROM monitoring_data
    WHERE EVENT_NAME='memory/sql/routing')
    INTO OUTFILE '~/tmp/prosjektoppgave/out.csv'
    FIELDS TERMINATED BY ','
    OPTIONALLY ENCLOSED BY '"'
    ;
```

A.6 Raw vertex IDs

We selected 5 vertex pairs randomly from each dataset, which totals to 15 vertex pairs. For each of these vertex pairs, the test was run 4 times, while gradually lowering the memory limit. In the thesis these vertex pairs were called Pair 1, Pair 2 etc. within each dataset. For reproducability, the global IDs from OSM for these 15 vertex pairs are as following:

Small dataset

Pair 1: *Source:* 8092361144, *target:* 653883113

Pair 2: *Source:* 258602783, *target:* 92273967

Pair 3: *Source:* 4835581659, *target:* 2467059117

Pair 4: *Source:* 1410398764, *target:* 4371567909

Pair 5: *Source:* 5218806429, *target:* 4427833373

Medium dataset

Pair 1: *Source:* 685711897, *target:* 7289575479

Pair 2: *Source:* 5502941627, *target:* 5763744238

Pair 3: *Source:* 7975080722, *target:* 837762674

Pair 4: *Source:* 6955626827, *target:* 5765354460

Pair 5: *Source:* 244137822, *target:* 2428432088

Large dataset

Pair 1: *Source:* 3252828955, *target:* 3134795820

Pair 2: *Source:* 5920192410, *target:* 2134589062

Pair 3: *Source:* 7310206959, *target:* 477237402

Pair 4: *Source:* 7154118225 , *target:* 6982207831

Pair 5: *Source:* 3409028252, *target:* 1957304984

A.7 Tables in pgRouting

This section shows the definitions of the tables produced by *osm2pgrouting* for use in pgRouting.

Table A.1: Definition for table *ways* generated by osm2pgrouting

Column name	Type	Nullable
gid	BIGINT	NOT NULL
osm_id	BIGINT	
tag_id	INTEGER	
length	DOUBLE PRECISION	
length_m	DOUBLE PRECISION	
name	TEXT	
source	BIGINT	
target	BIGINT	
source_osm	BIGINT	
target_osm	BIGINT	
cost	DOUBLE PRECISION	
reverse_cost	DOUBLE PRECISION	
cost_s	DOUBLE PRECISION	
reverse_cost_s	DOUBLE PRECISION	
rule	TEXT	
one_way	INTEGER	
oneway	TEXT	
x1	DOUBLE PRECISION	
y1	DOUBLE PRECISION	
x2	DOUBLE PRECISION	
y2	*DOUBLE PRECISION*	
maxspeed_forward	DOUBLE PRECISION	
maxspeed_backward	DOUBLE PRECISION	
priority	DOUBLE PRECISION	
the_geom	GEOMETRY(LineString, 4326)	

Table A.2: Definition for table *ways_vertices_pgr* generated by osm2pgrouting

Column name	Type	Nullable
id	BIGINT	NOT NULL
osm_id	BIGINT	
eout	INTEGER	
lon	NUMERIC(11, 8)	
lat	NUMERIC(11,8)	
cnt	INTEGER	
chk	INTEGER	
ein	INTEGER	
the_geom	GEOMETRY(Point, 4326)	