Master’s Thesis Nr. 135

Systems Group, Department of Computer Science, ETH Zurich

Distributed Graph Querying with Decoupled Storage and Smart Routing

by

Gustavo Segovia

Supervised by

Dr. Arijit Khan
Prof. Dr. Donald Kossmann

February 2015 – August 2015
Abstract

During this project we developed an online distributed graph query processing system that is capable of handling queries on very large graphs that do not fit on the main memory of a commodity server. A common approach for tackling this problem is to split the graph up into several partitions and assign each of them to a server which then has the dual responsibility of storing part of the graph and processing queries. This design leads to problems like servers being overloaded by workloads that concentrate in one area of the graph and a large amount of inter-machine communication if many queries are at the border of graph partitions. To solve these problems, parts of the graph can be replicated and the partitions can be updated to adapt to the workload however this relies expensive operations like online workload monitoring and copying large amounts of data between machines.

Our system avoids these problems by decoupling the graph storage and query processing. With this design, partitioning of the graph is not relevant since it is now encapsulated by the storage tier and queries no longer have specific servers they must be sent to, facilitating load balancing. However, query processing servers now need to retrieve graph data from storage servers which implies an extra communication cost. This cost is reduced by adding cache to the query processors that holds recently used graph data. In addition, smart routing schemes were developed to route queries to processors that are likely to have relevant data in their cache. We show that, not only our smart routing is able to significantly reduce communication overhead further but also that our system performs well, being able to outperform a state-of-the-art graph querying system.
Contents

Abstract

1 Introduction

2 Background and Related Work
   2.1 k-hop neighborhood search
   2.2 Current graph query processing systems

3 System Design
   3.1 General design
   3.2 Graph Storage Tier
   3.3 Query Processing Tier
   3.4 Router
   3.5 Communication

4 Routing Schemes
   4.1 Objectives
   4.2 Next Ready Routing
   4.3 Hash Routing
   4.4 Landmark Routing
      4.4.1 Preprocessing
      4.4.2 Routing
   4.5 Embed Routing
      4.5.1 Preprocessing
      4.5.2 Routing

5 Experimental Results
   5.1 Setup
   5.2 Workloads
      5.2.1 Locality based
      5.2.2 Uniformly selected
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.3</td>
<td>Concentrated</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Experiments</td>
<td>29</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Locality based workloads with 3-NS queries</td>
<td>29</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Locality based workload with 1-NS, 2-NS and 4-NS queries</td>
<td>31</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Concentrated workloads with 3-NS queries</td>
<td>33</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Uniform workload with 3-NS queries</td>
<td>35</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Varying the cache capacity</td>
<td>37</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Cold cache / warm cache</td>
<td>39</td>
</tr>
<tr>
<td>5.3.7</td>
<td>Preprocessing parameters</td>
<td>41</td>
</tr>
<tr>
<td>5.3.8</td>
<td>Runtime parameters</td>
<td>44</td>
</tr>
<tr>
<td>5.3.9</td>
<td>Performance on other graphs</td>
<td>46</td>
</tr>
<tr>
<td>5.3.10</td>
<td>Scaling out the processing/storage tiers</td>
<td>50</td>
</tr>
<tr>
<td>5.3.11</td>
<td>Comparison with other system</td>
<td>53</td>
</tr>
<tr>
<td>5.3.12</td>
<td>Robustness to graph modifications</td>
<td>56</td>
</tr>
</tbody>
</table>

6 Conclusions

Bibliography
Chapter 1

Introduction

The amount of space needed to store data from large graphs can easily exceed what a commodity server can hold in its main memory. Graphs that represent a social network or part of the World Wide Web, for example, commonly have millions of vertices and billions of edges, with a data footprint reaching the order of terabytes. To develop a system that is capable of answering queries on these graphs with low response times, the data must be split up across several servers so that the whole graph can be kept in the cluster’s aggregate memory.

Graph query processing systems can be designed to handle queries in two different manners. The first, offline query processing, is better prepared to handle queries that typically need to access every vertex and edge in the graph. This is useful for finding vertices’ page rank, partitioning a graph or running other analyses. In this project, we focus on designing and building a system that performs online query processing which, in turn, is well-suited to handle queries that require processing only a section of the graph data. Examples of such include $k$-hop neighborhood search and shortest path queries. It is expected, in general, that an online graph query processing system is able to handle queries very quickly and in parallel while this may not be the case for an offline system.

Commonly, servers in online graph query processing systems act as both query processor and graph storage. This is accomplished by partitioning the graph and assigning each part to a different server. When a query arrives to the system, it is routed to the server that contains its starting point. A problem with this
approach is that there might be a hotspot in the workload and many queries may end up at the same server while others remain idle, thus reducing throughput and efficiency of the system. Another problem is, if a workload contains many queries that act near the boundary of a graph partition, it is likely that the processing server would need to communicate with other servers to retrieve parts of the graph it needs to answer the query. This leads to increased network usage and response times. While solutions have been proposed to these problems like repartitioning and graph replication according to the workload, those solutions are expensive and penalizes the system’s performance.

In contrast to other systems, ours makes a different fundamental design decision; query processing and graph storage are decoupled into two distinct tiers, illustrated in the right side of figure 1.1. The servers allocated to system storage hold all the graph data in main memory and therefore encapsulate the partitioning done on the graph. This means that each query processor is no longer assigned to a fixed part of the graph and is consequently equally capable of handling any request, facilitating workload balancing. Another benefit due to the design is that each tier can be scaled independently; if a certain workload is particularly processing intensive, more servers could be allocated to the processing tier to improve system output. On the other hand, if this design is trivially implemented, every time a query processor needs to retrieve graph data, it must access the network, incurring a communication cost. To reduce this cost, cache has been added to each query processor, allowing it to retain and reuse some graph data to take advantage of the fact that a certain region of the graph might be frequently accessed.

![Figure 1.1: Typical system design compared to ours](image)

If the query processors had no cache, they would be equally efficient to process any given query since they would always need to access the same storage interface
for graph data. This would allow the router to only consider load balancing when determining where to send the next query. Adding cache to each processor changes that since a certain processor might be better prepared than others to answer a query if it has data in its cache that allows it to make fewer requests to the storage tier. This is where smart routing comes into play. We have developed two smart routing schemes that allows the router to make better choices by taking it to account the locality of a query in a graph. It makes the assumption that, if a processor handles a query from a certain part of the graph, other queries that are close to the same region are likely to have more cache hits if sent to the same processor, thus reducing response times and network usage.

The thesis is structured in the following manner: Chapter 2 contains information about the query we used for benchmarking as well as the current state of research regarding distributed query processing systems. In chapter 3, we go into more detail on how the system was developed and what choices were made when implementing it. Chapter 4 describes what is meant by smart routing followed by a description of each of our routing methods, from preprocessing to decision making. We evaluate our system under many different conditions and compare it to another graph query processing system in chapter 5. Finally, we conclude in chapter 6 and go over what could be the next steps for the system.
Chapter 2

Background and Related Work

In this chapter we will take a look at the $k$-hop neighborhood search query which is a good example of the type of query an online graph query processing system would handle and what we used as a benchmark query. Then we will examine current popular graph processing systems and understand some of the research that has been done to address their shortcomings.

2.1 $k$-hop neighborhood search

$k$-hop neighborhood search ($k$-NS) is a very simple but useful query. The input is one vertex of the graph and the result is composed of all vertices that are at most $k$-hops away from the input vertex. To illustrate a 2-hop search (2-NS), figure 2.1 shows what part of a graph that represents a social network might look like where vertices represent people and edges represent friendships. If a person would be interested in finding out how many people are friends of his friends, he needs to merely inform his vertex and run a 2-NS query.

We chose $k$-NS as a benchmark because it could be easily implemented and that it only accesses a small fraction of the graph near the input vertex, making it well suited for online graph query processing systems. A different query like the shortest path between two vertices would have similar behavior since, in essence, both can be implemented with breadth-first-traversal. Other examples of queries
Chapter 2. Background and Related Work

Social Network Example

Figure 2.1: 2-hop neighborhood search (2-NS) on a social network

that fall in the same category are: $k$-hop neighborhood aggregation (how many $k$-hop neighbors have a certain characteristic), top-$n$ nearest neighbors, $k$-hop random walks and personalized page rank computation.

2.2 Current graph query processing systems

Not all graph processing systems were made equal. Examples like PEGASUS [1], Pregel [2], GPS [3], GraphLab [4], GRACE [5], GraphChi [6], Giraph++ [7] and GraphX [8] can be considered offline graph processing systems. This is because they perform offline graph analytics, which is usually done iteratively, with batch processing, over the entire graph, e.g. page rank and graph partitioning. They typically follow a vertex or edge-centric scatter gather model and are ill suited for handling online graph queries for two reasons:

- Basic online graph operations, like breadth-first traversal do not have natural vertex-centric implementations [9], causing them to perform poorly.
- They were designed to handle one specific job at a time, not allowing multiple queries to run concurrently, which leads to low throughput.

Another example of graph query processing systems are the several distributed SPARQL engines that were developed to handle SPARQL queries, like [10, 11].
For these systems, data is usually modeled as a collection of triples and query processing uses traditional RDBMS. The problem with this form of processing is that it requires a large amount of join operations, generating large intermediate results which causes it to scale poorly. Graph traversal based systems are shown to be more effective in answering such queries ([12, 13]).

Systems or domain specific languages like Trinity [12], Green-Marl [14], Horton [15], Ligra [16], and [17] do facilitate graph traversal over small areas in the graph and therefore are able to quickly answer online query requests, like $k$-NS. They follow the same general approach where query processing and graph storage happen on the same machine. Since each query likely spans over multiple vertices, the manner in which the graph is partitioned between the system’s servers is critical for performance. If many queries exceed the limits of the servers’ partitions, the system will likely experience a large amount of inter-machine communication, slowing down response times. Another weakness in the design of these systems is that they are not prepared to handle skewed workloads. Since each server is assigned a fixed partition, if there is a hotspot and most queries happen to fall on said partition, one machine will be overloaded while the others remain idle. In addition, these systems do not handle well graph modifications. Typically the whole graph needs to be repartitioned when vertices or edges are added or removed.

Attempts have been made to solve some of these problems with techniques employed in PowerGraph [18], SEDGE [19], Mizan [20] and [21]. To address the inter-machine communication and workload issue, the general idea is to monitor the workload and dynamically redefine current partitions as well as duplicating parts of the graph by creating new partitions. Unfortunately, these operations are quite expensive and must be done while the system is running to be able to adapt to the current workload. This becomes a problem if the workload is erratic and frequently changes hotspots.

Our system fits in the online graph query processing with facilitated graph traversal category however it is able to bypass the common pitfalls of inadequate partitioning and unbalanced load. This is because the partitioning is encapsulated
by the storage tier and does not influence the amount of inter-machine communication. Furthermore, since each server is not assigned to a fixed partition, workload balancing is more easily done.
Chapter 3

System Design

In this chapter, we review general architecture decisions and implementation details of the system, discussing their pros and cons. Since the routing schemes’ design and implementation are quite elaborate, we devoted the subsequent chapter to them.

3.1 General design

As mentioned before, one of the main differentiating characteristics of the online graph query processing system we have developed is that is decouples query processing from graph storage in two tiers. As figure 3.1 illustrates, query processors do not keep a partition of the graph in their main memory and must access the data storage to retrieve any part of the graph they need to answer a query.

The advantage of this separation is threefold:

- In the case where one tier has a high load, for example if query processing requires a large amount of computation, it can be easily scaled-out by adding more machines. This advantage goes both ways, of course, since the number of servers could be reduced if they are sub-utilized.

- The graph storage tier encapsulation eliminates the need for any expensive and sophisticated partitioning or repartitioning mechanisms. A simple partitioning scheme, e.g. using a hash function, works just as well.
• Query processors are no longer assigned to a fixed graph partition and are more flexible, being able to answer queries that fall anywhere on the graph. This greatly facilitates load balancing.

However, we also must consider the drawbacks of having the graph storage apart:

• Query processors are likely to communicate frequently with the storage tier over the network. This adds an additional penalty to the response time of answering a query.

• It is possible that this design causes high contention rates on either the network layer, storage tier or both.

To mitigate these issues, several steps described in the next sections and chapter, in short, lessen the need for communication, reduce the amount of data transferred and leverage existing network hardware.

### 3.2 Graph Storage Tier

The storage tier is responsible for holding all of the graph data. This is composed of the id of each vertex and their connectivity information. For very large graphs, it is likely that a commodity server is not able to hold all of its information in memory. Therefore, the storage tier must be comprised of a group of computers.

To implement the system’s storage tier, we have chosen to use RamCloud [22]. This application is able to provide high throughput and very low read latency, in
the order of 5-10 µs for small reads. RamCloud is able to achieve this due to the fact that it keeps all stored values in distributed memory. In terms of API, the system acts as key-value database where the key hash of each entry determines on which server the value will be stored.

![RamCloud diagram](image)

**Figure 3.2:** General design with storage details

In the context of graph query processing, RamCloud is used to hold the graph data as an adjacency list. Every vertex in the graph is added as an entry in the storage where the key is the vertex id and the value is an array of 1-hop neighbors.

It is common, while processing queries, to need the adjacency values of several vertices at the same time. For example, when computing 2-NS, the processor must request the neighborhood information for each of the vertices in the 1-NS result. There would be a large amount of overhead if each request were made independently over the network. Fortunately, RamCloud offers a multi-read method in its API that allows the client to make batch requests and reduce communication costs.

### 3.3 Query Processing Tier

The query processing tier is where the actual query computation takes place. As shown in figure 3.3, query processing servers do not communicate between each other. They only receive requests and return results from and to the router. When computing the result of a query, they can request graph data from the storage tier if necessary.
Figure 3.3: General design with processor details

To reduce the amount of calls made to the storage tier for each query, cache has been added to every query processor. Whenever a value is retrieved from storage, it is saved in cache so the same storage call can be avoided in the future. The cache saves entries in the same format RamCloud does however it imposes a maximum data capacity where, whenever a new entry surpasses this limit, one or more entries are evicted. We chose the Least Recently Used (LRU) eviction policy since simple constant time insertion/read algorithms exist and it favors recent queries over old ones. This is expected to work well with the exponential moving average we use for Embed routing which is defined in section 4.5.

In chapter 2, we indicated that we chose k-NS as the main query to benchmark our system. The pseudo code in algorithm 1 shows how it was implemented. To achieve best performance not only does the algorithm use cache and RamCloud batch requests but it was also implemented in C++, as was the whole system. At the moment, each processor is only single-threaded which significantly reduces the systems throughput however the implementation was intended as a prototype and is adequate to validate the systems design and smart routing schemes.
**Data**: initial vertex, k hops

**Result**: result variable, containing k hop neighborhood of input vertex

```go
/* set initialization. sets ensure that all values are unique */
result, currentIds, nextIds, requestIds := empty set;
result.insert(initial vertex);
currentIds.insert(initial vertex);
for hop := 1 to k do
    for curId in currentIds do
        if curId is cached then
            for ngbId in curId’s neighbors do
                if result does not contain ngbId then
                    result.insert(ngbId);
                    nextIds.insert(ngbId);
                end
            end
        else
            requestIds.insert(curId);
        end
    end
    if requestIds is not empty then
        execute ramcloud batch request with ids from requestIds;
        for reqId in requestIds do
            for ngbId in reqId’s neighbors do
                if result does not contain ngbId then
                    result.insert(ngbId);
                    nextIds.insert(ngbId);
                end
            end
            insert cache entry for reqId;
        end
    end
    clear currentIds, requestIds;
    swap currentIds, nextIds;
end
```

**Algorithm 1**: k-NS with caching and batch requests
Chapter 3. System Design

In earlier versions of the code, it became clear that checking if vertex ids are contained in the result set was where the processing step spent most of its computing time, even though the very fast std::unordered_set was being used, which has average constant time find. To optimize the code, the set was replaced by a bit vector and an integer vector. The first keeps track of what is added to the second, thus acting as a set and reducing the find and insertion time.

3.4 Router

Aside from the routing schemes that are described in the next chapter, the router implementation is simple and lightweight. When the system is started, the router creates a thread for each processor and opens a connection so it can send requests when the workload is started. The router only sends the next request to the processor when it receives a response for the previous request, therefore it keeps a queue for each connection. By maintaining these queues on the router, it is able to have a sense of how busy a processor is by checking its queue's size. This also enables the router to rearrange requests to balance the load.

3.5 Communication

In order to reduce the communication overhead of the decoupled processing and storage, it was decided that the system would run on an Infiniband network with remote-memory-direct-access (RDMA). These modern network technologies allow the system to communicate with very low latency and high throughput. Fortunately, RamCloud is capable of running on this type of network with very little configuration. Not only was Infiniband used for processor-storage communication but also router-processor. This had to be implemented by hand and uses remote memory writing to transfer result data as well as message passing to notify either side when a request/response has arrived.

To allow a more fair comparison with other systems that are not compatible with Infiniband, we also wrote a communication layer for Ethernet (using TCP/IP) that can configured to override the default implementation. While this definitely
slows the system down, it does make the system more portable and easier to test on different clusters which could be useful in the future.
Chapter 4

Routing Schemes

Whenever a query arrives at the system, the router must decide to which query processor the request will be sent. For systems where each query processor is assigned a graph partition, this decision is simple; the query processor that contains the input vertex should handle the request. With our system, no such mapping exists and therefore we developed new routing schemes with the following objectives in mind:

1. Leverage each processors cached data
2. Balance workload even when it is skewed or contains hotspot
3. Fast routing decisions

We have named any routing scheme that is able to follow all objectives, smart routing. In the following sections, we go into more detail about what each objective entails and then we describe each the routing schemes we implemented, two of them simple (Next Ready and Hash) and two of them smart (Landmark and Embed).

4.1 Objectives

When processing a query, it is clear that, if a processor’s cache contents are relevant to a request it receives, it will able to answer faster, since less data
would need to be retrieved from the storage tier. So naturally, we desire that the router takes into account the contents of each processor’s cache before routing, leveraging its data.

While the router could inspect the cache of each processor, this would require network communication delay causing the system to slow down, thus conflicting with the fast routing decisions objective. With this, we conclude that the router must infer what is likely in each processor’s cache, only taking into account information that it already has.

To illustrate how this could be done, we take an example of a graph shown in figure 4.1. In this figure, vertices are represented as dots and edges are omitted for simplicity but we consider that they exist between dots that are nearby each other. The colored dots represent a workload of queries that will be executed on the graph.

![A Large Graph](image)

**Figure 4.1:** Workload with several hotspots

What we would expect from smart routing is, if a query is sent to a certain processor, all other nearby queries would be sent to the same processor. In our scenario, all blue queries, for example, would be routed to the same server. This is because, once a processor has handled a blue query, it has likely cached data
that would be relevant for handling other blue queries, since they are in the same neighborhood.

While always routing queries to processors that have the most useful cache data will likely reduce processing time per query, it might not be the best strategy overall. Sometimes a workload might be concentrated in one specific area of the graph. When that happens, all queries would be queued up for one processor, while others remain idle. That is a poor use of resources and lowers the system’s throughput. To rectify this, the router must also consider each processor’s load. On one hand, routing a query to another processor might lead to a slower response time however, for the system as a whole, it is better to balance the load and keep all processors busy. With our smart routing schemes, we are able to change how these factors influence the routing decisions to find the best tradeoff.

4.2 Next Ready Routing

*Next Ready* routing is the first simple routing scheme and it was created as a baseline for comparing with the smart routing schemes. The way it works is, the router decides where to send a query by choosing the next processor that has finished computing and is ready for a new request. The main advantages of using this scheme are:

- It is easy to implement
- Routing decisions are made in constant time
- No preprocessing is required
- The workload is well balanced

Even though this scheme presents several positive qualities, it fails to leverage processors’ cache. Every routing decision is made solely by choosing the next processor that is free and does not consider what a processor might have in cache.
4.3 Hash Routing

The second simple routing scheme implemented is Hash routing and it also serves as a comparison for the more sophisticated routing schemes. While using the scheme, the router applies a hash function on each query’s vertex id to determine where to send the request. The hash function used for our experiments is:

\[ \text{TargetProcessorId} := \text{VertexId} \mod \text{NumberOfProcessors} \]

This method shares most of Next Ready routing’s advantages except load balancing. Since each vertex id has a fixed target processor, defined by the hash function, it is easy to see that a workload with a single repeated query would only be routed to one processor. In addition, even if the whole workload is not completely targeted on one server, some queries naturally take longer than others, overloading some servers.

To address these issues, an additional mechanism was added to Hash routing called Query Stealing. Whenever a processor is idle and ready to handle a new query, if it does not have any other requests assigned to it, it may receive a request that was originally intended for another processor. Since queries are queued in the router, the router is able to take this decision when necessary, making sure there are no idle processors when there is work to be done.

Similar to Next Ready routing, this routing scheme does not consider the processor’s cache contents, however, since it employs fixed routing, it will likely send a repeated query to the same place it was sent earlier, possibly getting better use out of the cache.

4.4 Landmark Routing

The first smart routing scheme developed was Landmark routing. This method requires a preprocessing phase that creates a metric we call distance to processor. Strictly speaking, the processor is not part of the graph, so there is no distance between a vertex and a processor, however we use this term loosely and it is intended to indicate how close a vertex is to the other vertices that are routed to this processor.
The scheme gets its name from the idea that, if we define a small set of vertices in the graph to be landmarks and find the hop distance of every vertex to every landmark, we are able to estimate the distance between any two vertices, as described in work by Potamias et al. [23]. Also, it is proposed in the same paper that landmarks should be selected due to their degree and how well they are spread out over the graph.

4.4.1 Preprocessing

Preprocessing for Landmark routing is comprised by three general steps, shown in figure 4.2.

![Landmark Preprocessing Diagram]

**Figure 4.2: Landmark preprocessing steps**

The first step is to select a certain amount of landmarks and find their distances to every vertex. The process is done sequentially, iterating from the highest degree vertices down. For each vertex, if it is too close to any previously computed
landmark, we are able to quickly discard it since we already have computed distance information for that landmark. If it is far enough, we then find its distance to every other vertex using breadth-first-traversal. The worst case time complexity is $O(L \times E)$, where $L$ and $E$ are the number of landmarks and graph edges, respectively. The amount of information this step generates is $O(L \times V)$, since we now have the distance of every vertex to each landmark.

Once we have the hop distance information, we spread the landmarks out between the processors. We do this by assigning a main landmark for every processor and then every other landmark is assigned to the processor which contains the closest main landmark. In case of ties, the processor with the least amount of landmarks assigned is favored. With the intent of having the landmarks well spread between the processors, each main landmark is chosen as follows:

1. The first two are the two landmarks that are the farthest away from each other.

2. Each subsequent one is the farthest landmark from every previously defined main landmark.

This step takes $O(L \times P)$ time, where $L$ is the number of landmarks and $P$ the number of processors. Since both $L$ and $P$ are relatively small, this step is very fast.

Now that we know which landmark is assigned to which processor, we can finally calculate our distance from vertex to processor metric. The distance of vertex $V$ to processor $P$ is defined by the minimum distance of $V$ to every landmark assigned to $P$. Therefore, we end up with a distance value for every vertex to every processor. This information is kept in the router and takes up $O(V \times P)$ and needs $O(V \times L)$ time to compute.

As what one might realize, the information of the amount of processors in the system setup is used in the preprocessing step. This makes the routing scheme less flexible to the addition or removal of processors during runtime, since all processing that happens after finding the distances to the landmarks must be reevaluated.
4.4.2 Routing

To decide where to route a certain query, the Landmark Routing scheme checks the pre-computed distance of the vertex to processor metric for each processor and chooses the closest one. As a consequence, the routing time is linear with respect to the amount of processors ($O(P)$). This turns out not to be an issue since the amount of processors is small.

In contrast to previously described routing schemes, this method is able to leverage what exists in the processors’ cache. This is because it is likely that queries that are in the neighborhood of each other will have similar distances to the processors causing them to be routed in similar fashion.

Not only can the distance metric be useful for finding the best processor for a certain query but it can also be used for smarter load balancing. Suppose the case that the closest processor for a certain query is very busy. Since the distance metric gives us the distance to all processors, we are able to choose the second, third or so on closest processor. As with the routing, this form of load balancing will also have similar behavior with the neighborhood of the query and therefore will be able to make better use of the processors’ cache.

In practice, it can be tricky to define exactly when a query should be routed to its next best option. It is likely dependent on factors like the system’s performance, workload and type of query. We propose the following formula that calculates the balanced distance to processor:

\[
\text{BalancedDistToProcessor} := \text{DistToProcessor} + \frac{\text{ProcessorLoad}}{\text{LoadFactor}}
\]

With this, the query is routed to the processor with the smallest resulting value. In our system, we use the size of a processors’ queued requests as a measure of its load, while the load factor is a configurable positive value which allows us to tune how much the load affects the balanced distance to processor. We show in the experimental results how this parameter can affect the system’s performance.
4.5 Embed Routing

*Embed* routing is the last routing scheme developed. Akin to *Landmark* routing, it also possesses a preprocessing step which enables it to calculate a *distance to processor* metric, however the method in which it is done is quite different. Nevertheless, the metric can be used to make routing decisions that leverage processors’ cache and balance the workload in a similar fashion.

In short, the idea of this scheme is to embed a graph into a N dimensional space and use the resulting vertex coordinates to determine how far a vertex is from the recent history of vertices a processor has handled. In figure 4.3, we show an example of an embedding. On the left, we have a small graph where each vertex is labeled with a letter and, on the right, its embedding in two dimensional Euclidean space, plotted on a graph.

![Graph Embedding](image)

**Figure 4.3:** A graph embedded in two dimensional space

As we can see, the distance between every vertex connected by an edge is close to one unit on the plot’s scale. This is because the embedding is done in a way where the Euclidean distance between two vertices’ coordinates is as close as possible to the actual hop distance in the graph. So, not only are distances between neighbors maintained but also between vertices that are 2+ hops apart.
As a consequence, it is possible to have an estimate of the hop distance between any two vertices in $O(D)$ time instead of using traditional breadth-first-traversal that takes $O(E)$ time, where $D$ is the number of dimensions in the embedding and $E$ is the number of edges in the graph. Theoretically, it would be possible to calculate the actual distance between each vertex pair beforehand and store all values in a map, allowing constant time lookup however this map would have to be impractically large for graphs that have hundreds of millions of vertices, taking up $O(V^2)$ space.

### 4.5.1 Preprocessing

A method to generate embedding coordinates based on a graph in order to quickly estimate distances between vertices was first proposed by Eugene Ng et al. [24] in the context of predicting internet network distances. It is done by minimizing a prediction error function using an algorithm called Simplex Downhill [25]. Unfortunately, for the case where we want to embed a graph with hundreds of millions of vertices, this method is unviable since it runs with $O(V^2*D)$ time complexity, where $D$ is the number of dimensions and $V$, the number of vertices in the graph.

Work has been done by Zhao et al. [26] to allow using the Simplex Downhill algorithm for large graphs. The general idea is to select certain vertices as landmarks (L) and run Simplex Downhill only on them. Once that is done, every other vertices’ (V) coordinates can be found by minimizing the distance prediction error between it and all the landmarks. The approach produces a much more reasonable asymptotic running time, where the first step takes $O(L^2*D)$ and the second $O(V*L*D)$. Another benefit is, once the landmark coordinates are calculated, the second step is completely parallelizable per vertex. We chose to use this method for our embedding.

To start, we select a certain amount of landmarks and find their hop distances to each other vertex. This has the same requirements as one of Landmark routing’s preprocessing steps and therefore follows the exact same procedure. After, we define the prediction error function and apply the Downhill Simplex algorithm on the landmark vertices. We use the squared normalized error function proposed
Chapter 4. **Routing Schemes**

by [24] where the error value is made proportional to the actual distance and then squared.

\[
    f_{\text{error}}(v_1, v_2) := \left( \frac{\text{hopDist}(v_1, v_2) - \text{euclideanDist}(v_1, v_2)}{\text{hopDist}(v_1, v_2)} \right)^2
\]

This function was chosen since we care about the precision of small distances in our graph. For example, a 1 hop error compared to a measured distance of 2 hops is more severe than a 1 hop error on a measured distance of 10. The total error for each iteration of the algorithm is simply the sum of the squared normalized errors between each pair of distinct landmarks.

Finally, once all landmark coordinates are obtained, we proceed to calculate the coordinates for every other vertex. As stated previously, this step may be parallelized, where, for each vertex, we apply the Downhill Simplex vertex once again with the same error function but this time we minimize the prediction error of the current vertex in relation to every landmark. This step gives us D coordinates for every vertex and therefore we need \(O(V^D)\) space on the router to hold this information.

A benefit of this routing scheme in relation to **Landmark Routing** is that the preprocessing is independent from system topology, allowing, for example, more processors to easily be added during runtime, if needed.

### 4.5.2 Routing

When routing, we have access to each vertices’ coordinates. By keeping an average of the vertices’ coordinates that the system has sent to each processor, we are able to have a sense of which vertices a processor has handled in the past and also infer what is likely to be held in its cache. As a consequence, we are able to calculate the distance of vertex to processor metric by calculating the distance of the vertices’ coordinates to the historical mean of the processor.

Since recent queries are more likely to influence the contents of the cache then older ones, we substitute using the average function to determine a processor’s coordinates to using the exponential moving average (EMA). As shown below, this function allows us to determine how much new queries affect or update the EMA with a parameter \(\alpha\) that is configurable.
\[ EMA_{\text{coord}} := (1 - \alpha) \ast \text{QueryVertex}_{\text{coord}} + \alpha \ast EMA_{\text{coord}} \]

Analogous to Landmark routing, we now have a distance to each processor for a query and are able to make routing decisions taking into account cache leveraging and load balancing in a similar manner. However, the computation time differs slightly, whereas with this method, it is \( O(P \ast D) \), \( P \) being the number of processors and \( D \) the number of dimensions. Nevertheless, smart load balancing is done with the same formula described in 4.4.2.

**In summary**, we have shown four different routing schemes. The first two are simple due to their implementation and lack of preprocessing while the last two are better prepared to take advantage of processors’ cache contents. It is expected that the latter will perform better, especially when the workload presents hotspots. We will investigate these behaviors in the next chapter.
Chapter 5

Experimental Results

In this section we explore how our system behaves in several different scenarios. We see how the choice of certain parameters during preprocessing can affect performance. Also, we validate the effect changing runtime system configurations like adding more machines to the processing tier or simply adding more cache to each machine. In addition, we investigate how different routing schemes handle a selection of workloads. Lastly, we compare our system to a reference query processing system and also explore how robust the preprocessed data is to graph modifications.

5.1 Setup

All of the experiments in this chapter have been run on the Euler cluster at ETH Zürich. This cluster is comprised of 12 servers with Intel® Xeon® CPU E5-2609 quad core processors and is interconnected by both a 10 Gbps Ethernet network and a 40 Gbps Infiniband network. Most experiments use the following configuration unless otherwise stated: 1 server as the router, 4 servers in the processing tier, 4 servers in the storage tier, and communication over Infiniband.

The graphs used in the experiments are listed in table 5.1. They were selected because they are examples of real world large graphs that have the number of vertices and edges in the order of several million.
Table 5.1: Large graphs used in experiments

<table>
<thead>
<tr>
<th>Graph name</th>
<th>Number of vertices (in millions)</th>
<th>Number of edges (in millions)</th>
<th>Size on disk (in GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web UK</td>
<td>106</td>
<td>3,302</td>
<td>56.1</td>
</tr>
<tr>
<td>Friendster</td>
<td>125</td>
<td>1,806</td>
<td>31.2</td>
</tr>
<tr>
<td>Memetracker</td>
<td>185</td>
<td>354</td>
<td>7.7</td>
</tr>
<tr>
<td>Freebase</td>
<td>50</td>
<td>46</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Web UK represents part of the World Wide Web while Freebase is a knowledge graph that was built from many sources including wikis. Memetracker is also derived from the internet; its structure is based on how popular quotes and phrases are and how they were spread online, on news sites and blogs. Lastly, we also run experiments on Friendster, which represents a social network. By default, most of the experiments done were evaluated on Web UK since it is the largest graph of the set.

Either mean response time or throughput measurements were taken for each experiment. For the first case, the measured quantity is the average time it takes for a query to be sent from and returned to the router, fully answered. Load balancing is always enabled so some routing decisions might be made to favor overall throughput in detriment of response time. While throughput is also measured at the router, the value is defined by how long the system takes to fully answer 1000 queries of a specific workload. In most experiments, these values are obtained with the processors’ cache initially empty and also with enough capacity to not need any cache entry evictions.

In the results where different routing schemes are compared we also add an extra configuration: No cache. While this might not be considered a routing scheme, it is valid to also verify how the system behaves when it does not need to check or maintain a cache. When running in No Cache mode, all queries are routed using the Next Ready scheme and a no operation (NOP) cache is put in place of the real one.

The items below describe how the parameters of the system were adjusted. They are valid for all experiments that do not otherwise state they have been modified.
They were chosen since they have been found to work well however experiments were also done to evaluate how the system behaves with alternative values.

- The amount of landmarks selected are 96 with at least 3 hops of separation from each other. This is used in both smart routing schemes’ preprocessing steps.
- 10 dimensions were used for the graph embedding preprocessing.
- $LoadFactor$, a load balancing parameter for smart routing schemes, was set to 20
- $\alpha$, the parameter used for updating $Embed$’s routing scheme EMA was set to 0.5

5.2 Workloads

A selection of different workloads were created to evaluate the system. While having a real world workload would be beneficial to understand the system’s behavior under more realistic usage, such workloads are difficult to come by and it was not possible to acquire one during the thesis. Therefore, workloads that represent certain query patterns were developed.

5.2.1 Locality based

Two workloads pertain to this category. In both of them, one hundred different locations in the graph were chosen uniformly at random. Then, for each one of these locations, ten different queries were selected at a maximum distance of one hop for the first workload ($Locality\ 1 \ hop$) and two hops for the second ($Locality\ 2 \ hop$). Finally, each set of queries were grouped together in the total workload sequence, that is, the first ten queries are from one location, the second ten are from another and so on. These workloads are intended to evaluate how the system behaves and adapts when there are several query hotspots throughout the graph.
5.2.2 Uniformly selected

This workload (*Uniform*) was created by simply selecting one thousand queries uniformly at random. In contrast to the locality based workloads, this is less likely to have queries that are neighbors of one another.

5.2.3 Concentrated

The last two workloads are considered in the concentrated category. They were created using different methods but they define queries at are strongly focused in certain areas of the graph. The first (*20 Repeated*), chooses twenty vertices uniformly at random and repeats each one fifty times in a row for a total of one thousand queries. The second (*2 Hotspots*), chooses two distinct spots on the graph and select five hundred queries in the one hop neighborhood of each spot. The first five hundred queries are from the first spot and the last five hundred are from the second.

5.3 Experiments

In this section we investigate system behavior by analyzing the results of each experiment.

5.3.1 Locality based workloads with 3-NS queries

In this experiment, we run both *Locality 1 hop* and *Locality 2 hop* workloads evaluating 3-NS queries on Web UK. The mean response time results can be seen in figure 5.1.

The first thing that we can notice is that the result for both workloads are very similar aside from scale. The difference in scale is explained by the fact that *Locality 1 hop* and *Locality 2 hop* are in fact two different workloads, while the first has a mean result size of around 717 thousand ids, the second is smaller, at around 367 thousand.
Another aspect of this result is that it is clear that adding cache to the processors for this configuration reduces mean response times significantly. Even when using simple routing schemes, the system is able to obtain up to a 44% reduction.

Figure 5.1: Mean response times for locality based workloads and 3-NS query

Figure 5.2: Cache hits/misses for locality based workloads and 3-NS query
Finally, we can observe that the smart routing schemes, Landmark and Embed, are both outperforming the other configurations. As discussed in chapter 4, the smart routing schemes were designed to make better use of processors’ cache, thus obtaining more cache hits and accessing the data storage less to improve overall response time. In figure 5.2, we can see that the smart running schemes are indeed obtaining more cache hits and this a likely explanation for their bettered performance.

5.3.2 Locality based workload with 1-NS, 2-NS and 4-NS queries

This experiment is also run on Web UK and is similar to the previous one except that now three different queries are executed (1-NS, 2-NS, and 4-NS) instead of one while only one workload is used (Locality 2 hop). As can be seen in figures 5.3 and 5.4, the results for 2-NS and 4-NS are similar to what has been observed for 3-NS however 1-NS behaves quite differently.

The most surprising characteristic is that the No Cache configuration is actually performing just as well or slightly better than other routing schemes. Furthermore, we observe no or slight difference between the cached schemes and also
that the mean response times are in the order of microseconds, in contrast to other experiments. When verifying the cache hit count for each method, it was found that there was not one single cache hit for any of the routing schemes.

Although unexpected, after careful consideration, the result makes sense. 1-NS
queries are very simple to process; once one arrives at the processor, the cache or the decoupled storage needs to be checked only once, since the result of 1-NS is a vertices’ immediate neighbors. Therefore, there can only be a cache hit if the same query is requested twice, which does not happen in Locality 2 hop.

As a consequence, the smart routing schemes do not perform better to simple schemes since the contents of the processors’ cache do not contain any useful data to be leveraged. Moreover, all cached routing schemes perform two additional operations compared to No Cache: they always check if the cache contains an entry and store any data retrieved from the storage tier. While both these operations are fast, they do take time and do not bring any benefits in this scenario. This explains why No Cache’s response time can be smaller than other scheme’s.

Considering the results for 4-NS, it can be seen that all routing schemes perform much better than No Cache however the difference between smart routing schemes and simple ones has diminished compared to 3-NS queries. Smart routing schemes still obtain more cache hits per query and have lower response times however, when handling 4-NS queries, they must process a much larger amount of data. The mean result size for 3-NS queries is 367 thousand whereas, for 4-NS, it is 2 million. This means that processing takes up a more significant part of the response time, reducing the possible benefits afforded by smart routing.

5.3.3 Concentrated workloads with 3-NS queries

Due to how the concentrated workloads (20 Repeated and 2 Hotspots) were created we expect that having cache is a great advantage. The charts in figure 5.5 confirm this hypothesis when running these workloads on Web UK with 3-NS queries. The routing schemes that use cache have large reduction of up to 75% when compared to No Cache, nonetheless smart routing methods do not seem to outperform simple ones.

Taking a look at each method’s cache hits, plotted in figure 5.6, we can note that all cached methods have similar results, causing them to generally have the same performance. We infer that this behavior is likely due to how the system balances the load for these workloads.
Let us consider the example of Next Ready routing the 20 Repeated workload and compare it with Embed. As described previously in this chapter, this workload’s first 50 queries are the same. When these queries arrive at the server, Next Ready balances them out, resulting in each processor working on the exact same query.
Therefore, all processors will have the same content in their cache and will be able to effectively handle subsequent queries in the same manner, independent of which processor they are routed to. Considering Embed, the initial queries likely tend to be sent all to one processor since the distance between two identical queries is equal to zero. When this happens, the processor’s queue grows, giving more weight to the load balancing factor of the scheme’s routing decision formula. With that, some of the identical queries are spread out, causing all processors’ cache to have similar contents thus creating a similar configuration to Next Ready. With the same arguments, the other schemes’ and the 2 Hotspots’ results can be explained.

These results indicate how well the system design is able to cope with highly skewed loads. If the system had fixed processors for each partition, it would likely be underutilized, however, this experiment shows that even simple routing schemes are able to balance the load effectively.

5.3.4 Uniform workload with 3-NS queries

With this experiment, we evaluated how the system behaves when queries are chosen completely at random. Since queries are not likely near each other, we do not expect that cache will be very well utilized.

We can observe the mean response times in figure 5.7 when running this workload with the Web UK graph and 3-NS queries. In the results, having cache does not give the system a significant advantage for most of the methods. Similar to the 1-NS query results, the contents of each processors cache does not likely have data that is useful for the next queries and therefore maintaining the cache, by always storing data read from the decoupled storage, becomes a time investment that doesn’t pay off. Surprisingly however, the Landmark routing scheme was able to significantly outperform the rest.

To investigate the cause of this, we take a look at each method’s cache hit count in figure 5.8. Here we confirm that Landmark is able to get a larger amount of hits than the rest. This means that, even though the queries were selected at random, there was some locality between them that this routing scheme was able to take advantage of. On the other hand, the Embed scheme was not able
to find the best processors for each request. We believe this is due to its use of EMA. While Landmark will likely send two queries, A and B, that are in the neighborhood of each other to the same processor even if many queries occurred
between them, this might not happen with Embed. This is because EMA favors more recent values than older ones, so it would be possible for the multiple in between queries to change the value of the processor that handled query A enough so that query B might not be sent to it. Since in a uniformly chosen random workload it is more likely that if any two queries are in the neighborhood of each other they are not necessary close by in the workload execution sequence, Landmark routing has an advantage.

What we conclude from this is that Embed routing is more sensitive to the ordering of the queries than Landmark and therefore the latter is better suited for handling workloads that do not present such order.

### 5.3.5 Varying the cache capacity

In all previous experiments, each processor always had enough cache capacity to never need to discard any entries however, in real world applications, this is not likely feasible. In this experiment, we verify how the system behaves when it needs to start evicting cache entries to add new ones. For the configuration, we take one that worked well in previous experiments: locality based workload, 3-NS queries on Web UK where cached routing schemes clearly outperform No Cache and that smart routing schemes perform better than simple ones.

In figure 5.9, we see the queries’ average response times at several different cache capacities. At the largest, 4 GB, no evictions occur, therefore there is no additional performance gain by increasing the capacity.

We see that, for the experiment’s scenario, having cache with less than 64 MB results in worse response times than what was obtained after disabling the cache, represented by the horizontal dashed gray line. The explanation for this is similar to what has been observed when running 1-NS queries or uniform workloads; there are not enough cache hits to justify its maintenance, shown in figure 5.10. However, in this case, the low hit count has a different cause. Since the cache does not have much space, it ends up evicting entries that might have been useful in the future.

Interestingly, we can see that the Next Ready performs significantly better than Hash for small cache sizes. Once cache is increased to a certain level, both
Chapter 5. Experimental Results

Effect on Locality 2 Hop Workload Response Time with Respect to Cache Size and Routing Schemes

**Figure 5.9:** Mean response times for different cache sizes

Effect on Locality 2 Hop Workload Cache Hits with Respect to Cache Size and Routing Schemes

**Figure 5.10:** Cache hits for different cache sizes
methods have similar response times. This is confirmed by each method’s cache hit amount in figure 5.10. A possible explanation for this is, \textit{Next Ready} is more likely to have consecutive queries from the workload to be sent to the same processor than \textit{Hash} does because the former distributes queries equally amongst the processors while the latter sends queries to whichever processor is free. Since the cache is so small, hits are likely obtained by consecutive queries. Once cache is increased, this becomes less of a factor.

Using the same measurements, we can evaluate the system through a different viewpoint. We can ask the question, how much cache capacity is needed to reach a response time of 86ms, the breakeven point of deciding whether or not to add cache. We can observe in figure 5.11 that the smart routing schemes are able to achieve the same performance with a much lower memory footprint.

### 5.3.6 Cold cache / warm cache

As mentioned before, all experiments are done with the cache initially empty (cold cache). Here, we explore what happens when the cache is already warmed up or, in other words, has contents. This is done by executing a workload and
then re-executing, only measuring the second run. Two workloads are tested, *Locality 2 hop* and *Locality 20 repeated* with 3-NS queries and Web UK graph.

![Graph of Locality 2 Hop Workload Response Time](image)

**Figure 5.12:** Mean response times comparison for cold and warm cache running Locality 2 hop workload

![Graph of 20 Repeated Workload Response Time](image)

**Figure 5.13:** Mean response times comparison for cold and warm cache running 20 Repeated workload
We can see in figure 5.12 that all routing schemes are greatly benefited by warm cache, resulting in mean response times in the low 20 milliseconds. While all methods have similar times with warm cache, the *Hash* scheme has the best time. This can be explained by the fact that almost all of cache reads that occur with this method will result in hits because it is a fixed routing method meaning that all queries will be routed to the same processors. In contrast, there is *Next Ready*, which decides where to route requests on the fly, depending on the current load. This method is less likely to send the same query exactly where it sent it in the first run and therefore has worse performance.

By taking a look at figure 5.13, we see less of an improvement of warm compared to cold cache. This is due to the nature of the *20 Repeated* workload. Since it inherently causes the processors to have more cache hits, even when the cache is cold, by using warm cache, there is a benefit but it is not as large as what has been observed from the previous workload.

### 5.3.7 Preprocessing parameters

Parameters like the amount of dimensions for embedding, number of landmarks or their separation directly affect how long preprocessing takes and how much space is needed on the router to hold generated data. Nonetheless, we are interested in investigating how they indirectly affect the runtime performance of smart routing schemes.

As discussed in chapter 4, there are a few options when it comes to selecting landmarks. Here, we first explore selecting the vertices of highest degree and evaluate how the amount of landmarks affect the system’s response time. We can see the results in figure 5.14, using a locality based workload, 3-NS queries and Web UK graph. Only *Landmark, Embed*, and the best simple routing result are shown since the first two are the only ones affected by this parameter and the last serves as a baseline.

For the *Embed* routing scheme, we can observe that, at 128 landmarks, there is a slight gain in performance, however, for both routing schemes, measurements do not vary much.
Next we take a look at how landmark separation affects performance using the same configuration. In figure 5.15, both smart routing schemes’ mean response times are plotted according to different minimum distance between landmarks.
We can see that, similar to the previous experiment, the separation does not have great influence on performance, with Embed’s best performance at two hops of separation and Landmark at four hops.

Lastly, we consider the performance implications of the number of dimensions for an embedding. For this experiment, we create several different embeddings, from 2 to 18 dimensions, and figure 5.16 shows the prediction error of each of them. More specifically, the plotted values are the sum of the squared distance predictions errors of the distances between the Locality 2 hop workload vertices and the graph’s landmarks. The formula used is the same as described in section 4.5.1 however now we use it as a metric to evaluate how well the embedding is able to predict distances for a certain workload.

![Figure 5.16: Prediction error metric for Locality 2 Hop workload](image)

For the range evaluated, we can clearly see that adding more dimensions lowers the error metric sub-linearly. To verify how the accuracy translates to system performance, we measured the mean response time of 3-NS queries and Locality 2 hop for each embedding, shown in figure 5.17.

We can see that, although lower dimensions have larger errors, the effect on system performance is small. In general, mean response times do not vary much, with a small outlier at 14 dimensions. The worse performance at 14 dimensions is
caused by a lower cache hit ratio than other configurations however the result for 18 dimensions shows that it is not a trend. We believe that the worse performance point is simply the result of the Embed scheme’s variance.

In general, what we have found is that the preprocessing parameters evaluated have a small effect on system performance. These ranges were selected considering experimental results reported in [26] and also values that would be practical for a system to use. In addition, even the worst measurements obtained for the smart routing schemes were significantly faster than simple routing schemes for the same configuration.

5.3.8 Runtime parameters

The system possesses two parameters that influence its smart routing schemes: \( \alpha \) and LoadFactor. \( \alpha \), described in section 4.5.2, only affects Embed routing and it can be defined as a number from 0 to 1, indicating how much new queries affect the query processor’s EMA. A value of 1 would mean that the processor’s EMA would be equal to the last query’s coordinates while a value of 0 would cause the processor’s EMA never to update, always maintaining its initial value.
In figure 5.18, we measure system’s mean response times according to different values of $\alpha$ using a locality based workload, 3-NS queries and Web UK graph.

![Effect of $\alpha$ on Response Time](image)

Figure 5.18: Mean response times for Embed routing scheme in relation to different $\alpha$ values

For comparison, the best simple routing scheme is indicated with a dashed line. We can see that, for all values of $\alpha$ measured, the Embed scheme is faster than simple routing and the effect of the parameter is small, varying response time less than 10%. Nevertheless, we do see an improvement at $\alpha$ equal to 0.75. Better performance at high $\alpha$ values is expected since the workload contains queries from hotspots in sequence, which is benefited from a routing policy that takes into account recent history with a high weight.

Now we take a look at LoadFactor. Here we evaluate both smart routing schemes since they use the smart balancing scheme described in section 4.4.2. In figure 5.19, we use the same configuration as the previous experiment, but now we measure throughput, since LoadFactor is intended to improve overall system performance, not necessarily individual query response times.

We can see that, at around the values of 10 to 100, the best throughput is achieved for both smart routing schemes. Furthermore, when LoadFactor is very
large, i.e. much larger than possible queue sizes, the load balancing influence in the routing formula tends to zero. The measurements at 10,000 indicate how large the system’s throughput would be if smart balancing were not implemented.

What can be concluded is that both $\alpha$ and $LoadFactor$ runtime parameters can be tweaked to enhance the performance of the system however, if they are not optimal, the smart routing schemes still perform well since routing decisions based on query locality are still being made.

### 5.3.9 Performance on other graphs

Most of the experiments were done on Web UK. Here we investigate how the system behaves when evaluating queries on other graphs. The graphs chosen were Friendster, Memetracker, and Freebase. A description of each one can be found in section 5.1 and they have been queried using 3-NS and a locality based workload.
Friendster: We first take a look at the results obtained for Friendster, in figure 5.20. We can observe that, in contrast to what has been seen for Web UK, using cache does not benefit the mean response time greatly. If we compare each of their cache hit rates, Web UK achieves up to 79% while Friendster only manages 59%.

![Friendster Graph Response Time with Locality 2 Hop Workload and Different Routing Schemes](image)

**Figure 5.20:** Mean response times for Friendster Graph

In an attempt to explain the low hit rate, we developed a metric called hotspot overlap metric. In the case of $k$-NS queries, it takes the $(k-1)$-NS of the center of the hotspot and verifies how many intersections occur with the $(k-1)$-NS of each other vertex in the hotspot. The results are summed up and then divided by the sum of the result sizes obtained from $(k-1)$-NS of the hotspot’s vertices. The formula is shown below, using the notation of $NS_k$ instead of $k$-NS, to facilitate reading.

$$NS_k \text{ HotspotOverlap} = \frac{\sum_{i=1}^{\left|\text{Hotspot}\right|} NS_{(k-1)}(i) \cap NS_{(k-1)}(\text{center})}{\sum_{i=1}^{\left|\text{Hotspot}\right|} NS_{(k-1)}(i)}$$

This formula is intended to indicate how many cache hits relative to reads a workload achieves due to the connectivity of the vertices in a hotspot. We
compare the amount of vertices in the \((k - 1)\)-NS result since that is the amount of cache reads that occur on any \(k\)-NS query.

The overlap metric results for Web UK and Friendster are 50\% and 3\%, respectively. This clearly shows for Friendster that there is not much intersection between vertices in a hotspot and therefore the content of processors’ cache is not likely useful for future queries, causing low hit rates.

In addition, the workload’s mean result size is 2.9 million for Friendster, compared to 367 thousand for Web UK. This ends up causing the actual computation that occurs in the processing tier to take up a larger portion of the response time, reducing possible benefits of avoiding storage access through efficient use of cache.

**Freebase:** Running the system on Freebase, we have the results presented in figure 5.21. We can observe this time that using cache in the processors has a large positive effect on response times, similar to what we have seen with Web UK. On the other hand, there is no significant difference between using smart routing schemes compared to simple ones.

![Figure 5.21: Mean response times for Freebase Graph](image-url)
To understand why, we take a look again at the graphs’ hotspot overlap metrics. Web UK and Freebase have 50% and 49%, respectively. This indicates that queries from the same hotspot on Freebase would have a similar amount of cache hits to Web UK. Nevertheless, Freebase presents larger hit rates, over 88% for all Freebase routing schemes compared to 79% for Web UK’s best. Therefore, there is another important factor in play which is not dependent on hotspot overlaps.

If we consider Freebase’s size, it is the smallest graph evaluated however the average amount of vertices traversed per query is in the order of millions and it is around 4 times larger than what is seen with Web UK. Consequently, it is much more likely that query results overlap during the execution of a workload. This means that all cached routing schemes benefit from the fact that previous queries in the workload have warmed up the cache for later ones, even though they do not necessarily pertain to the same hotspot. This behavior is similar to what we observed when testing concentrated workloads where all processors’ cache end up having similar content.

**Figure 5.22:** Mean response times for Memetracker Graph

**Memetracker:** Lastly, we observe Memetracker’s result in figure 5.22. For this graph we see that using cache shows improvement however smart routing methods only slightly improve response times. It presents a hotspot overlap
metric of 8%, which is larger than Friendster’s, indicating there is a higher chance of being able to leverage cache however it is not as high as Web UK’s value. The average result size relative to graph size is smaller than what has been observed with Freebase, indicating that a higher relative amount of cache hits is obtained from query locality.

5.3.10 Scaling out the processing/storage tiers

As previously mentioned, one of the main benefits of separating the processing and storage tiers is that they can be scaled independently. In this section, we investigate how performance is affected when doing so.

**Processing tier:** For most experiments, the processing tier was comprised of four servers. In this experiment, the number of servers will vary from one to seven, while the *Locality 2 Hop* workload is applied with 3-NS queries on Web UK. The maximum of seven servers was chosen since the cluster on which all experiments were run on has twelve machines, being that one is reserved for the router and four for the storage tier. In figure 5.23 we show throughput of each configuration since adding machines to a system typically is intended to increase this metric.

First of all, we can see that, for one processor, all cached routing schemes have the same throughput. This makes sense since there is only one routing decision each scheme can make. When adding more servers, gradually the smart routing schemes scale-out better than the simple ones, although it is still sub-linear. The main reason for that can be seen in figure 5.24.

When there is enough cache capacity, having only one server is the best option in terms achieving the most amount of cache hits. The reason is clearly because all cache stores happen in the same machine; if any previous query has retrieved useful data, it will be found. Once more servers are added to the system, routing schemes start playing an important role to send queries where they are likely to have more cache hits. We can see in figure 5.24 that the *Embed* routing scheme achieves the best hit rates. This translates to better throughput.

Another behavior that can be observed is that the simple routing schemes’ throughput saturate at around three to five servers. At that point, adding new
Figure 5.23: System throughput for different amounts of query processors

Figure 5.24: Cache hits for different amounts of query processors
servers to the system is not beneficial. This is due to the same reason previously discussed, by adding more servers, the processors’ cache hit rates fall, causing them to handle requests more slowly.

It could be argued that the storage tier is being saturated however, as the results for No Cache scheme show, it is not likely the main cause since this scheme has the highest access rate to storage and its throughput climbs faster per added processor than the simple schemes. The scheme is benefited by the fact that it does not have to maintain cache. It is possible that, for larger amounts of processing servers, having cache would penalize more than benefit performance, causing the No Cache to surpass the simple routing schemes.

**Storage tier:** Analogous to what was done for the processor tier, the amount of storage servers is scaled from one to seven, where one server is the router and four are query processors. Although the Web UK graph takes up around 60 GB of memory, it was still possible to load it completely on to one storage server since the cluster machines have enough RAM. This is not likely possible on today’s commodity servers.

![System Throughput with Respect to Number of Query Processing Servers](image)

**Figure 5.25:** System throughput for different amounts of storage servers
As shown in figure 5.25, all routing schemes behave similarly according to the number of storage servers. The worst performance occurs when there is only one storage server and improves when adding new ones, up to four servers. After that, throughput saturates and slightly degrades after adding even more servers.

The performance boost for the first few servers is a clear indication that the storage tier was a significant throughput bottleneck. It is evident that, for this scenario, one or two servers are not enough to handle the demand created by the four query processors. Once more servers were added, the bottleneck was transferred to the query processors, as seen in the previous experiment.

The slight throughput reduction caused by having more than four servers is likely caused by the characteristics of how RamCloud functions. For this experiment, one coordinator instance was used. This coordinator acts as the storage interface, dispatches requests to the storage servers and aggregates the results. Once there are more servers to handle, there are more sources to aggregate, causing the coordinator to spend more time per request.

In this section, it was discovered that scaling out processing or storage can dramatically affect system performance. The flexibility that the system’s design affords is greatly beneficial for maximizing throughput. Furthermore, when adding machines to the processing tier, the routing schemes play an important part; choosing a smart routing scheme can enable added servers to function more efficiently.

In the scenario where a system administrator has an extra server and needs to decide whether to add it to the processing tier or the storage, a good rule of thumb is to simply add the server to the processing tier. If throughput does not improve, then the storage tier is likely overloaded so the machine should be used as storage. This suggestion is mostly valid for the Embed routing scheme, since it is the one that has been shown to scale out the best.

### 5.3.11 Comparison with other system

To validate the design decisions and implementation of our system, we compared it with a modern distributed graph querying system called SEDGE [19].
of the main attributes we wanted to compare against is that the system implemented a repartitioning that would adapt the graph’s partitions to the current workload. Unfortunately, we found that the freely available version of SEDGE does not have this feature included.

In an attempt to make up for the missing feature, we chose to partition the graph ourselves using an application called ParMETIS [27]. This application is able to partition weighted graphs by minimizing the edge-cut value. With this, we can define edge weights according to the workload we would apply, giving larger weights to edges that are traversed during query processing. By using the newly computed partitioning with SEDGE, we simulate the repartitioning that would be done according to workload.

One clear disadvantage that the SEDGE system possesses is that it is not compatible with Infiniband networks, thus all network communication goes through the Ethernet network, which has higher latency and lower throughput. To allow the comparison with our system to be fairer, we have enabled our system to also work completely on Ethernet, from the router to the storage tier. While this slows the system down considerably, it also makes the system more portable; Ethernet networks are more common and widespread.

Since query processing and graph storage in SEDGE occur on the same machine, it is not clear what would be an equal amount of machines for comparison between the two systems. It could be argued that, since the default configuration for our system uses four servers for query processing, SEDGE should also use four servers, however our system uses nine machines in total. To be safe, we run SEDGE with both configurations: four servers and nine.

Another important difference between the systems is that ours was implemented on C++ while SEDGE is in Java. While C++ programs typically run faster than their Java counterparts, SEDGE can make up for it since its query processors were implemented with multi-threading while ours weren’t.

The chart in figure 5.26 shows the results of our evaluations done on the Freebase graph using 3-NS queries and a locality based workload. What we can first notice is that our system is able to obtain a much higher throughput, even when running on an Ethernet network. Moreover, using the workload based partitioning did
perform better than the non-workload partitioning however the difference is very small. As expected, SEDGE’s best throughput was obtained when using the nine machines configuration however it was still much slower than our system’s slowest configuration.

A likely explanation for SEDGE’s poor performance for the experiment’s workload is, processing is separated by steps and each server has a fixed partition.

To implement the 3-NS query in SEDGE, we developed a 4 step algorithm:

1. Query vertex requests adjacent vertices of 1 hop neighborhood
2. 1 hop neighborhood requests adjacent vertices of 2 hop neighborhood; 1 hop neighborhood responds to query vertex with adjacent vertices
3. 2 hop neighborhood responds to query vertex with adjacent vertices; query vertex processes responses from 1 hop neighborhood
4. Query vertex processes responses from 2 hop neighborhood

All servers must finish the processing of the current step to be able to start the next. Since each query processor has a fixed partition assigned to it, it

---

**Figure 5.26**: Throughput comparison between our system and SEDGE (3-NS queries). * Non-workload based partitioning. ** Workload based partitioning.
may end up being assigned a higher load than other servers. This can create a situation where most servers are waiting for one server to complete processing so the system as a whole can’t start the next step, resulting in lower throughput. In contrast, our system does not have this problem since each query processor can process any request and a server is never idle waiting for another to finish.

5.3.12 Robustness to graph modifications

The preprocessing steps for both Embed and Landmark routing schemes take a substantial amount of time. Since their computation is based on the topology of the graph, once a modification takes place, i.e. adding or removing graph vertices, the preprocessed data ultimately becomes stale.

In this section we investigate how the smart routing schemes perform using outdated preprocessed data, in other words, how robust these schemes are to graph modification. For this experiment, we run all queries on Web UK at its original size, however we execute the preprocessing at several different reduced sizes. For example, at 20%, the preprocessing takes place on a modified Web UK where 80% of its vertices are selected uniformly at random and removed. This means that, after preprocessing had occurred, the graph grew fivefold. Other examples are 95% and 100% where the former’s graph grew around 5.3% and the latter’s remained unchanged. The results for this experiments can be seen in figure 5.27. As in previous experiments, the simple routing schemes do not possess a preprocessing phase and are therefore not affected by it. We show the best performing simple routing scheme as a comparison baseline.

It is important to note that we refrained to remove any vertex that was also part of the query workload. The reason for this is, if a vertex is not part of the reduced graph, it will not have preprocessing information related to it. We have proposals on what to do in case one would want to query such a vertex but since we did not implement or test it, we describe the idea in the future work section of the next chapter.

The results we find for the Embed routing scheme is expected. If preprocessing takes place on a graph that is very different than the current state, the coordinates obtained are not very useful. This changes when the graph size at
preprocessing is at least 90% of final size. We can also see that when the vertices coordinates are outdated, the Embed routing scheme performance degrades to similar levels obtained by the best simple routing scheme. Once the Embed scheme falls to that level of performance, it could be a good indication that the graph’s preprocessing should be redone.

In relation to Landmark routing, we have found it to be surprisingly robust. Even after the graph had grown 5 times in size, it achieves similar performance to what it would have if preprocessing was redone. It is likely that this is not indefinitely true and at some point below 20% performance should worsen. A possible explanation for this is that, since vertices were removed uniformly at random, the general structure of the graph did not change much and that the highest degree vertices at 20% are also high ranked degree vertices at 100%. This leads to the data that we obtained for the smaller graph size to still hold useful data after the graph has grown.

In short, we have learned that knowing how frequently your graph will suffer topological changes is an important factor to consider when choosing which routing scheme to implement. Although the Embed method shows best mean response times, they can quickly degrade after the graph has grown a certain
amount. While simple routing schemes are unaffected by graph change, *Landmark* outperforms them even after very large graph modifications meaning that it is likely the best suited for graphs that change over time.
Chapter 6

Conclusions

In this thesis, we presented a new design for online distributed query processing systems. Its defining characteristic is that graph storage and query processing are decoupled into separate tiers. While this decision facilitates tasks like load balancing and scaling the systems processing power or storage capacity, it also presents challenges to achieve the system’s best performance.

To avoid queries’ response times to be penalized by the tier separation, cache was added to each query processor, however, as shown throughout the experimental chapter, it is not a trivial task to decide whether to enable it or not. This decision depends on factors like the workload, graph topology, size of cache and effectiveness of the system’s routing scheme.

Four routing schemes were evaluated in the thesis, two of which were specifically tailored to make best use of query spatial and temporal locality. However, the smart routing schemes require a preprocessing phase which takes a substantial amount of time. In addition, a large amount of space is required to hold the preprocessed data on the router (in the order of number of vertices times a small constant). While it is viable to hold this amount of data on one commodity server, the simple routing schemes have no such requirement. Nonetheless, not only has it been shown that in many cases, the smart routing schemes outperform simple ones but even when the former are not favored, their performance does not become worse than the latter’s.
It has also been shown, that by using a more effective routing scheme, the system scales out more efficiently when adding more servers to query processing tier. In the case of Embed routing scheme, it is possible to add or remove processors during runtime, allowing the system to quickly adapt to sudden waves of queries. While Landmark routing scheme would require some extra preprocessing in case the number of processors changed, it is possible that the computation can be done beforehand for several different processing tier configurations, affording the scheme some flexibility.

When comparing system performance with another state-of-the-art distributed query processing system that was made available, it was found that our system performed very well, even when running it on a slower network. This not only validates the design as a viable option but indicates that it is not necessary to run the system on an Infiniband network for it to be useful.

Moreover, we have investigated how the system behaved when the graph had been modified and the preprocessed data is no longer up-to-date. Although there isn’t a method to update the preprocessed data effectively, we have found that the system showed a robustness to change, especially in the case for Landmark routing. This means that the whole graph would not need to be reprocessed even after several topological changes occur.

In summary, the system, together with its smart routing schemes, have shown to be effective under many scenarios. It is possible that their designs would be useful in real world applications.

**Future work** Throughout the thesis, only one query type has been evaluated: $k$-NS. Although it is a basic query that gives a sense on how other queries would perform, it would be interesting to actually evaluate the behavior of some case studies like $k$-hop random walk and personalized page rank computation.

In relation to graph modification, we have not explored how to create routing data needed for a vertex that did not exist during the preprocessing phase. In the case of Embed routing, we need to determine a vertex’s coordinates. Although it would be possible to use the Downhill Simplex to minimize the objective function proposed in section 4.5.1, this might be too slow to compute on-the-fly. A heuristic that could be used is to take the average of the coordinates of all its
neighbors. For *Landmark* routing, we need to determine the shortest distance to each landmark. A quick method of calculating this could be by taking the minimum distance of each neighbor to each landmark and add one, since the vertex is at most one more hop away from any landmark than its neighbors. The effectiveness of these heuristics would need to be evaluated. If positive results were obtained, it would lessen the need for periodical reprocessing.
Bibliography


