Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 37

Distributed Graph Processing and Partitioning for Spatiotemporal Queries in the Context of Camera Networks

Steffen Maaß

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dr. Kirak Hong (Georgia Institute of Technology), Prof. Dr. Umakishore Ramachandran (Georgia Institute of Technology)
Commenced:	April 22, 2015
Completed:	October 21, 2015
CR-Classification:	C.2.1, C.2.4, C.3

Abstract

English: This work presents a scalable, distributed architecture for processing spatiotemporal queries in the context of camera networks based on a graph structure. With the ever-increasing presence of cameras and the emergence of camera-networks, e.g., in the context of campus security, it becomes increasingly important to provide a robust and scalable architecture to store and retrieve detected events. In this work a distributed graph processing engine will be presented which is well suited for read and write tasks in the environment of spatiotemporal image-similarity based workloads. The key ideas presented in this work are the architecture of a scalable graph processing system well-suited for processing spatio-temporal queries and the design of a distributed and robust vertex-partitioning strategy for the graph which is being defined by the spatiotemporal attributes of the stored events. The work will show multiple lightweight heuristics for partitioning the graph among the nodes participating in the system, focusing on load-balancing between workers and high edge-locality for vertices. The system and the partitioning strategies will be evaluated, showing that the system scales with the number of workers and the problem size and is able to answer proportionally more queries per second. It will also be shown that the lightweight heuristics for partitioning the graph produce a relatively good balancing of the vertices on the worker-nodes and can be executed in an online-fashion, resulting in similar performance when compared to a traditional hash-partitioning while providing far superior edge-locality.

Deutsch: Diese Arbeit stellt eine verteilte und skalierbare Architektur zur Verarbeitung von räumlich-zeitlichen Anfragen auf Kamera-Netzwerken vor und basiert dabei auf einer Graph-Struktur. Aufgrund der allgegenwärtigen Präsenz von Kameras und dem Aufkommen von Kamera-Netzwerken, zum Beispiel im Bereich der Überwachung eines Universitätsgeländes, wird es immer wichtiger, eine robuste und skalierbare Architektur zur Speicherung und Auffindung von detektierten Ereignissen anzubieten. In dieser Arbeit soll daher eine verteilte Graph-Verarbeitungs-Architektur vorgestellt werden, welche für die Verarbeitung von Einfüge- und Anfrage-Operationen im Umfeld von räumlich-zeitlichen Bild-Ähnlichkeits-basierten Aufgaben gut geeignet ist. Die Haupt-Ideen dieser Arbeit sind dabei die Architektur eines skalierbaren Graph-Verarbeitungs-Systems zur Ausführung der räumlich-zeitlichen Anfragen und der Entwurf verteilter und robuster Knoten-Partitionierungs-Schemas für den Graph der detektierten Ereignisse. Dafür werden mehrere, leichtgewichtigte Heuristiken zur Partitionierung des Graphen zwischen den am System teilnehmenden Rechner-Knoten vorgestellt, wobei der Fokus auf einer Lasten-Verteilung zwischen den Rechner-Knoten des Systems liegt, bei gleichzeitiger Optimierung der Kanten-Lokalität. Das dabei entstehende System und die Partitionierungsstrategieen werden evaluiert, wobei die Skalierbarkeit des Systems sowohl in der Zahl der teilnehmenden Rechner-Knoten als auch der Größe des Graphen demonstriert wird, dies hat einen höheren Durchsatz an Anfragen pro Sekunde bei größerer Anzahl an verfügbaren Rechner-Knoten als Konsequenz. Zudem wird für die leichtgewichtigten Heuristiken zur Partitionierung des Graphen gezeigt, dass diese eine relativ gute Lastverteilung aufweisen und parallel zu anderen Systemfunktionen ausgeführt werden können, dies erlaubt ein ähnliches Performanz-Verhalten wie eine naive, traditionelle Hash-basierende Partitionierung, weist allerdings eine deutlich verbesserte Kanten-Lokalität auf.

Acknowledgments

First and foremost, I would like to express my gratitude to Prof. Dr. Kurt Rothermel for the support in creating this thesis and the chance to study abroad at the Georgia Institute of Technology. I am very thankful for his feedback on the concepts covered in this thesis.

I would also like to thank Prof. Dr. Umakishore Ramachandran for his support and input in creating this thesis and for the chance to work in his lab as part of the exchange program with the University of Stuttgart. He helped me to focus on the interesting questions in this field and encouraged me to try new approaches to improve the resulting system and the results.

I would also like to thank Dr. Kirak Hong for his extremely helpful and valuable input during the course of this work, giving me advice on the design and implementation of the system and on systems-research in general and for giving feedback on the writeup.

Furthermore I would like to mention and thank Patrick Alt and Fabian Müller for their companionship during our stay in Atlanta.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

1 Introduction

This thesis presents the systems architecture for a distributed graph processing system, focusing on sparse graphs using spatiotemporal data. The source of the spatiotemporal data will be a camera network while the application for the graph processing system will be queries to efficiently retrieve similar events to a given event from the system.

In general, graph processing systems and graph databases have a wide array of potential applications, for example when looking at social relationships between users of a social networking site to infer common interests or potential friends. These systems use graph processing engines such as Pregel or Giraph [MAB+10, Gir12] to do the heavy-lifting of managing a distributed graph, providing an easy to use interface for a programmer. But, as these systems strive to provide a one-size-fits-all solution, there is a lot of potential for optimization when looking at a general-purpose system like Pregel or Giraph and the scenario described in this work.

This thesis will thus focus on an image-similarity based workload in a 3D-graph where the dimensions are the 2D-location (x, y) and time t. Each of the 3D-vertices will be tagged with a feature (for example a face, a car, etc.) For this, in the first step the abstraction of a feature-vertex is being defined which is a four-tuple: It consists of said feature and the 3D-location: $(feature, x, y, t)$. Whenever there is a detection of a specific feature from a source (e.g. a camera being used in a camera-network) the location as well as the time will construct a feature-vertex which will in turn be reported to the graph system. Edges for the graph will be instantiated between feature-vertices having similar features (e.g., determined by using face-recognition) and are spatiotemporally close to one another. The challenge in this domain is the fast growing nature of the 3D-graph, as there are constantly new features being added to the graph. Figure 1.1 gives a high-level overview of this system. Also, an interface to search and explore the feature-graph will be provided: The motivation for this is for example the operator of a camera network who wants to query for similar vertices given a reference-feature within a spatiotemporal bound.

In this thesis, the design and implementation of a distributed graph processing system is being presented which provides for efficient execution of both read and write workloads, that is inserting vertices tagged with detected events and issuing spatiotemporal queries to the system. This task is of latency-sensitive nature as a fast responses to queries is being desired and will be adding a continuous stream of new vertices in rapid succession also, thereby requiring the

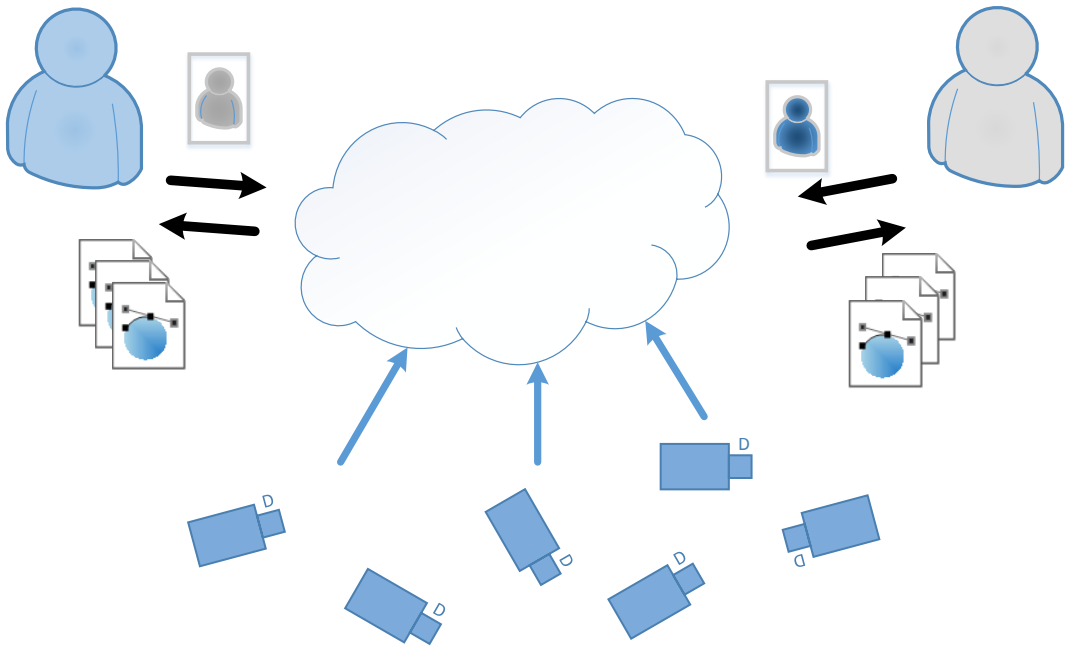


Figure 1.1: A high-level overview of a camera-network and its processing system.

system to also insert vertices fast enough to keep up with the source of the detections, i.e., a camera network.

Two different architectures will be presented: On one hand a master-slave architecture is introduced which relies on a centralized server to process all incoming tasks and distribute them to the workers. On the other hand a distributed architecture is presented which also distributes this function (i.e. processing incoming tasks and assigning these to workers) by using a message-queue. Also multiple graph-partitioning heuristics are being presented which will enable the repartitioning of the graph between the worker-nodes in an online-fashion, i.e., without downtime of the system. The efficiency of the distributed system will be demonstrated and compared against the centralized system with various configurations of worker-nodes. Also a comparable performance in terms of vertex-insertion time for the distributed system shall be shown while employing an online partitioning heuristic (compared to hash partitioning) when inserting new vertices, providing far better edge locality for the vertices.

The rest of this thesis is structured as follows: Related work will be discussed in chapter 2. Chapter 3 will give an overview of the techniques and concepts being used in this work and section 4 describes the different architectures and details needed to understand the system and its execution. Chapter 5 describes the implementation of the system and chapter 6 provides a detailed evaluation of the system. Chapter 7 details potential future work before chapter 8 concludes this thesis.

1.1 Background

This section will give a quick overview on graphs and camera networks which will be used throughout this thesis.

1.1.1 Graph & Graph Processing

A graph is generally defined as an ordered pair:

$$G = (V, E)$$

Herein V is the set of vertices with E being the set of edges with $e = \{v_i, v_j\} \in E,\ v_i, v_j \in V$ comprising an exemplary edge e. This kind of graph is also called an undirected graph, as edges between two vertices, i.e. v_i, v_j, do not have a direction but are either connected in both directions or not at all.

An important metric which will be used throughout this thesis is the degree of a vertex $deg(v)$ which is the number of incoming edges to a specific vertex v:

$$deg(v) = |\{e = \{v_i, v_j\} \in E | v \in e\}|$$

The degree of a vertex will be used to define the term "edge locality" when the graph is being partitioned. The ratio of the degree of a node in terms of local and remote neighbors is being used as the criterion for this:

$$degLocal(v) = |\{e = \{v_i, v_j\} \in E | v \in e \wedge partition(v_i) = partition(v_j)\}|$$

$$edgeLocality(v) = \frac{degLocal(v)}{deg(v)}$$

This metric has the extreme points of 0 (no local neighbors) and 1 (all local neighbors) and will be used further on when discussing heuristics for partitioning the graph to maximize the edge-locality.

1.1.2 Camera Networks

In this work, a camera network will be defined as a set of smart cameras which are connected to a common network covering an area of interest (i.e. an airport terminal or a university campus). These cameras have capabilities which exceed the pure task of recording images but can also perform more complex operations like face detection or target tracking directly on the camera itself instead of handing these tasks of to other computing resources. For the purpose of this system, the camera network is being used as a black-box, reporting detected features (e.g. faces or vehicles with license plates) to the system described in this work. These features will be individually tagged by every camera with the 2D-location and the time of the detection.

2 Related Work

This chapter will give an overview of previous work done in the area of this thesis.

2.1 Graph Processing & Graph Databases

Other graph processing systems that have been developed include Google Pregel [MAB+10] (which hasn't been released to the public but is only being used internally at Google) and Apache Giraph [Gir12], an open-source graph processing system based on the published description of Pregel by Malewicz et al. [MAB+10]. Both of these systems follow a Bulk Synchronous Parallel model, rendering it not a viable solution for the latency-sensitive task at hand [CFSV96], [GV92]. Additionally, both of these systems are not well suited for load-balancing as required by this work out-of-the-box as the partitioning strategy for both of these systems defaults to simple hash partitioning, aiming to evenly distribute the number of vertices between worker-nodes. However, as the system presented in this thesis focuses much more on locality and has a very specific graph-structure and retrieval pattern, a general purpose system like Pregel or Giraph is not perfectly well-suited for this task.

Another downside to both of these full-fledged systems is their reliance on a distributed file system and an underlying execution engine, in the case of Giraph it is the Hadoop-Engine and the Hadoop File System (HDFS) [SKRC10] while Pregel can use either the Google File System [GGL03] or BigTable [CDG+08]. This reliance on a distributed file system will render the system not perfectly well suited for e.g. sparse graphs where most of the computational workload is being generated due to processing on the data that graph is storing and not due to simply traversing and using the graph structure. Thus, the work presented in this thesis opts to using an in-memory approach.

Another graph processing system, similar to Pregel, is GPS (Graph Processing System) published by Salihoglu et al [SW13]. GPS build upon the same primitives as Pregel and is also designed to run on a distributed system of cluster-nodes. Similarly to Giraph, GPS also uses the HDFS to store persistent data and intermediate checkpoints. GPS also provides for custom graph partitioning, but only in an once-and-for-all times manner and not adaptively as the graph is evolving which is especially desirable for a scenario like the one being targeted in this work with a highly dynamic workload.

Another approach is to use a graph database which, instead of a graph processing system, is actually optimized for online processing, examples include Neo4j [Neo15] and OrientDB [Ori15]. But, the issue with these kind of systems is their reliance on replication (or sharding) [Mon13], resulting in potential memory-shortage when scaling to larger workloads. But, this is not only a problem from the perspective of how much memory is being used: When using a replicated system, every change on one partition of the graph has to be reflected in all the nodes connected to the system, even if the change to the graph is only affecting local knowledge (i.e. vertices and edges). As the scenario at hand is a highly dynamic one, it will result in a large number of changes to the local parts of the graph which, in a replicated system, would have to be reflected in the whole system which creates a lot of unneeded overhead, especially when seen from the perspective of only using a single node. This issue, of creating a lot of overhead when running distributed algorithms is also being discussed in [MIM15]. This work by McSherry et al shows that lot of graph systems seem to be scaling well, when just looking at the speed-up when increasing the number of cores, but actually perform poorly when compared to even a simple single-node implementation, running the same algorithms on the same data. The result of this work indicates that one should be careful when adding an extra layer of overhead (e.g. replication) in the presence of alternatives. For the work of this thesis, the same problem is actually not present, as the system presented in this work will not only store the graph and do simple computation on it, but will also execute long-running ($\approx$ a few hundred milliseconds) on each vertex. For a workload like this, the actual computing power has to be increased to allow for faster processing, which in this work will be done by utilizing multiple nodes of a cluster.

There is one more set of graph engines which shall be discussed in here which are temporal graph engines: One such engine is Chronos, presented by Han et al [HML+14]. The system presented in there focuses on answering temporal graph queries, e.g. “How did the PageRank of a given page change over time?”. It presents a batch-processing oriented computation model based on location-awareness, focusing on both time-locality (i.e. two vertices across time are closely connected to one another) and structural locality (i.e. two vertices at the same time-point are closely connected). In comparison to the work presented in this thesis, the kind of computational model presented in Chronos does not provide for the kind of queries asked in the system presented in this thesis due to the unique computational tasks being executed on the vertices themselves (i.e. image-processing).

Other systems with the same kind of restrictions on the the tasks which can be executed on the vertices include GraphChi by Kyrola et al [KBG12] which introduces a *parallel sliding windows* method to process smaller parts of the graph in parallel in order to compute algorithms on a large graph on a single machine. Another set of systems which operate on a streaming-based approach are Kineograph by Cheng et al [CHK+12] and GraphInc by Cai et al. [CLS12]. Both systems present an abstraction on top of a computational model that leans heavily onto the Pregel computational model, the Bulk Synchronous Parallel model. This renders both of these systems as a non-realtime-suited alternative to the system presented in this thesis, which allows insertions and queries in a realtime-fashion.

2.2 Graph Partitioning

The approach employed by Facebook for graph partitioning is to use one of these general purpose systems with a self-defined graph partitioning strategy [PS14]. Facebook uses Apache Giraph to process social media related queries (e.g., finding common interest between friends) and employs the Kernighan-Lin algorithm [KL70] for repartitioning the graph to achieve higher edge-locality while preserving load-balancing. However, this algorithm comes at a cost: the running time of the algorithm of $\mathcal{O}(n^2 * \log n)$ ()n: number of vertices) and the requirement to execute this algorithm between every pair of worker-nodes in the system, i.e., incurring scaling with the square of the number of worker-nodes. Also, the algorithm relies on central coordination of the individual steps of execution. Also, note that the system has to be "offline" when the partitioning is being run, meaning that new queries on the graph can't be executed at the time of repartitioning which renders this solution inviable for a streaming-based, always-on system. Thus using light-weight heuristics to repartition the graph is of great interest, as it can be executed in concurrent with the other functions of the system and doesn't render the system offline for the duration of rebalancing.

Approaches to this include graph balancing based on streaming algorithms for deciding which partition to put a vertex into [TGRV14, SK12]. However, these algorithms are designed for being used in a one-time, startup fashion as compared to a continuous online-fashion. Another approach to online balanced graph partitioning resulting in equal-sized partitions as presented by Rahimian et al. [RPG+13]. However, this algorithm needs more than just local knowledge (it needs a few random samples of the graph at each worker) and it needs multiple iterations to converge which results in locking the system for new queries and insertions of vertices during the time of repartitioning, rendering it infeasible for continuous execution.

2.3 Related Ideas

The idea to use a graph-structure for a camera-network was presented by Xu et al. [XJN+13]. However, the approach in this work is only focused on using the graph for image-processing purposes in a centralized manner, not taking account any scalability issues.

Another approach to the problem of how to find the best possible matches for a given feature is presented by Akdere et al. [AHC13], however this work focuses on tracking objects through multiple live-video streams instead of the storage-retrieve scenario presented in this thesis.

Other state-of-the-art distributed graph processing systems do repartitioning and rebalancing either in an offline fashion [PS14], or assign a vertex to a worker once and for all times [MAB+10, Gir12]. Given the 24×7 nature of situation awareness applications, there is a need to explore online rebalancing strategies in the context of distributed graphs in the presented setting.

3 Concepts

This section will discuss details of the algorithmic approach which are essential for the execution of the system, both in the case of the centralized architecture as well as the distributed architecture.

3.1 Why a Graph?

The system uses a graph as the abstraction for storing the events as it allows for easy traversal of related events given some reference event. It also allows for using the class of well-researched graph algorithms, especially graph partitioning algorithms to enable good load-balancing (e.g. Kernighan-Lin [KL70] or JA-BE-JA [RPG$^+$13] among others) instead of relying on a self-defined load-balancing strategy. Formally, the graph will be defined as $G = (V, E)$ with V being the set of feature-vertices, i.e. events, and E being the relationships between events, with the presence of an edge e, i.e. $e = \{v_1, v_2\} \in E$ depicting that the vertices v_1 and v_2 are spatiotemporally close and their features are similar to one another.

3.2 Graph Partitioning

Multiple heuristical approaches will be used for how to partition the graph in an online-fashion, some of them do not require global knowledge while some will use global knowledge but do not require this information to be consistent at all times (which would be an obvious concern regarding the scalability of the system).

3.2.1 Hashing / Round-Robin

With the baseline approach, vertices are assigned in a round-robin-fashion to workers once and for all times when they are inserted into the system. This allows for evenly balanced partitions but doesn't provide for edge-locality of a vertex. However, it doesn't require any more updates when the system is running, rendering it very cheap in terms of (ongoing) execution time.

3.2.2 Greedy

Whenever the partitioning is being triggered, the greedy strategy will check for all vertices affected by changes that happened since the last partitioning. If a given vertex has more edges to vertices in another worker-node (partition), the greedy strategy will move this vertex to this specific other worker-node.

On one hand, this algorithm can easily be executed in parallel on every worker-node in the system and scales linearly with the number of vertices per worker-node in terms of execution time of the strategy. On the other hand however, this heuristic does not provide for an even load-balancing as there might be a large number of vertices that are being sent to one worker-node in the system to improve edge-locality while rendering this worker-node a hotspot in the system.

The decision criteria for all local vertices, with w_l being the local worker, will therefore be:

$$\max_{w_i} |\{e = \{v_1, v_2\} | v_1 \in w_l, v_2 \in w_i\}|$$

3.2.3 Greedy-Weighted

The greedy method will be extended by balancing the raw number of edges to a specific other worker/partition with the number of vertices already present on that worker. The new decision criteria, with w_l being the local worker, will thereby be:

$$\max_{w_i} \frac{|\{e = \{v_1, v_2\} | v_1 \in w_l, v_2 \in w_i\}|}{|partition(w_i)|}$$

Note that the size of the partition of a specific worker is global knowledge, but doesn't require strong consistency semantics but can also be relaxed to eventual consistency as long as the delay on getting to a consistent state is bounded.

The motivation of this strategy is to not only pick the best match from the perspective of number of neighbors but to favor smaller partitions with comparably high edge-locality as opposed to larger partitions which might have a higher edge-locality in terms of number of neighbors but a lower relative edge-locality. The intuition for this strategy is to favor smaller partitions over larger partitions when they have about an equal number of local edges, thus enabling the smaller partition to grow and allowing a higher relative edge-locality.

3.2.4 Probabilistic

This strategy is comparable in its execution to the greedy strategy, but will not just pick the worker with the highest edge-locality for a given vertex, but will pick proportional to

the number of edges to a given worker. This will lead to a smaller probability of creating hotspots in workers and enables a more evenly balanced partitioning among the different workers/partitions.

Thus, the probability for, e.g. a vertex v_1, to be assigned to the partition of worker w_i is defined as:

$$\Pr(w_i) = \frac{|\{e = \{v_1, v_2\} | v_1 \in w_l \vee v_2 \in w_i\}|}{|E_{w_l}|}$$

In this formula, E_{w_i} denotes the edge-set that has at least one of the vertices assigned to the worker w_i.

3.2.5 Probabilistic-Weighted

This strategy extends the probabilistic strategy with the same idea as the greedy-weighted strategy extended the greedy strategy: Balance the number of vertices in the other partition by the size of the other partition and use a discrete probabilistic distribution, reflecting this strategy, to pick the next worker for a given vertex.

This discrete distribution can be characterized, for the partition of worker w_i, as follows:

$$\Pr(w_i) = \frac{|\{e = \{v_1, v_2\} | v_1 \in w_l \vee v_2 \in w_i\}|}{|partition(w_i)|}$$

This strategy again requires global knowledge in form of the partition sizes but as discussed before this doesn't need strong consistency but will also work with eventual consistency if the delay on reaching a consistent state is bounded, thus providing a viable strategy.

3.2.6 Related Work

Other options for partitioning the graph are for example the Kernighan-Lin-algorithm [KL70] or the JA-BE-JA-algorithm [RPG+13] which both use iterative processing to improve the result of earlier rounds. Note that both these algorithms produce good results in terms of edges crossing between partitions and achieve a perfect load-balancing. However, in this specific problem this work doesn't need to rely on such a heavy-weight partitioning-scheme, as for every round of rebalancing, only a small portion of the graph will actually change, at the point of insertion of the new vertex. This allows the system to only run the partitioning scheme on a few select vertices which in turn renders it cheap to execute and enables an online repartitioning without having to schedule down-time for the system to run the partitioning.

Algorithmus 3.1 The algorithm for the Round-Robin vertex insertion strategy.

```
procedure ASSIGNWORKER(Vertex v)
    v.worker = nextWorker++ % |workers|
end procedure
```

Algorithmus 3.2 The algorithm for the Guided-Insert vertex insertion strategy.

```
procedure ASSIGNWORKER(Vertex v)
    v.worker = spatialIndex.selectNearest(v.spatioTemporalLocation)
end procedure
```

3.3 Worker-Selection Strategies

There are two different modes for inserting vertices:

3.3.1 Round-Robin

Insert vertices in a **round-robin**-fashion which assures that an even balance of vertices is initially given to every worker-node. However, this strategy does not address the issue of maximizing the edge-locality of a given vertex already at the time of insertion.

3.3.2 Guided-Insert

To address the issue of maximizing edge-locality, a second strategy is being proposed , called **GuidedInsert**: In this strategy, a vertex will be assigned to the worker which mean vertex-position most closely resembles the given vertex, therefore maximizing the potential edge locality. While this strategy does not provide for load-balancing, this issue will be addressed in a second level by the vertex-partitioning strategy, which is designed to provide for load-balancing on its own. At the startup of the system, the round-robin scheme is still being run to eliminate problems with having too few vertices per worker rendering a good decision of the `GuidedInsert` strategy unlikely. The algorithm is depicted in algorithm 3.2.

The general process after a worker has been chosen for a given vertex and the vertex has been handed off to this worker is then as follows: The worker-node will first query its local vertices for matches that will result in edges and will then continue to query its neighbors for suitable vertices, which match the spatiotemporal constraints and the feature. The query-type being used for this is a `TimeConeQuery` (see section 3.4), as it is being assumed that an object can only move at a maximum speed and in continuous fashion through the area of interest.

3.3.3 Graphical Comparison between Insertion Strategies

In figure 3.1, both previously presented strategies and their effect on the distribution of the vertices to partitions is being shown. First, in figure 3.1a, the base scenario is being presented, introducing the vertex v_1 which will be inserted into the graph and will be assigned to the partition of either of the workers W_1 or W_2. The scenario employing the Round-Robin strategy is being depicted in figure 3.1b and shows that the vertex v_1 will simply be assigned to the partition of the worker W_2 due to the round-robing-assignment. This creates a naturally balanced partitioning but also induces a low edge-locality as the vertex v_1 now only has remote edges and no local edges. A different scenario can be observed in figure 3.1c when using the Guided-Insert-strategy: Now the constraint on balanced partitions has been given up and v_1 will be assigned to the partition of W_1 due to the proximity of v_1 to the vertices in this partition. This leads to high edge-locality for v_1 at the expense of creating an unbalanced partitioning.

It is important to note that this decision is not being made based on the number of edges a vertex is showing to a respective partition but only on the proximity of a vertex to the mid-point of a partition. This is due to two reasons: On one hand, the worker-selection strategy is designed to be executed in a fast manner, without the need for heavy, latency-prone computation if edges. On the other hand, this shortcoming of the worker-selection will easily be overcome when repartitioning the graph based on one of the partitioning strategies (see section 3.2).

3.4 Supported Queries

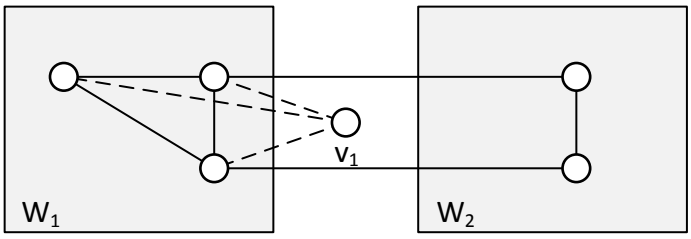
The system supports three different kinds of queries which will be explained in detail in this section.

3.4.1 BoxAreaQuery

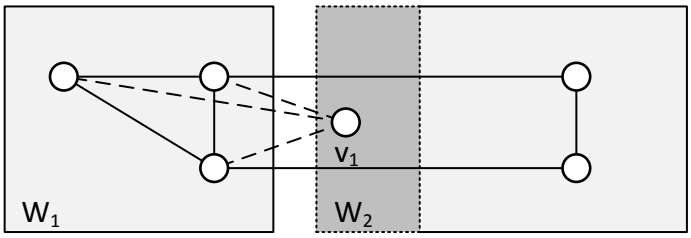
The `BoxAreaQuery` contains the following fields:

- boxCoordinates: an area defined by a box of four spatiotemporal coordinates
- feature: a reference feature to match against

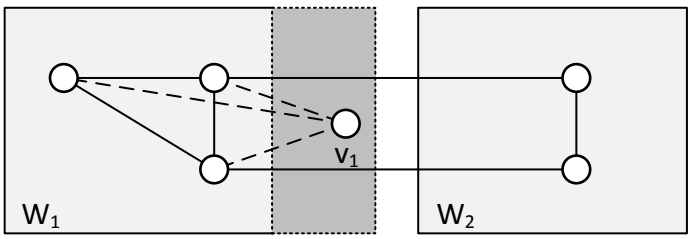
This query will carve a 3D-box out of the graph with the x and y axes being represented by the box given in the query and the z axis being represented by the time-frame of the query. All vertices of the result set have to be similar to the reference image. The algorithm for this kind of query is given in algorithm 3.3.



(a) The base state for the scenario at hand, a new vertex v_1 is being inserted with edges to all three vertices in the partition of worker W_1 which are being depicted with dashes.



(b) This shows the assignment of v_1 to a partition if the Round-Robin-strategy is being used: v_1 is being assigned to the partition of W_2 and a balanced number of vertices are being controlled by each worker.



(c) This shows the assignment of v_2 to a partition if the Guided-Insert strategy is being used: Due to the smaller distance from the vertices in W_1 to v_1, this strategy will assign v_1 to the partition of W_2.

Figure 3.1: The effect of different insertion strategies on the assignment of a vertex to a worker-partition with v_1 being the vertex which is currently being inserted. The new edges from v_1 to other vertices in the graph are depicted as dashed lines.

Algorithmus 3.3 The algorithm for the BoxAreaQuery, as seen from a central perspective without taking the characteristics of the distribution of the graph among the worker-nodes into account.

```
procedure EXECUTE(BoxAreaQuery query)
    Set resultSet
    for all vertices as v do
        if v.matchesTimeBox(query.boxCoordinates) then
            if v.similar(query.feature) then
                resultSet.add(v)
            end if
        end if
    end for
    return resultSet
end procedure
```

3.4.2 TimeConeQuery

The `TimeConeQuery` contains the following fields:

- spatioTemporalCoord: a 3D (x, y, t) starting point
- maxSpeed: the maximum speed of the object of interest
- maxTime: a maximum time bound for a vertex to be considered part of the query-result
- feature: a reference feature to match against

This query is very useful to track the previous detections of an object of interest throughout time as it carves a 3D-cone out of the graph with the time-frame being the z axis again. The rationale of this query being that if one wants to find past occurrences of a specific object which has got a maximum speed, one is only interested in the occurrences of the object in the area it could potentially cover. The algorithm for the query is depicted in algorithm 3.4.

3.4.3 InteractiveQuery

The `InteractiveQuery` has only one field:

- `vertexID`: The id of the vertex to be retrieved by this query, along with all its neighbors.

The semantics of this query are to retrieve the vertex in question itself, with the feature, as well as all of the vertices connected to its outgoing edges all the respective features.

This query is different from the other two query-types as it actually uses the graph-structure in itself for fetching results: It allows for exploring the similarity graph by retrieving a vertex and

Algorithmus 3.4 The algorithm for executing a `TimeConeQuery`.

```
procedure EXECUTE(TimeConeQuery query)
    Set resultSet
    for all vertices as v do
        if v.matchesTimeCone(query.spatioTemporalCoord, query.maxSpeed, query.maxTime)
then
            if v.similar(query.feature) then
                resultSet.add(v)
            end if
        end if
    end for
    return resultSet
end procedure
```

Algorithmus 3.5 The algorithm for executing an `InteractiveQuery`.

```
procedure EXECUTE(InteractiveQuery query)
    if localVertices.exists(query.vertexID) then
        v ← localVertices.get(query.vertexID)
        return v
    else
        worker_new ← hashTable.getHost(query.vertexID)
        send(worker_new, query)
    end if
end procedure
```

its neighbors. The proposed use of this query is to allow an operator to explore the graph, step by step, e.g., to track a target throughout space and time.

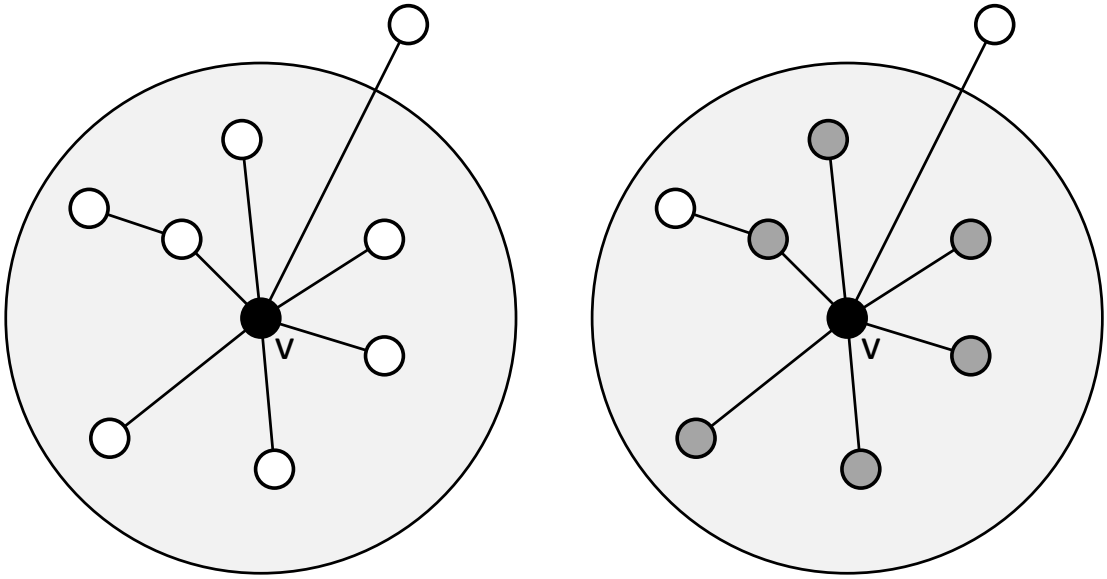
The execution of this query is relatively simple: A worker will simply return its local copy of the vertex in question alongside with all vertices connected via outgoing edges. Those of the vertices that are located on a remote worker will be retrieved from those workers in question as vertices connected via remote edges are stored as remote references, containing only the spatiotemporal coordinates of the vertex and not the feature itself. This task is being carried out by the worker-node which currently owns the vertex that the original query is asking for.

The algorithm for carrying out these actions is depicted in algorithm 3.5.

3.5 Query Optimization

When processing a query the system can also make use of the very nature of the graph structure itself and the semantics of a query: A query submitted by a user is looking for similar vertices as the reference-feature given by the user. The graph structure is build according to these very semantics which enables the system to do the following when it is processing a query: Whenever there is a match between a vertex and the query-contents given by the user, it puts the vertex in question into the result set *as well as* its neighbors (as long as these neighbors match the spatiotemporal bounds set forth by the query).

This can also be seen in figure 3.2 which depicts the situation with and without the graph-optimization being active. In the case of the graph-optimization not being active (figure 3.2a), every vertex inside the target-area (light grey circle) has to be checked individually, in this example this sums up to 8 operations, each taking $50\ ms$ to $100\ ms$. In the case of the graph-optimization being enabled (figure 3.2b), all neighbors of a vertex will be inserted into the result set, if the vertex itself matches. Thus, if v matches the query-criteria, all its 6 neighbors do not have to be checked individually but can directly be included in the result set. This way, only 2 operations have to be executed, as two-hop neighbors won't be directly included in the result-set and thus have to be processed separately.



(a) The state with the graph optimization being disabled.

(b) The state with the graph optimization being enabled. The grey neighbors of v have been included in the result-set and don't need to be checked anymore.

Figure 3.2: The state with and without the graph optimization Vertex v is currently being analyzed and the target-area is depicted as a grey circle. Depicted are the neighbors of v along with the edges connecting those vertices to one another.

4 Architecture

In this chapter the details of the architecture and the associated algorithms will be explained in depth. Two different kinds of architectures will be presented for the problem of organizing and managing the distributed graph of features. On one hand, there is a more traditional master-slave architecture employing a centralized server deciding on the actions to take when inserting a new vertex or querying the actual graph. This architecture will offload the resource-intensive tasks of storing and retrieving feature-vertices to its worker-nodes but will channel all requests through the master-node.
On the other hand, a fully distributed architecture will be presented. This architecture will not rely on a single master-node for executing queries or adding new vertices to the distributed graph but will use every worker-node as a potential master-node for an insertion of a vertex or the execution of a query by using a message queue. The message queue itself becomes the new de-facto entry point for the system and is currently designed as a simple push-based queue and is implemented in a centralized manner. It will be shown that the queue itself does not limit the scalability of the system which will be done by the help of a micro-benchmark of the queue-implementation on its own.

4.1 Messages

As a short reference and to better understand the purpose and functionality of the architectures being presented in this chapter, this section will list the most important messages being used for this section (all other messages along with their complete definition can be found in chapter A in listing A.1). Please note that these messages are not presented entirely in this section but will only be listed with their major and most important components.

- `Vertex`
  This is the basic abstraction for a vertex in the system:
  - `ID`: A string-id of the vertex in question, composed of the id of the worker which was initially responsible for the vertex at hand and a local id to identify different vertices on the same worker.
  - `Feature`: A representation of the feature (e.g. a face) attached to the vertex in question

  - `SpatioTemporalCoordinates`: The 3D-location of the vertex (composed of a 2D-location and time).
  - `Neighbors`: A list of neighboring vertices, identified by their respective ID. This list will be empty during the initial insertion into the system but might carry actual content when moving a vertex from one worker-node to another during the later stages of execution of the system.

- `AddFeatureVertexMessage`
  This message is basically only a wrapper around a vertex-message, which essentially only contains a `Vertex` as an internal field, i.e.:
  - `Vertex`: The vertex which shall be added by the worker receiving this message.
- `PartitioningMessage`
  This message is being used a simple signaling-message to announce to neighboring workers to start the rebalancing algorithm. This message contains the following field:
  - `Vertices`: A list of vertices which were affected by the last insertion. This is useful, as it helps the receiving worker to only run the repartitioning algorithm on the vertices affected (i.e. the vertices neighboring any of the vertices in the set given).
- `QueryResult`
  This message is simply a wrapper around a set of vertices:
  - `Vertices`: A list of vertices which match the original query.
  - `Meta`: A set of attributes to allow the receiving party of this message to identify and match this result-list to their local state.

4.2 Master-Slave Architecture

The Master-Slave architecture implements a network of decentralized worker-nodes while exposing a central interface to the graph-system via the centralized worker-registry. The worker-registry thus functions as the central scheduling unit and is the only point of interaction with any external clients. Communication between two worker-nodes is being implemented by the means of peer-to-peer connections to allow the system to scale and to not overload the central worker-registry while spatiotemporal queries and new feature-vertices are being inserted to the system via the worker-registry. There also exists a broadcast-layer which all worker-nodes are connected to. This broadcast-layer is implemented via a Publish-Subscribe-network. Also, to support the spatiotemporal queries posted to the system, the position and time of each feature-vertex is additionally being inserted into a local database on the specific worker governing a vertex in question to allow for efficient retrieval of feature-vertices matching a spatiotemporal query, employing a spatial index for fast access times. Also, the centralized worker-registry

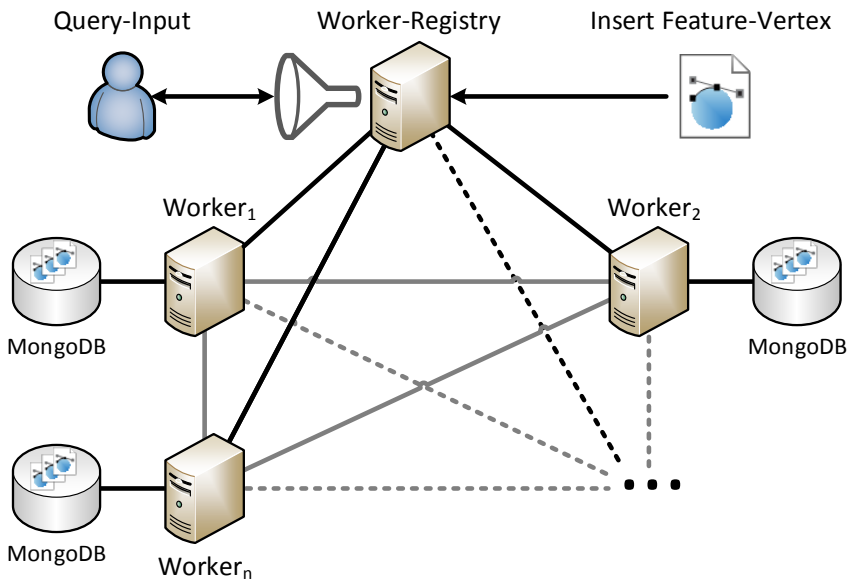


Figure 4.1: A high-level overview of the system architecture of the centralized Master-Slave architecture. Connections between the worker-registry and the worker-nodes are depicted in black while peer-to-peer - connections between worker-nodes are drawn in gray.

keeps track of the areas governed by the respective worker-nodes to only forward a query to the workers inside the area of interest of a interest of the respective query. The area governed by a worker-node is thereby defined as the area covering all the vertices of the respective worker. As this would actually be a complex 3D-volume which would be difficult to compute, the system will keep track of time-sliced versions of the areas the workers are governing through their association to their current vertices. An overview of the Master-Slave architecture is given in figure 4.1.

Note that the actual computational workload of processing the feature-set and retrieving results from the graph is still being carried out in a distributed fashion on all the worker-nodes in the system, only the entry point to the system is realized as a centralized solution and has to carry out the tasks of forwarding the insert- and query-messages to the worker-nodes responsible for this task. However, the worker-registry also has to carry out all lookups in its spatial index for determining which workers to forward a query to according to their governed areas. This observation motivates the design of a fully distributed design where this work is being carried out in a distributed fashion.

In the Master-Slave architecture, a vertex-partitioning scheme (see section 3.2 for more details on this) to determine and potentially reassign the best worker-node in terms of edge-locality for a specific vertex is being run after every insertion of a vertex on the part of the graph which

Algorithmus 4.1 The algorithm running on the master-node (the worker-registry) for inserting a vertex in the Master-Slave architecture.

```
procedure ONRECEIVE(AddFeatureVertexMessage message)
    w_i ← GETRESPONSIBLEWORKER(message)
    id ← GENID(message)
    message.vertex.id ← id
    vertexToWorker.store([id, w_i])
    SEND(w_i, message)
end procedure
procedure RESPONSIBLEWORKER(AddFeatureVertexMessage message)
    if mode = RoundRobin then
        return nextWorker++ % |workers|
    else if mode = GuidedInsert then
        return spatialIndex.findNearestWorker(message.spatioTemporalCoordinates)
    end if
end procedure
procedure GENID(AddFeatureVertexMessage message)
    return localID++
end procedure
```

is actually affected by the insertion of the vertex. This part of the graph are usually these vertices which are the neighbors of the recently inserted vertex.

All these actions can also be seen in algorithmic form: Algorithm 4.1 gives an overview of the actions the master-node of the architecture is carrying out, before handing the newly inserted vertex off to a suitable worker-node. In algorithm 4.2 the sequence of action from the point of view of the worker-nodes is being depicted, which mostly revolves around finding edges for the vertex in question and running the partitioning algorithm. Algorithm 4.3 details the process of finding remote edges, for both the Master-Slave architecture and the Distributed Masters architecture.

4.3 Distributed Masters Architecture

The reliance of the Master-Slave architecture on a centralized master raises obvious scalability concerns. Thus, a fully distributed architecture which does not rely on a centralized component to carry out the workload, i.e., adding vertices to the worker-nodes and retrieving them, shall now be presented.

The entry-point to this distributed system is a message-queue taking as input tasks (to-be-added vertices or queries for the system) and dispatching them to a *ready* worker-node (see section

Algorithmus 4.2 The algorithm running on the worker-nodes for inserting a vertex in the Master-Slave architecture.

```
procedure ONRECEIVE(AddFeatureVertexMessage message)
    v ← addToLocal(message.vertex)          // This returns a reference to the local vertex
    GENERATELOCALEDGES(v) // Simply runs a local search for neighbors among all possible
neighbors
    FINDREMOTEEDGES(v)
    RUNPARTITIONING(v)
end procedure
procedure RUNPARTITIONING(Vertex v)
    message ← PartitioningMessage(v.neighbors)
    BROADCAST(message)
end procedure
procedure ONRECEIVE(PartitioningMessage message)
    vertices ← selectLocalVertices(message.vertices)
    for all vertices as v do
        RUNPARTITIONINGSTRATEGY(v)
    end for
end procedure
```

Algorithmus 4.3 The algorithm for finding remote edges for a given vertex. The function INSERTEDGE will simply ensure that both vertices are present at the worker in question (potentially by the means of a remote-reference) and will then set up the edge in both directions. The send-call to the "master"-node refers to either the centralized master in the Master-Slave architecture or a distributed master-node in the Distributed Masters architecture.

```
procedure FINDREMOTEEGGES(Vertex v)
    query ← TimeConeQuery(v.location, maximumTimeDifference = 30 s, speed = s m/s)
    SEND(master, query)
end procedure
procedure ONRECEIVE(TimeConeQueryResult result)
    for all result.vertices as v2 do
        INSERTEDGE(v, v2)
    end for
end procedure
```

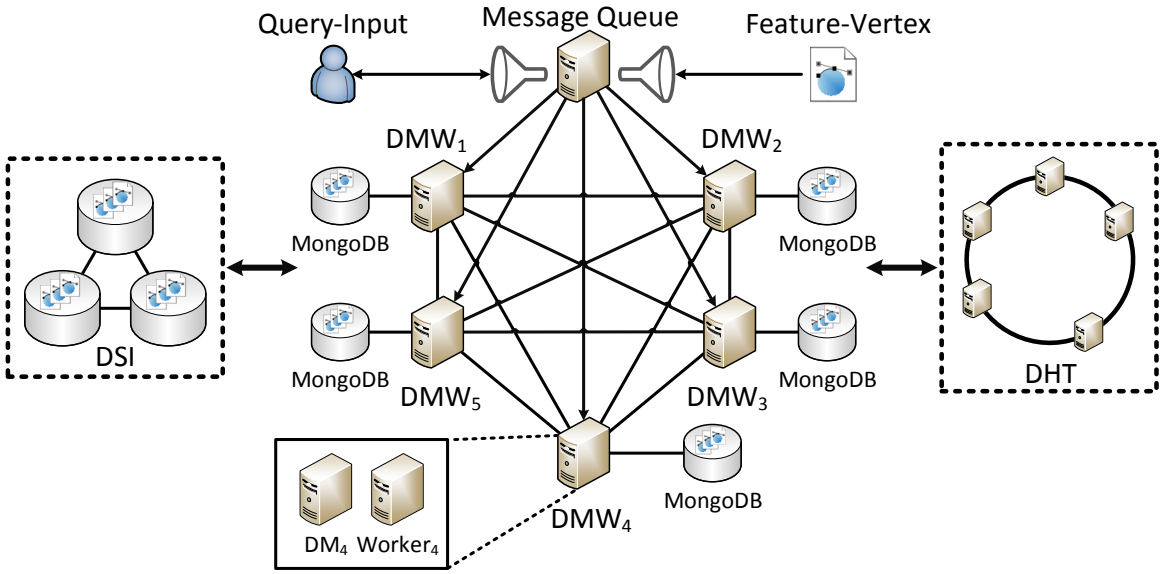


Figure 4.2: A high-level overview of the system architecture for the fully distributed architecture. One node can act as both a worker for executing queries or storing information as well as a distributed master-node (as depicted by DMW_4: **D**istributed-**M**aster & **W**orker), executing the tasks given to it from the message-queue.

4.3.1 for more details on how the queue is being designed). Whenever a worker-node receives a new task from the message-queue, it will act as a master-node for this specific task, be it inserting a vertex or executing a query. As a master-node, the worker has to distribute queries to the appropriate worker-nodes on the system or find the right worker to put a vertex onto. Every worker-node can act as the master-node for a query or a new vertex and will only be executing one of these tasks at a time and will report as *ready* to the queue only after finishing the specific task.

To enable the workers to do these tasks, there exists a distributed spatial index (DSI), to gather the knowledge of which worker governs which area at a specific point in time and what its mean vertex is. To efficiently store the mapping of which worker-node is currently responsible for which vertex, there also exists a distributed hash table (the mapping thereby consists of `vertexID` $\Rightarrow$ `workerID`) as the vertices might be handed over from one worker to another one, e.g. to improve the edge locality. Each of the workers has got a direct peer-to-peer connection to all the other workers to facilitate directed communication between two workers. In addition to the direct peer-to-peer connections between the different worker-nodes, there also exists a pub-sub-network to which every worker subscribes on start-up. This serves the purpose of signaling the insertion of a vertex on one specific node to all other nodes to give those nodes a chance to run the selected partitioning scheme, if needed (see section 3.2 for an overview of the partitioning schemes). An overview of this architecture is given in figure 4.2.

Algorithmus 4.4 The algorithm running on a distributed master-node for inserting a vertex in the Distributed Masters architecture.

```
procedure ONRECEIVE(AddFeatureVertexMessage message)
    w_i ← GETRESPONSIBLEWORKER(message)
    id ← GENID(message)
    message.vertex.id ← id
    DHT.insert([id, w_i])
    SEND(w_i, message)
    BROADCAST(workerInserted)          // This gives all workers the chance to increase the
nextWorker-count
end procedure
procedure RESPONSIBLEWORKER(AddFeatureVertexMessage message)
    if mode = RoundRobin then
        return nextWorker % |workers|
    else if mode = GuidedInsert then
        return distributedSpatialIndex.
                findNearestWorker(message.vertex.spatioTemporalCoordinates)
    end if
end procedure
procedure GENID(AddFeatureVertexMessage message)
    return CONCATENATE(workerID, localID++)
end procedure
```

The algorithms for the Distributed Masters architecture are fundamentally equal to the Master-Slave architecture, but use the distributed components where applicable. The algorithm for the insertion at a distributed master-node is given in algorithm 4.4 while the algorithm for the insertion of a vertex is shown in algorithm 4.5. Note that the algorithm for the worker-node is virtually unchanged with the exception of finding the remote-edges, which will route the query-message to the queue first for distribution among the set of distributed master-nodes.

4.3.1 Queue

As part of the distributed architecture a messaging queue is being used, basically to multiplex requests to the various worker-nodes acting as a distributed master-node.

The queue itself has been designed specifically for the task at hand and is based around the principle of a push-based service. In contrast to a regular, pull-based service where the nodes periodically poll the queue for new messages, a different model is being applied in this work: Every worker will report when it is ready to take a new message from the queue and will block locally until the queue will actually send a new message to the worker, reducing the overhead at the worker-node. A broad overview of the queue-design can be seen in figure 4.3. As can

Algorithmus 4.5 The algorithm running on the worker-nodes for inserting a vertex in the Distributed Masters architecture.

```
procedure ONRECEIVE(AddFeatureVertexMessage message)
    v ← addToLocal(message.vertex)          // This returns a reference to the local vertex
    GENERATELOCALEDGES(v) // Simply runs a local search for neighbors among all possible neighbors
    FINDREMOTEEDGES(v)
    RUNPARTITIONING(v)
end procedure
procedure RUNPARTITIONING(Vertex v)
    message ← PartitioningMessage(v.neighbors)
    BROADCAST(message)
end procedure
procedure ONRECEIVE(PartitioningMessage message)
    vertices ← selectLocalVertices(message.vertices)
    for all vertices as v do
        RUNPARTITIONINGSTRATEGY(v)
    end for
end procedure
```

be seen in the picture, the queue itself only acts as a proxy for forwarding messages to the individual queues which will in turn handle the actual workload on distributing the messages. The algorithm for the `QueueServer` itself can be seen in algorithm 4.6, this component takes care of managing the individual queues while being available over the network via a message-based-interface. The algorithm for a single queue itself can then be seen in algorithm 4.7.

The performance of the queue itself is a crucial part of allowing the Distributed Masters architecture to scale well as the queue-operations are on the critical path, the message queue thus has to support low latency and high throughput. As such, it is of great importance to carefully evaluate whether the queue-design does in fact support thousands of message per second with minimal delay. The results of this evaluation can be found in section 6.5.1.

4.4 Sequence of actions while inserting a vertex

To understand the sequence of actions in both the centralized Master-Slave architecture and the Distributed Masters architecture, a closer look at the insertion of one vertex shall be taken.

The general schematics of an insertion in the Master-Slave architecture is depicted in figure 4.4. The message is first being submitted to the worker-registry: In the Master-Slave architecture,

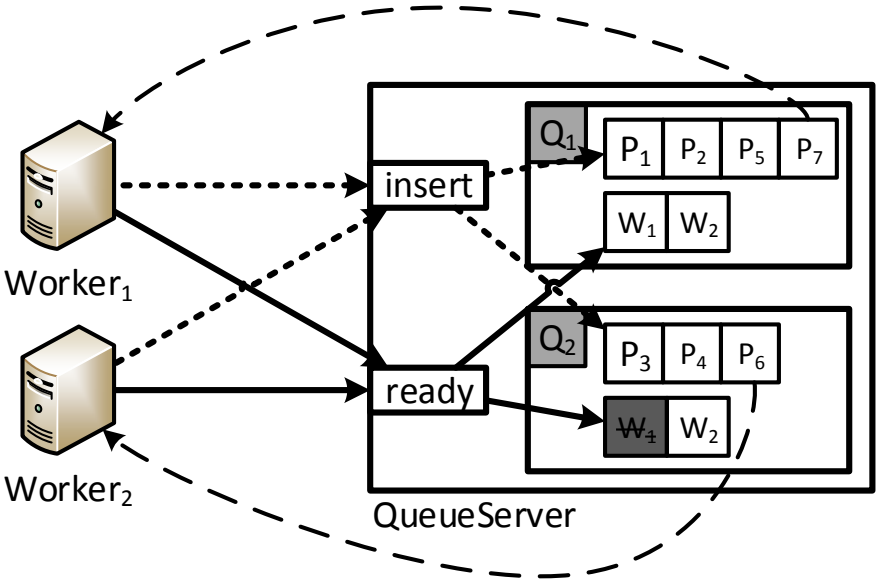


Figure 4.3: A detailed overview of the queue-architecture. In this example, the `QueueServer` contains two queues, Q_1 and $Q2$. For Q_1, both workers W_1 and W_2 are available for new messages (depicted as payload, i.e. P_i) while for Q_2 only the worker W_2 is available for new messages. The `QueueServer` acts as proxy for forwarding requests to the individual queues.

Algorithmus 4.6 The algorithm of the push-based queue for the `QueueServer` which is exposed to clients using the queue via a message-based-interface.

procedure REGISTER($worker$)
    $worker$.openSocket()
    workerList.push($worker$)
end procedure
procedure READY($worker$, $queue_i$)
    $queue_i$.ready($worker$)
end procedure
procedure INSERT($message$, $queue_i$)
    $queue_i$.insert($message$)
end procedure

Algorithmus 4.7 The algorithm of the push-based queue for the `SingleQueue` which implements the actual message- and worker-handling. The procedure `run` is being executed as a thread.

```
procedure READY(worker)
    readyList.insert(worker)
    readySemaphore.notify
end procedure
procedure INSERT(message)
    messageList.insert(message)
    messageSemaphore.notify
end procedure
procedure RUN
    while alive do
        messageSemaphore.wait
        nextMessage = messageList.popFirst
        readySemaphore.wait
        worker = readyList.popFirst
        worker.send(nextMessage)
    end while
end procedure
```

the worker-registry can do all the tasks necessary to fulfill the task of inserting a vertex (e.g. selecting a good worker etc.) directly, locally, at the point of entrance to the system. The message is then being routed to a worker suitable for inserting the vertex at hand (i.e. W_k), which in turn will query its neighbors (after looking them up in the spatial index at the worker-registry) for similar vertices to build the edges of the newly inserted vertex. After doing so, the worker will, if necessary, update its coverage-area at the worker-registry and send an `insertSucessful`-message to the original node inserting the vertex to end the insertion process.

A very similar course of actions is being followed in the Distributed Masters-architecture and is depicted in figure 4.5. For this architecture, the following course of actions is taken: From the camera network, the message is being routed to the message-queue, which in turn elects one worker-node, in this case W_i as the master-node for this particular task. W_i will then do the same tasks the regular master-node would have done, querying the distributed spatial index for the best matching worker-node for the particular coordinates of the vertex and handing off the vertex to this worker, here W_k. This worker will then query its neighbors for similar vertices, add its local and remote edges, insert the mapping of the given vertex to the worker W_k into the distributed hash table and updates the distributed spatial index with the new set of bounds and a new mean vertex (if applicable). In the end, a `success`-message is being transmitted back to the node that the request originated from.

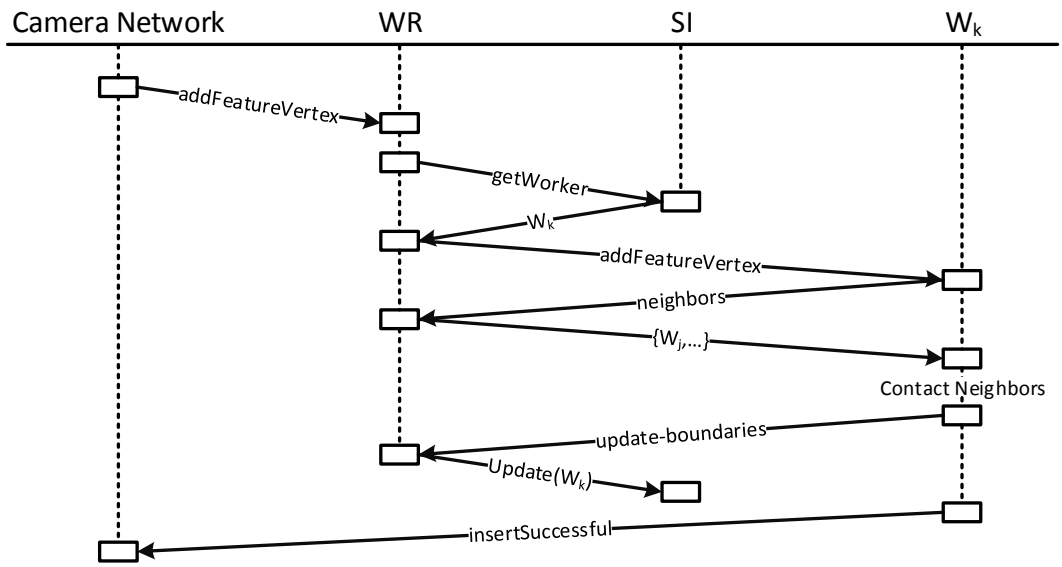


Figure 4.4: A detailed overview of the messages being sent and processed during an insert-operation of a vertex in the Master-Slave architecture. The nodes in the network participating in this examples are depicted on the top, where WR is the worker-registry, W_k is the worker responsible for the vertex and SI is the spatial index.

Thus, the main difference between the message-flow in both architecture is the additional step of submitting the message to the queue and finding a suitable distributed-master-worker (e.g. W_i in this example) in the case of the Distributed Masters architecture. Also, the distributed hash table or the distributed spatial index (as opposed to the local spatial index that is being used in the setting of the Master-Slave architecture) are missing in the Master-Slave architecture as all the information is being stored at the centralized worker-registry, i.e. the master-node.

4.5 Executing a query

When a new query is being received by a worker-node, it will look up the vertices that match the spatiotemporal constraints set forth by the query and will then perform the similarity function. If there is a similarity-match, the respective vertex is being put into the result set. One optimization which can be used based on the graph structure is the following: If there is a match between a query and a vertex based on the image-similarity there is a high potential for the endpoints of all outgoing edges to also match the query. This is a direct result from the construction of the edges based on similarity between the features in question. However, still all vertices in question have to be checked for whether or not the corresponding vertex matches the spatiotemporal constraints. If so, they will be included in the result set and will be removed

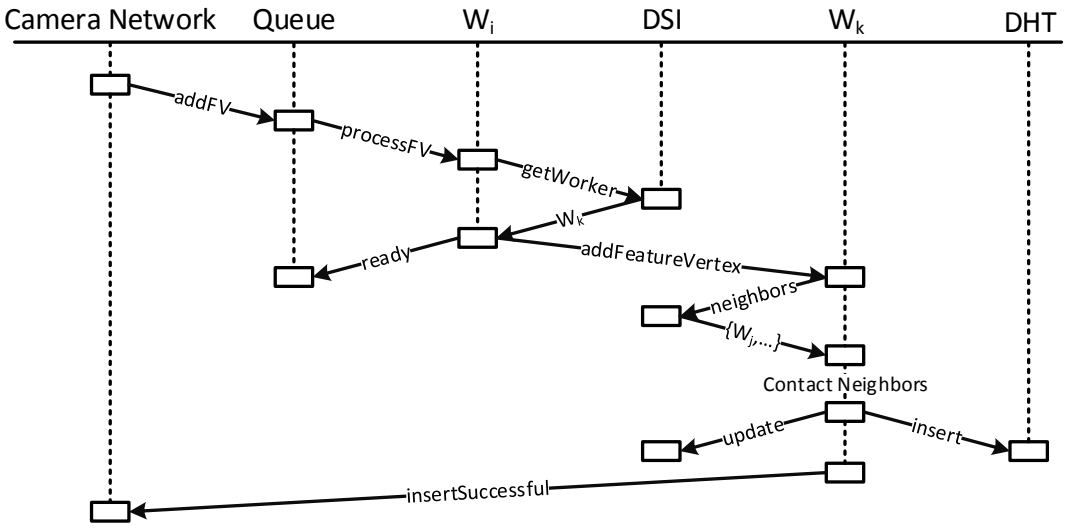


Figure 4.5: A detailed overview of the messages being sent and processed during an insert-operation of a vertex in the Distributed Masters architecture. The nodes in the network participating in this examples are depicted on the top, where W_i and W_k are two workers, DSI is the distributed spatial index and DHT is the distributed hash table.

from further consideration for the specific query, thus circumventing the execution of the costly similarity-analysis. Also, see section 3.5 for more details on this kind of optimization.

The general flow of messages and actions that follow from that in the different architectures will now be discussed. The general course of actions for the Master-Slave architecture can be seen in figure 4.6. From the user posting the query to the system, the query-message gets routed to the worker-registry. It will look up the workers which are responsible for the area in question and will then hand off the query to each of the workers (e.g. in this example W_i and W_j), those workers will process the query based on their local knowledge and will then report the result back to the worker-registry which will in turn hand over the result-set to the user, initially posting the query, after all responses have been received.

The message-diagram for the Distributed Masters architecture is depicted in figure 4.7. In place of the centralized worker-registry as the entry point to the system as in case of the Master-Slave architecture there is the message-queue. It will distribute the query-message to a distributed-master node (in this case W_i) which will act as the master for this query and execute all tasks just as the worker-registry does in the Master-Slave architecture. Once again, a distributed spatial index is now being used to query the workers affected by this query. The query itself will then be handed off to the workers in question (in this example, W_j and W_k) which will then locally process the query and return the result to the distributed-worker node. On receipt of the last query-result, the distributed-worker node will return the query-result-set

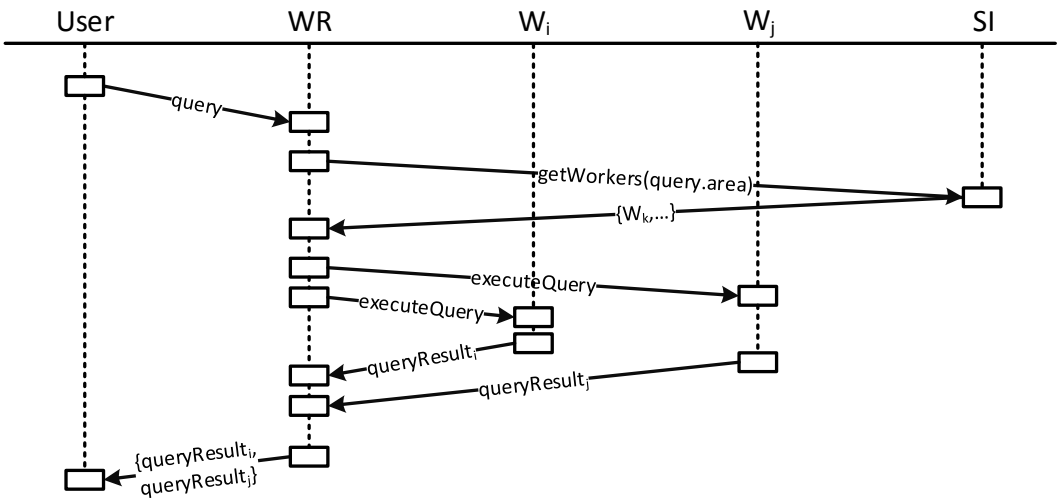


Figure 4.6: A detailed overview of the messages being sent and processed during an query-operation in the Master-Slave architecture. The nodes in the network participating in this examples are depicted on the top, where W_i and W_k are two workers, DSI is the distributed spatial index and DHT is the distributed hash table.

to the user posting the query and will report `ready` for new queries to the queue (see section 4.3.1 for details on how the queue interface operates).

4.6 Consistency Requirements

This section will now discuss the consistency requirements for the various operations in the network, thus discussing the opportunity to parallelize multiple concurrent operations in the system, trading strong consistency for performance.

4.6.1 Master-Slave Architecture

First, the focus will be on the Master-Slave architecture: For this system-layout, one can notice that most operations, e.g. the mapping of vertices to workers and the areas and mean-locations of workers, are consistent by the very point of being handled in a centralized manner. But, the graph itself is only eventually consistent, as the graph will for some time during the insertion not contain all edges that are actually in the graph: When inserting a remote edge, this edge will at first only be inserted on the local worker which executes the query associated with the insertion first (see figure 4.5 for details) before sending the response containing the remote-vertex of the remote edge back to the host currently responsible for the vertex. But, this kind of non-consistency (when seen in a global sense) is not a problem, as remote edges will

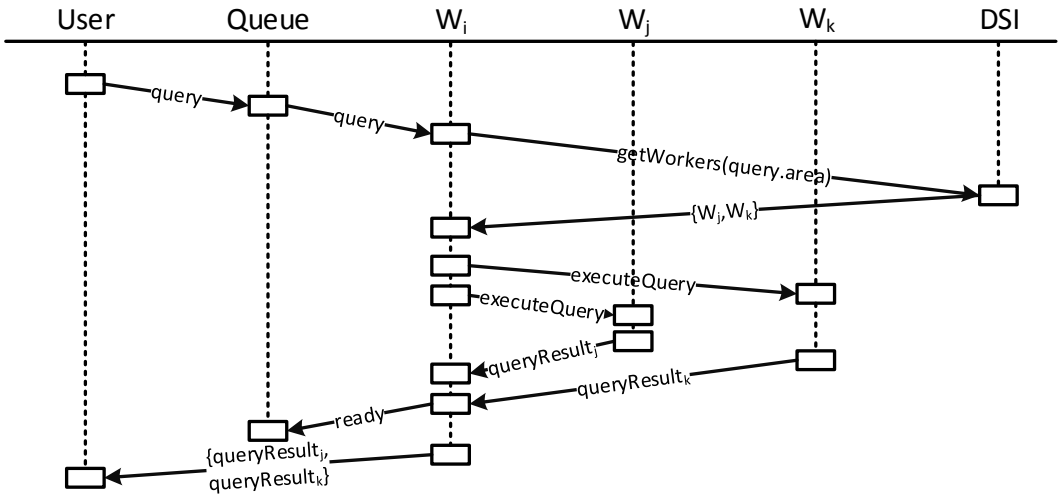


Figure 4.7: A detailed overview of the messages being sent and processed during an query-operation in the Distributed Masters architecture. The nodes in the network participating in this examples are depicted on the top, where W_i and W_k are two workers, DSI is the distributed spatial index and DHT is the distributed hash table.

only be used for rebalancing purposes or for fulfilling `InteractiveQueries` and both operations will still work, even though the underlying graph structure is not strongly consistent.

4.6.2 Distributed Masters Architecture

In the case of the Distributed Masters architecture, the two functions of mapping a vertex to its worker and storing the current area and the mean-location of the workers has to be handled in a distributed manner. The mapping of a vertex-id to the current worker is being handled using a distributed hash table while storing the areas and mean-locations is being made accessible using a distributed spatial index. This in turn means that the consistency implications and requirements for these components have to be discussed as well.

- **Vertex to Worker-Mapping**: For this function, the system can allow eventual consistency. The only part where this functionality is critical is an `InteractiveQuery` (see section 3.4.3 for details). But, as depicted in algorithm 3.5, if the vertex in question is not found in the local data-store, it will be forwarded to the worker actually in charge of the vertex to fulfill the query, allowing the system to rely on eventual consistency.
- **Distributed Spatial Index**: For this component, the system can also allow eventual consistency, at least for the location of the mean-vertex for every worker: Even though this information might change frequently, it won't change by a large margin in most cases and even if it changes by a larger amount, it will still not render a problem as this information is mainly being used for the insertion-mode to find a suitable worker that is

close to the vertex which is being inserted. But, even if the “wrong” decision is being made due to stale data, this still won’t render a problem as this will get taken care of during the rebalancing/repartitioning phase. But, the areas of the respective workers require more of a strongly consistent update, as stale information in this component might lead to a query not being fulfilled in some circumstances (e.g. the area of governance of a worker has not been adjusted to cover a newly inserted vertex yet, a query for the area of the vertex in question might thus not be answered correctly.). But, this consistency-requirement is also mitigated by the fact that the area of governance of a worker will only change very infrequently (in a steady-state system), thus allowing the system to not rely on a strongly consistent operation during every insertion, which would render the system potentially inefficient.

5 Implementation

This chapter will give an overview of the actual implementation of the system which has been described in detail in chapter 4. This section will first give an overview of the different kind of technologies being used in this systems, also focusing on the strategies and patterns used in this work before going into detail on the actual implementation.

5.1 Technology Overview

The system presented work has been implemented as a prototype in C++. Coupling of the various modules is being done via message-passing using Google Protobuf [Goo08] as the serialization-solution as well as ∅MQ[Zer12] as the networking-layer. This choice allows for easy access to the system from software-solutions based on other languages (e.g. Java, Python, ...) as both the messaging as well as the serialization framework are available for a large set of languages, allowing interoperability between systems based on a wide array of languages.

5.1.1 Network, Messages & Serialization

The broadcast-layer that was introduced in section 4.3 is being implemented by the means of pub-sub-network using ∅MQ(often written as ZeroMQ). This kind of network allows for easy multicast-support and thus also allows for a straightforward application-layer broadcast-support. Image processing and analysis is done using the OpenCV-library [BK08], with the task of face-recognition being carried out by Fisherface-algorithm [BHK97] which showed the best recognition-rates for the given sample scenario when compared with other, relatively simple face-recognition algorithms. A downside to the Fisherface-algorithm is its dependence on training data to accurately recognize faces (i.e. assign labels to faces), there are other algorithms that operate on less or no training data and achieve similar accuracy, but were not considered in this work due to their complexity, as for example the work of Zhuang et al [ZYZ+13]. The database, described in section 4 which is being used to store location and time for each feature-vertex for easier query-retrieval is a MongoDB. To enable fast querying, a compound index consisting of the location and the time for each feature-vertex is being used inside the respective MongoDB-instances.

Algorithmus 5.1 An example for non-blocking message-based communication. The procedure RUN is being executed as a separate thread. The queue q is being implemented in a thread-safe manner.

```
Queue q
procedure ONRECEIVE(Message message)
    q.push_back(message)
    q.notify()
end procedure
procedure RUN
    while true do
        q.wait()
        message ← q.pop_front()
        HANDLEMESSAGE(message)
    end while
end procedure
```

To support a low-latency and high-bandwidth oriented system design, all communication happening in the system has been implemented in a non-blocking fashion: messages that have been received in a node will be put in a processing-queue, allowing the receiving function to immediately acknowledge the receipt of the message without having to wait for the message to be processed. Also, the processing of these messages is being done in an event-based manner, ensuring efficient usage of the given system resources.

5.1.2 Data Storage

The distributed spatial index has been implemented using a sharded collection in MongoDB, distributing the dataset among a set of client-instances. This allows for scalability in terms of storing all mean-points and areas of the different workers among the cluster. Unfortunately, due to restrictions with MongoDB which doesn't support sharding according to the geospatial data, also called geosharding, an artificial sharding key and a second-level geospatial index for actually accessing the results has to be used [1]. But, even though the system can't make use of this feature, it will still ensure that every instance of MongoDB in the sharded setup will get an equal share of workload by using uniformly distributed random keys as the sharding key.

For storing the association of which vertex resides on which worker, a distributed hash table is being employed. The implementation chosen to do this is a zero-hop hash-table (ZHT), a flavor of hash tables specifically optimized for high-performance computing by taking into account

[1] see http://docs.mongodb.org/manual/applications/geospatial-indexes/#geospatial-indexes-and-sharding and https://jira.mongodb.org/browse/SERVER-926 for the actual bug-entry.

that these kind of environments see considerable less amount of churn than a distributed hash table deployed in a general setting, thereby sacrificing on this aspect in exchange for a $\mathcal{O}(1)$ lookup-time for a given key [LZB+13].

Also, to enable a quick and efficient access method for a worker to its local set of vertices, given a spatiotemporal bound (e.g. as given by a query), there exists a local spatial index, containing all vertices known to a worker. This index stores for every vertex their spatiotemporal location, (x, y, t) and a flag which indicates whether the specific vertex is a local or remote vertex.

5.2 Components

In this section, the individual components, as introduced in the architecture-overview in chapter 4 will be explained in detail as well as their actual class-layout will be shown.

5.2.1 Worker-Registry

The worker-registry acts as multiple roles, depending on which architecture is being chosen: In the Master-Slave architecture, the worker-registry will play the fundamental role of distributing all tasks to the various worker-nodes (the slaves) connected to the system. It will also serve as the single entry-point to the graph system. On the other hand, in the Distributed Masters architecture, the worker-registry is merely being used for bootstrapping purposes, to distribute the settings to each participating worker and to enable all workers to discover one another, or for statistics-purposes, relaying the requests for statistics-collections to all workers. The main components for the worker-registry are:

- `AddFeatureVertexProcessor`: It receives an `AddFeatureVertexMessage` which contains a feature-vertex and related meta-data. The steps taken during the handling of this message can also be seen in figure 4.4, they mainly involve finding the workers responsible for the area of interest for the current vertex and running the selection scheme to determine which worker shall be handed the vertex in question (see section 3.3 for more details on the available selection algorithms).
- `QueryMessageProcessor`: This component dispatches an incoming `QueryMessage` to the appropriate workers in the system. The detailed process for this operation can also be seen in figure 4.6 and shows that the worker-registry functions as the master for this query and will interact with the spatial index (to retrieve the set of workers which are responsible for the query at hand) as well as storing the partial answers from the workers until all answers have arrived and can be shipped to the party originally posting the query.

- `QueryResponseProcessor`: The component responsible for accepting the responses from the workers when executing a query.

5.2.2 Worker

The worker itself is arguably the most important component, as it sits at the heart of the system and has to process all tasks on the graph and store its local representation of the graph. Internally, the worker-structure uses a number of concurrently running, so called *processors*, to handle interactions with incoming messages. The most important processors are the following:

- `AddFeatureVertexProcessor`: It receives an `AddFeatureVertexMessage` which contains a feature-vertex and related meta-data. The actions taken by the worker on receipt of this message can also be seen in figure 4.5 and involve querying for similar and spatiotemporally close vertices on the neighbors. This allows setting up the edges as remote references on the neighbors as well as the worker. The next step will then involve running the rebalancing algorithm, striving for high edge locality.
- `RebalancingProcessor`: This processor is being notified on the receipt of a rebalancing-request and will trigger the chosen vertex-partitioning algorithm (see section 3.2 for more details on the available algorithms). It will then move the vertices, and their internal state such as the set of edges and the feature, to the new worker. Note that it is important to include the set of edges (local and remote) in the vertex when migrating it as the new worker only has a partial view of the edge-set of the vertex in question and e.g. doesn't store the formerly local edges of the vertex in question.
- `InteractiveQueryProcessor`: The purpose of this processor is to answer incoming `InteractiveQueries` (see section 3.4 for details on the query-types). It will simply look up the vertex in question in its local data-store and return it, if available. Should the vertex not be available in the local data-store, because it has been migrated in the mean-time, this processor will forward the query to the new host of the vertex which will also do the same sequence of actions. This might then result in further forwarding-steps if the vertex in question is being migrated in a rapid fashion but will eventually converge.
- `SpatioTemporalQueryProcessor`: This processor will execute an incoming `SpatioTemporalQuery` by simply looking up the vertices matching the spatiotemporal bound set forth by the query in the local spatial index. For all vertices matching the spatiotemporal bound, the image-similarity algorithm (i.e. face recognition using the Fisherfaces-algorithm [BHK97]) will be run to match the faces. All matching vertices will then be returned in a `QueryResponse` to the master of the query in question (see figure 4.6 for a sample).

- `TimeConeQueryProcessor`: This processor will execute incoming `TimeConeQueries`, in a similar fashion to the `SpatioTemporalQueryProcessor` but using a 3D-cone to carve out the results instead of a simple box as the `SpatioTemporalQueryProcessor`. The intention for this is the limited maximum distance an object of interest can have (due to a maximum moving speed) when starting out from a known spatiotemporal location. The comparison-process is the same as for a `SpatioTemporalQuery`, besides the change to the area of interest, and will also involve the image-similarity computation. The result will then, once again, be returned to the master of the current query.
- `MoveFeatureVertexProcessor`: This processor is responsible for simply inserting a migrated feature-vertex into the local data-store while taking care of creating all local and remote edges for the vertex in questions, potentially creating new remote references as needed.
- `QueryResponseProcessor`: This processor will receive income responses to queries posted by the worker it is running on. This component usually comes into play during a vertex-insertion when a worker is searching for potential remote edges by querying the system using a `TimeConeQuery` (see section 3.4 for more details on the available queries in the system).
- `DistributedMasterQueryResponseMessageProcessor`: This processor is only active when the system is running in the Distributed Masters architecture. It is being used to collect responses from all workers being involved in the query in question. It will collect all responses and upon receipt of the last response, the aggregated response-set will be sent back to the original party posting the query to the system.
- `QueueTaskProcessor`: This processor is only active when running with the Distributed Masters architecture. It serves the purpose of interacting with the queue (see section 4.3.1 for details on the queue-design and the interface) and will execute new tasks which are being handed to the worker from the queue.
- `ZeroMQPullProcessor`: This processor serves the basic purpose of listening on the ingress-socket of the worker and is the main entry-point to the worker. After receiving the raw byte-message, it will then use metadata-information contained in the bytestream to decode the actual message stored in the stream. After decoding the message, a `handleMessage`-method in the worker will be called. This method will multiplex the message into the appropriate processor as described above.

5.2.3 Queue

A critical component for the Distributed Masters architecture (also see section 4.3 for more details) is the message-queue. This component will carry the burden of distributing messages to the subscribed and *ready* workers as fast as possible to allow the distributed system to scale

beyond the potential limitations of the centralized Master-Slave approach (see section 4.2 for details).

The individual, logical units of the Queue-component are:

- `SingleQueue`: This unit is responsible for a single queue. It will store newly submitted messages in a local queue and hand them off to a worker which reported that it is ready for the queue in question. This can either happen immediately, when there is already a worker waiting for new tasks, or it might happen asynchronously whenever a new worker becomes available for the queue in question.
- `QueueServer`: This unit is responsible for fielding incoming new tasks and newly active workers. It acts as a multiplexer between the different queues and will simply forward messages to the appropriate queue. It is also responsible for maintaining the streams to the respective workers for transmitting new message without the need to open new connections for each message.

6 Evaluation

This section will first present the setup, methodology and various metrics and variables used in the evaluation of the presented architecture before evaluation the various aspects of the system, paying particular attention to how well the different architectures and algorithms perform when the problem size and the number of worker-nodes is being increased. The partitioning strategies will be compared against a traditional *hash* partitioning and are expected to at least show the same performance as the *hashing* strategy while providing for better edge-locality.

6.1 Overview

First, an overview shall be given what to expect in this evaluation and how to put the results into context. Even though the sizes of the graphs seem to be somewhat limited (up to $\approx$ 3000 vertices), the number of vertices in the graph is actually not the main cost-driver: To support the queries (presented in section 3.4), a naive algorithm would compare every vertex to the reference-feature given in the query. This comparison will take about $50\ ms$ to $100\ ms$ per vertex, thus a naive implementation on a single core would take about $100\ s$ to $200\ s$ to fulfill only one query (given a graph size of 2000 vertices). This obviously is nowhere near a result that supports online-queries and this is only for queries, which take considerably less time, compared to an insert operation due to more messages being sent during an insert-operation (see chapter 4 for details). The following evaluation will now show the improvements possible using the architectures presented in this work.

6.2 Setup

The evaluation was performed using the Microsoft Azure platform, utilizing (up to) eight D14-instances, each containing 16 CPU-cores (Intel Xeon E5-2660, 2.20 GHz) and 112GB RAM. Every participating Azure-instance runs a local instance of MongoDB which runs a sharded collection for the distributed spatial index and a number of local collections for the workers of that specific instance. Also, every instance runs a distributed server for the distributed hash table. Per Azure-instance, up to 16 workers will be started to ensure a rough 1-to-1 setting between workers and CPU-cores (which is only roughly equal as every worker will spawn a number of threads, thus possibly taking up more CPU-cores than only one).

6.3 Methodology

Two sample components which insert feature-vertices and post queries to the system will be used, both components use the "Labeled Faces in the Wild" (LFW) [HRBLM07] database of faces to post new feature-vertices and generate queries. To enable a realistic workload this component also generates traces of walking-movements of persons on a given 2D-topology which are being used to tag the feature-vertices with the 2D-location. The movement of up to 15 persons will be simulated at a time, simulating these persons on a $100m \times 100m$-map. Every person will take 1 to 3 steps in the same direction until a new direction is chosen. Every person will generate a new vertex at every step, to ensure consistent results the faces of a specific person will be reused, but only after at least 20 steps.
This component thereby simulates the presence of an actual image-capturing component (e.g. a camera network). A new location for a given person is generated on average every $750\ ms$, while the walking speed of a person varies from $1.2\ \frac{m}{s}$ to $2.0\ \frac{m}{s}$ [Ral58].

200 vertices will be inserted into the graph for every person, which simulates a scenario of about 2.5 minutes for every person.
Every scenario is being run with two different random-seeds, simulating different workloads and each of these scenarios will be run two times to mitigate for spurious events in the network or on the computing platform, as the evaluation-setup uses a virtualized environment and is prone to events like these.

6.4 Metrics & Variables

The metrics and control variables being used in this evaluation are also shown in table 6.1.

6.4.1 Metrics

As listed in the table, there are several metrics that have been identified in the system: For one the time it takes to insert one feature-vertex into the system will be compared and evaluated. Also the time to execute a query and return the result is an important metric, which is being determined by submitting a set of **InteractiveQueries** to the system, observing the latency on those queries. To determine whether a specific configuration of the system scales with the size of the problem, another important metric is the maximum number of insertions of feature-vertices per second and respectively the maximum number of queries per second that the system can handle without getting backlogged and overloaded. Another metric is the total number of remote edges (edges between two feature-vertices on different worker-nodes) in the system and the number of vertices assigned to each worker-node.

Control Variables				Metrics
Partitioning Strategies	Architectures	Workers	$\frac{\text{Vertices}}{\text{second}}$	Time to Insert One Vertex
Hash	Master-Slave	2 to 64	5 to 25	Time to Execute Query
Greedy	Distributed			Maximum Insertions per Second
Weighted Greedy				Number of Remote Edges
Probabilistic				Number of Vertices per Node
Weighted Probabilistic				

Table 6.1: The various metrics and control variables which will be used in the evaluation. The number of vertices per second is directly determined by the number of persons being tracked in the system. The time toe execute a query is determined by submitting a set of **InteractiveQueries** to the system, observing the latency on those queries.

6.4.2 Variables

Variables are the number of worker-nodes, which varies between 2 and 64, and the chosen partitioning algorithm, the options are the base-line *hash* partitioning, the *greedy* partitioning, the *weighted greedy* partitioning, the *probabilistic* partitioning and the *weighted probabilistic* partitioning. Another variable is the chosen architecture, either the Master-Slave architecture or the Distributed Masters architecture. Another variable is the number of persons being tracked in the system, which also determines the number of vertices per seconds that are being inserted into the system as the insertion rate can be seen as a constant per person. The insertion rate will for every insertion be chosen uniformly at random with a mean of $500\ ms$ and a maximum of $1000\ ms$.

6.5 Results

This section will now cover the actual results of the evaluation while focusing on the aforementioned variables and metrics. The evaluation will cover both architectures as described in chapter 4, the centralized Master-Slave architecture and the Distributed Masters architecture.

6.5.1 Queue

First, the evaluation will focus on showing the number of messages that can be processed per second. The setup for this is the queue running on one machine while a load-generator and message-consumer is being run on another machine. The messages for this test are simple integers (for identification-purposes) and a random-length string with a length of 1000 to 5000

characters, resulting in message of 1kB to 5kB. This test is modeled after the performance tests for ActiveMQ [Act] which employ a similar message-size-distribution.

The results for the custom queue design (as presented in section 4.3.1) can be seen in figure 6.1. They show near-linear scalability in term of messages per queue which isn't surprising as every queue is being run in its own thread in the current design. It can also be seen that the current queue-design supports about 5000 $\frac{\text{messages}}{s*\text{queue}}$. This experiment has been run with 1 to 5 queues, with 50000 messages being inserted into each queue. Also, the number of consumers has been varied from 4 to 8 and 16, but no significant differences can be seen between those runs, which is why the experiment presented on here was being done with 8 consumers. The results look even better when being compared to the results obtained using ActiveMQ (see [Act]), which average about 20000 $\frac{\text{messages}}{s}$. But, even though it looks surprising at first that a well-tested message-queue seems to perform worse than a custom-implemented solution, this can be easily explained: First, ActiveMQ is implemented on top of the JVM, which puts it in slight disadvantage over the native implementation in C++of the queue presented in this thesis. Also, one has to keep in mind that this custom message-queue-solution doesn't implement a lot of the features of ActiveMQ, e.g. durable messages, and is really focused on raw performance. Additionally, the usage of ZeroMQ as the socket-library allows the custom queue-design to take advantage of non-blocking sending and other small improvements.

All this explains why ActiveMQ was not chosen as the main messaging-queue for this work but a custom solution has been implemented which shows comparable throughput and allows a push-based messaging-pattern which opens up further optimization potential due to less overhead on the message-consumers as compared to a pull-based pattern.

6.5.2 Vertex Insertion

The focus of this section will be the insertion-rate of new vertices to the system as well as the response times when inserting new vertices. The protocol of inserting new vertices is given in section 4.4 for both the centralized Master-Slave architecture as well as the Distributed Masters architecture. Please note that even though the results seem to suggest a very slow system with only a few insertions per second possible, this is actually a quite remarkable reduction when compared to the naive setting, as outlined in section 6.1.

The settings for this evaluation are: 10 persons being tracked (results in an expected mean time between two vertices of $75\ ms$ and 2000 vertices), using the distributed system and the *weighted greedy* partitioning method without using the guided insertion mode. The results show the time it takes to insert one vertex, averaged over 50 vertices respectively and can be seen in figure 6.2, from 50 to 1000 vertices, and in figure 6.3 from 1000 to 2000 vertices. These plots show on one hand that initially inserting a feature-vertex in a normal, non-backlogged system takes between $150\ ms$ to $250\ ms$, depending on the number of workers participating in the system. The results show that increasing the number of workers from 2 to 16 will give

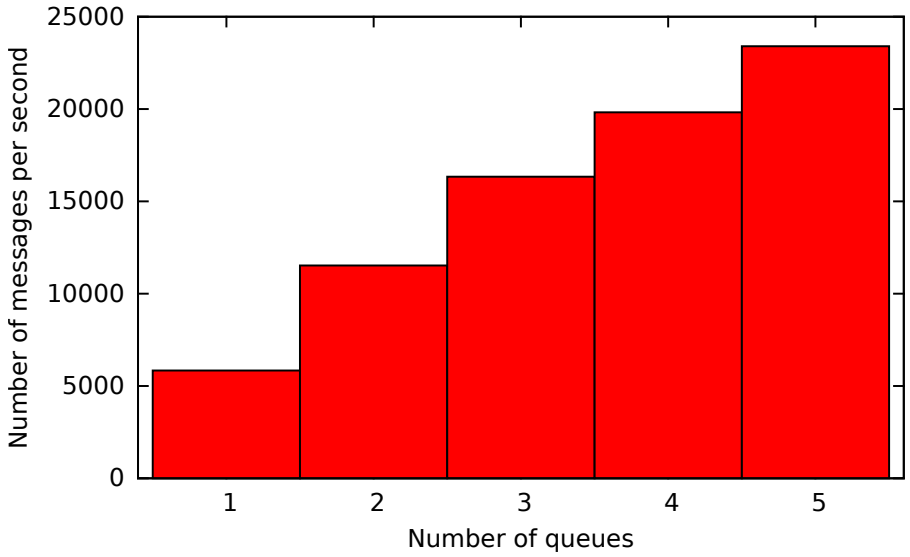


Figure 6.1: The mean number of messages per second using 1 to 5 queues and 8 message-consumers. This experiment was run five times and the numbers were averaged. Each queue had 50000 messages inserted and pushed to the consumers.

an improvement on the insertion time, however there is not a lot of change anymore when increasing the number of workers beyond that. The reason for this is that with a small number of workers, most if not all of them will participate in every query (which is being posted to the system during the insertion process to find the remote edges for the vertex which is being inserted as part of that process) as they will cover a larger area of vertices as compared to a setting with more workers due to the fact that every worker has to be responsible for a larger number of vertices. As number of workers in the system is being increased, it is possible for multiple workers to insert a vertex in parallel, accessing a unique set of workers during the insertion process (i.e., when searching for similar vertices on neighbors) which results in faster response times and most notably an increase in possible insertions per second. After crossing this threshold, the insertion time are only bound by the network delay and actual processing times as summarized in table 6.2 for the case of 64 workers while using the distributed architecture with the *weighted greedy* partitioning strategy and 2000 vertices already in the system. Note that the processing time for the face-recognition is bound by $\approx 45\ ms$ on the machines provided by Microsoft Azure.

But, taking figure 6.3 into account, one can recognize the scalability-bounds when very few workers are being deployed in the system. Taking the case of a 2-worker system as an example and with the knowledge that a new vertex is being inserted on average every $75\ ms$, the process of adding one vertex has to be finished after $150\ ms$, as otherwise new vertices have to be queued before they can be processed. In general, the processing time can be at most $\#workers * insertion\_rate$, otherwise the system will eventually get backlogged. This effect

Component		Time per Component (ms)
Queue	Vertices-Queue	$< 1\ ms$
DM	Add-Vertex	$< 1\ ms$
Worker	Add-Vertex	$55\ ms$
Queue	Queries	$< 1\ ms$
DM	Execute-Query	$76\ ms$
DM	Collect-Responses	$118\ ms$
	Other	$6\ ms$
	Total	$255\ ms$

Table 6.2: The timing breakdown of one vertex insertion with 2000 vertices already in the system, using the centralized Distributed Masters architecture with 64 worker-nodes and the *weighted greedy* partitioning strategy. "Other" includes serialization and deserialization of messages, sending messages, receiving messages and network delay. DM is the abbreviation of a worker-node functioning as a **D**istributed **M**aster.

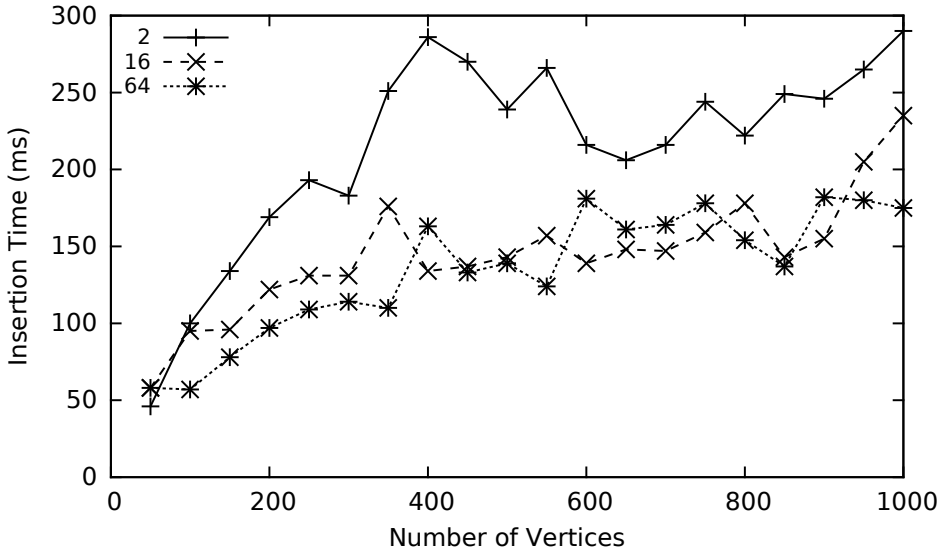


Figure 6.2: The insertion time in milliseconds for a new vertex, ranging from 50 to 1000 vertices, using the Distributed Masters architecture and the *weighted greedy* partitioning strategy, varying the number of worker-nodes from 2 to 64.

can be seen in figure 6.3, as the settings with 2, 4 and 8 workers eventually get overloaded (with the given settings) and start backlogging while there are still new vertices being added, resulting in high delays in processing these requests.

To determine the maximum possible rate of vertex-insertions, the following setup is being used: The system is being run with the number of workers varying from 2 to 64, while the number of

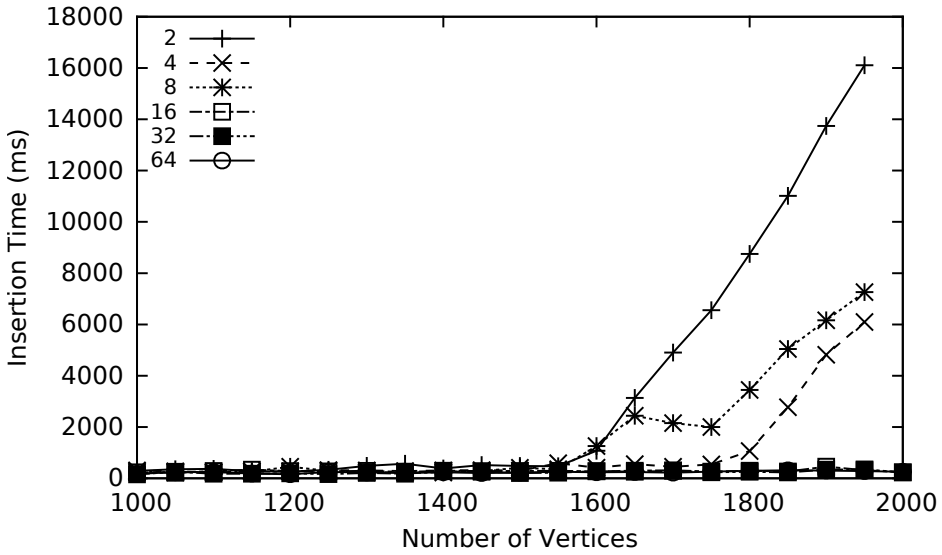


Figure 6.3: The insertion time in milliseconds for a new vertex, ranging from 1000 to 2000 vertices, using the Distributed Masters architecture and the *weighted greedy* partitioning strategy, varying the number of worker-nodes from 2 to 64.

persons, and thus the insertion rate, is being varied until a setup is found that doesn't incur a backlogged system (response to insertion smaller than $500\ ms$) with the final amount of vertices present in the system (e.g. for 10 persons: 2000 vertices). Figure 6.4 and figure 6.5 show the results of this experiment. It is interesting to note that the maximum insertion rate is actually larger than the required minimum with $\#workers * insertion\_rate$ (e.g., $150\ ms$ for 2 workers). This can be explained by taking side-effects of other actions in the system into account, e.g., moving vertices to other workers or executing queries, which will happen concurrently to the actual insertions, resulting in slightly larger maximum insertion rates. The figure also shows that the maximum insertion rate doesn't scale perfectly with the number of workers which is due to multiple reasons: On one hand, as the repartitioning algorithm is being run after every insertion, there is some network contention when moving vertices onto new workers which adds to the latency. Another reason is the usage of the distributed spatial index which also needs time to reflect changes in its data-structures among all connected peers.

To see how well the insertion rate scales with the number of workers, figure 6.6 shows a logarithmic scale plot of the maximum insertion rate in vertices per second. From this plot it becomes clear that the speedup of the system is not perfect (which would result in a straight line in the plot) but still achieves a notable speedup when doubling the number of workers participating in the system, validating the claim for a well-scaling system. The shortcomings in the speedup (compared to an ideal speedup) are a result of the co-dependencies and added computational tasks that were noted above.

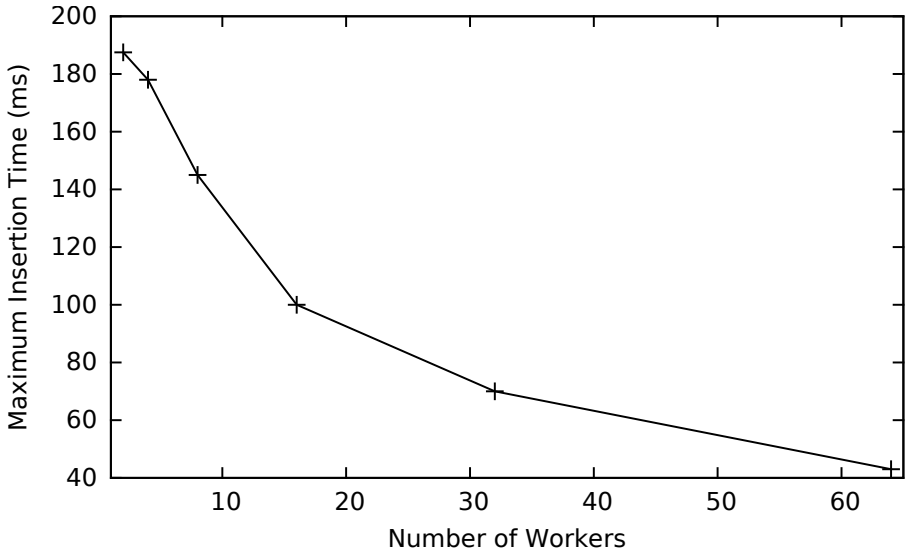


Figure 6.4: The maximum insertion time which results in a non-backlogged system, given in milliseconds, between two vertices, using 2 to 64 workers, the Distributed Masters architecture and the *weighted greedy* partitioning strategy.

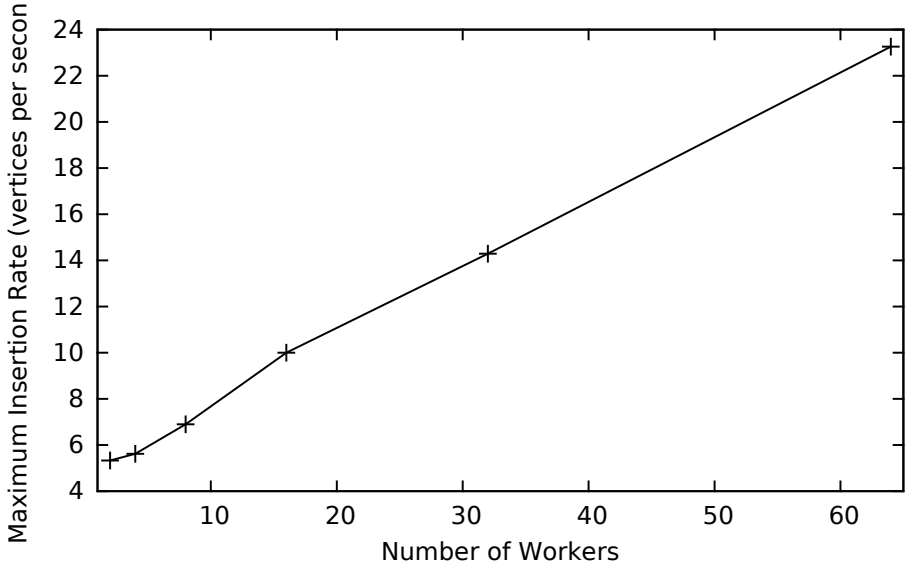


Figure 6.5: The maximum insertion rate which results in a non-backlogged system, given in vertices per second, using 2 to 64 workers, the Distributed Mastersuted architecture and the *weighted greedy* partitioning strategy.

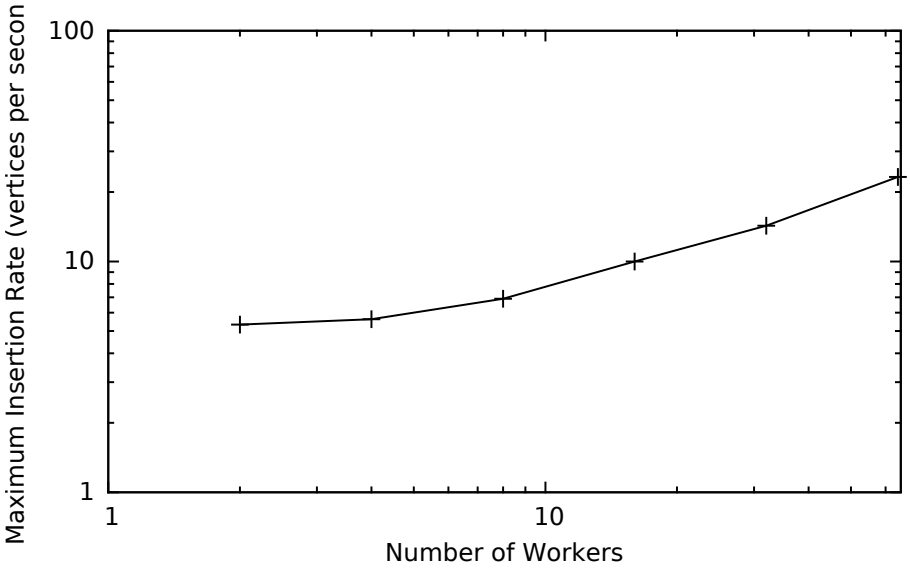


Figure 6.6: The maximum insertion rate which results in a non-backlogged system, given in vertices per second, using 2 to 64 workers, the Distributed Masters architecture and the *weighted greedy* partitioning strategy, plotted in logscale on both axes.

6.5.3 Master-Slave versus Distributed Masters

The purpose of this evaluation is to compare the centralized Master-Slavearchitecture and the Distributed Mastersarchitecture against each other. The setup is the same as before, 10 persons will be tracked (resulting in an expected insertion of a vertex every $75\ ms$) using the *weighted greedy* partitioning strategy. The number of workers will be varied from 16 to 64 and the chosen architecture will be varied as well. The results of this experiment are also split into two parts for better readability and are shown in figure 6.7, from 50 to 1000 vertices, and in figure 6.8 from 1000 vertices to 2000 vertices.

In the case of figure 6.7, the distributed strategies are shown to perform better, resulting in lower insertion times, as the number of vertices increases, reaching up to a factor of 3 times reduction in insertion times. This can be explained with the added latency due to backlogging at the centralized master-nodes, which not only has to process all insertion-requests but also every query and all responses to queries and simply cannot cope with all the additional requests and thus delays the whole operation of the system. Table 6.3 shows the breakdown of time being spent in the various parts of the system, which shows that the actual backlogging happens in the query-execution at the master-node. This means that there are too many queries queued up at the master. As the query-execution is a crucial part of the system and is being used every time a new vertex is being inserted (see figure 4.4 for details on this) any delays in this component will ultimately result in delays and back-logging for the whole system. This constitutes a major disadvantage for the Master-Slave architecture.

Figure 6.8 then shows the situation if the number of vertices is being increased furthermore, beyond 1000 vertices. It is obvious that the centralized Master-Slave architecture is not scalable and response times increase drastically as the number of vertices is being increased. However, it is also interesting to note that the system is actually getting overloaded at a later point in time if more workers are added to the system. This is at first a counter-intuitive result (as one would expect the centralized master to get overloaded more quickly with more workers participating in the system) but has got an easy explanation: If the number of workers in the system is doubled, the execution of one vertex-insertion may take up to twice as much time as the setting with only half as many workers to not cause back-logging at the worker-nodes. However, it can be seen in the graph that even with more workers added and thus a larger tolerance for missed deadlines, the response times increase drastically after the maximum processing time is being passed. The scalability problem also occurs in the query-execution rather than in the actual processing of the feature-vertex as the query-execution involves access to the spatial index and not only has to process the query-request but also has to collect the query-responses from all participating nodes, rendering a centralized architecture an ineffective solution to the given problem.

A final comparison between the two architectures is depicted in figure 6.9 and shows the time it takes to query for one vertex and its neighbors, using an **InteractiveQuery**. The results show again that the centralized architecture doesn't scale well with the size of the data-set and the number of workers. Additionally, the same effect as seen in the earlier scenario can be noticed as the response times actually decrease with an increasing number of workers, which can again be explained due to the added tolerance in missing deadlines in a system with more worker-nodes. Also, an interesting effect concerning the periodicity of events is noticeable in this figure: It can be seen that the query-response times for especially the 64-worker-nodes instance shows some backlogging as well but seems to be recovering from this at times before the response times get worse again. This can be traced to the effects of the randomized input, which also randomizes the time between two consecutive operations being posted to the system. If this time is too short for the system to accurately and timely process the input, backlogging starts to be noticeable. But, the system can also partially recover from this state of backlogging if the rate of input is randomly lower than before, giving the system time to catch up on missed deadlines. This effect can be seen very well in the figure at hand, especially for the configuration of 64-worker-nodes.

6.5.4 Partitioning Strategies

In this section, the partitioning strategies introduced in section 3.2 will be evaluated. In particular, the strategies involved in this evaluation are: *hash* (no partitioning), *greedy*, *weighted greedy*, *probabilistic* and *weighted probabilistic*.

The setting for this evaluation is the Distributed Masters architecture with 64 worker-nodes, tracking 10 persons, resulting in an expected time between two vertex-insertions of $75\ ms$

Component		Time per Component (ms)
Master	Add-Vertex	$< 1\ ms$
Worker	Add-Vertex	$43\ ms$
Master	Execute-Query	$24079\ ms$
Master	Collect-Responses	$2394\ ms$
	Other	$6\ ms$
	Total	$26522\ ms$

Table 6.3: The timing breakdown of one vertex insertion with 2000 vertices already in the system, using the centralized Master-Slave architecture with 64 worker-nodes and the *weighted greedy* partitioning strategy. "Other" includes serialization and deserialization of messages, sending and receiving messages and network delay.

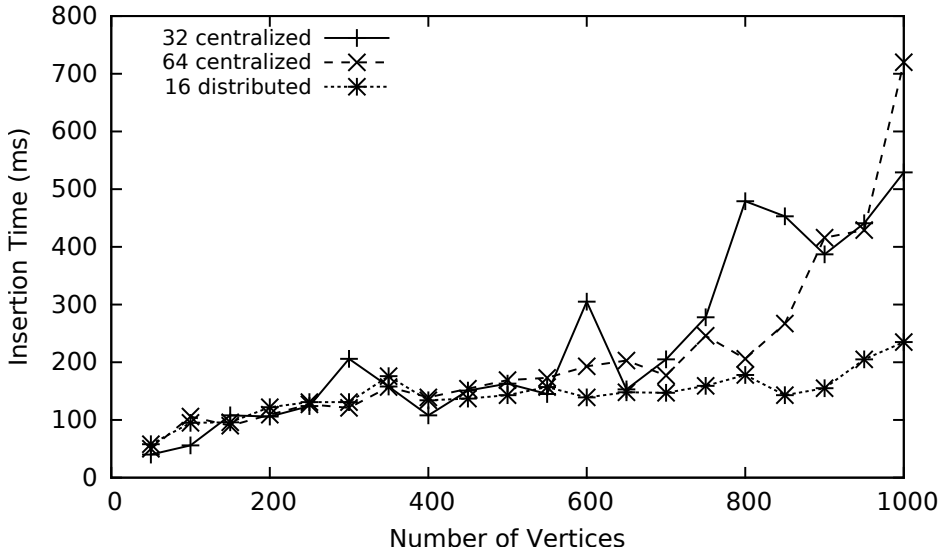


Figure 6.7: The insertion times in milliseconds for one vertex, when inserting 50 to 1000 vertices, comparing the centralized Master-Slave architecture and Distributed Masters architecture using 16 to 64 workers each.

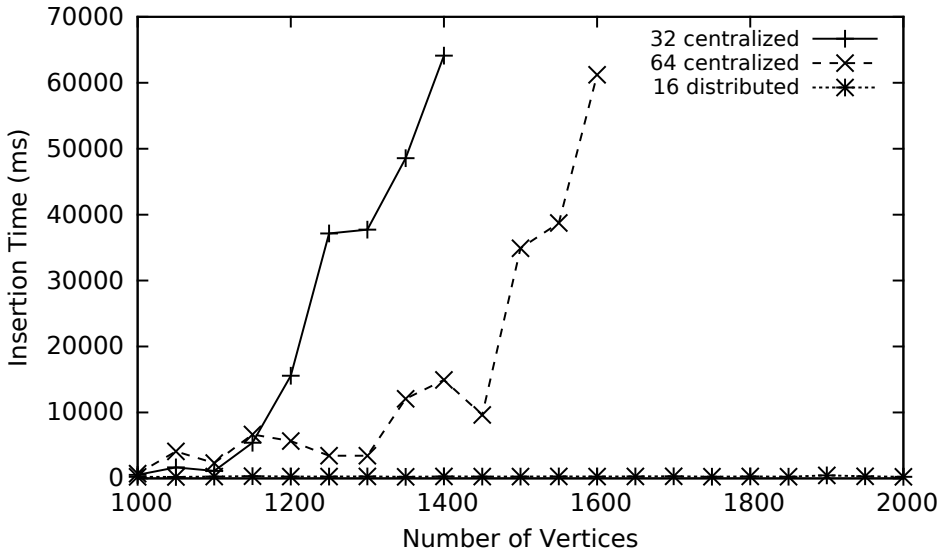


Figure 6.8: The insertion times in milliseconds for one vertex, when inserting 1000 to 2000 vertices, comparing the centralized Master-Slave architecture and Distributed Masters architecture using 16 to 64 workers each. The plots for the centralized settings are only plotted until the end of the insertion-phase after 2.5 minutes.

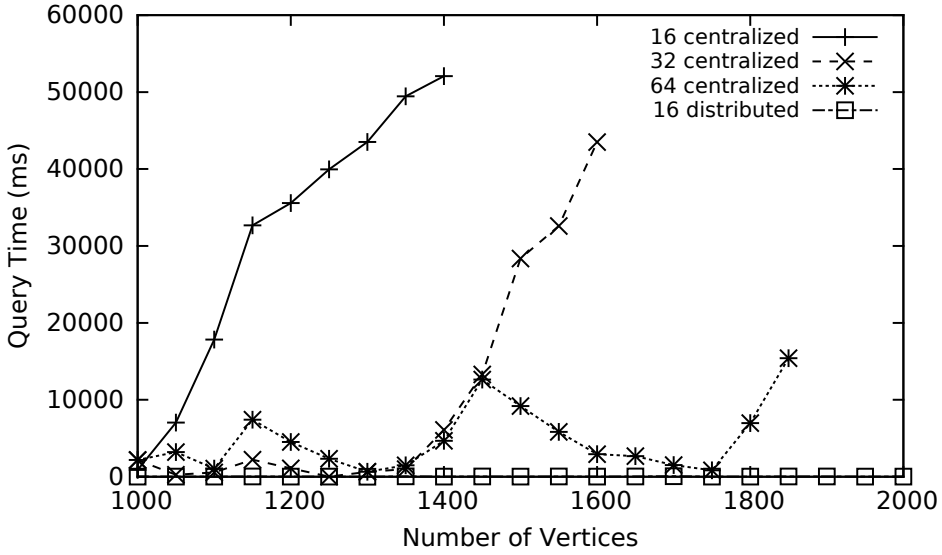


Figure 6.9: The time in milliseconds to query for one vertex using an **InteractiveQuery**, with 1000 to 2000 vertices already in the system, comparing the centralized Master-Slave architecture and Distributed Masters architecture using 16 to 64 workers each. The plots for the centralized settings are only plotted until the end of the insertion-phase after 2.5 minutes.

and 2000 total vertices. The first evaluation for this experiment is depicted in figure 6.11 and displays the ratio of local and remote edges for the various strategies, demonstrating that the *weighted greedy* strategy exhibits the lowest number of remote edges. Note that the number of total edges is not the same for every experiment as both probabilistic partitioning strategies, *probabilistic* and *weighted probabilistic*, show excessive backlogging (see figure 6.12 for reference). As such these configurations result in a different graph-structure as the system will only focus on spatiotemporally "close" vertices which due to the backlogging induced with these strategies will result in less potential edges (e.g. the last vertices are inserted much later in these configurations which results in less remote edges for these particular vertices due to a "stretched" time-graph and less temporally "close" vertices). Figure 6.11 also shows that the *weighted greedy* strategy results in the fewest remote edges and shows a considerable improvement compared to the general *greedy* strategy. This is a result of taking the partition size of a possible new worker-node into account for this specific strategy. This allows for better judging the possible clustering in that worker-node for a specific vertex, resulting in better decisions when actually executing the balancing strategy. The same can be said for the *weighted probabilistic* method, but only to a lesser extent as the improvement in remote edges is only moderate over the general *probabilistic* strategy. The reason for this is the already present improvement which the *probabilistic* strategy is showing in comparison to the *greedy* strategy, with the *probabilistic strategy* only having about a third of the remote edges than the *greedy* strategy.

Another interesting results is depicted in table 6.4 which shows the minimal and maximal number of vertices on every worker-node in a setting using the Distributed Masters architecture with 64 worker-nodes, after inserting 2000 vertices and 10 tracked people. This shows that both the probabilistic strategies perform very poorly in distributing the vertices evenly among the workers, with the *probabilistic* strategy performing worst by putting almost a quarter of all vertices onto one worker-node, with the *weighted probabilistic* strategy showing some better, but still not good results compared to both the greedy strategies. However, even both the greedy strategies don't perform exceptionally well, as both of these strategies still show some considerable difference in minimal and maximal number of vertices as compared to the *hash* strategy which exhibits a perfect load-balancing. To add to these results, figure 6.10 shows the minimal and maximal number of vertices per worker using the distributed architecture with 2 to 64 worker-nodes after inserting 2000 vertices. This result shows that the good results for the *weighted greedy* strategy in terms of load-balancing seem to work for a range of different numbers of worker-nodes. It is interesting to note that the largest variance in terms of load-balancing the vertices is visible at 8 and 16 workers. The explanation for this result can be found in the design of the scenario: Once again, 10 persons are being tracked, which naturally results in about 10 to 15 clusters of well connected vertices. With 8 or 16 worker-nodes, it is possible to put such clusters completely onto one worker-node and due to the low number of worker-nodes and thus the increased probability of a matching vertex to be put on the same machine, larger clusters may be completely put on a single worker-node, resulting in a skewed statistic in terms of load-balancing while also minimizing the number of remote edges.

Strategy	Min Vertices	Max Vertices
Hash	31	32
Weighted Greedy	3	170
Greedy	7	192
Weighted Probabilistic	8	233
Probabilistic	2	493

Table 6.4: The minimal and maximal number of vertices using the Distributed Masters architecture with 64 worker-nodes, after inserting 2000 vertices into the system using a varying vertex partitioning strategy.

The results showing the effect which the different partitioning strategies have on the insertion times of vertices for this experiment are depicted in figure 6.12 and figure 6.13. Figure 6.12 shows that both probabilistic partitioning strategies do not perform well in respect to insertion times. The reason for this are excessive movements of vertices between workers that these strategies induce. Figure 6.14 shows the number of movements for all partitioning strategies in the system and shows the excessive movements that the *probabilistic* strategies induce, especially with respect to the *weighted probabilistic* strategy. The movement of a vertex involves a number of operations such as updating the distributed hash table, moving the attached feature over the network and updating local spatial indices and needs exclusive access to the local data-store of vertices on a worker. This drives the decision of focusing on the greedy strategies which do not show any excessive back-logging. The results for this evaluation are depicted in figure 6.13 and show the improvements of the *weighted greedy* strategy over the general *greedy* strategy. It is also interesting to note that the *hashing* partitioning strategy performs comparably well to the *weighted greedy* strategy. This is particularly of importance as the performance-gains of the *weighted greedy* strategy compared to the general *greedy* strategy show that a better clustering of the graph leads to lesser insertion times. In addition to this result, the current setting of the experiment focuses on a testbed with high network speeds and low latency, favoring the *hashing* strategy considerably: Even if the *hashing* strategy uses more remote edges to fulfill a query than the other partitioning strategies, this is not a large problem as queries only need read-access to the data-store in the worker-nodes and can thus be executed in parallel. Also, the *hashing* strategy doesn't incur movements of vertices between workers, which in turn results in a lot less contention at the worker-nodes when processing new vertices. Even with all these favorable settings, the *weighted greedy* strategy still performs equally well compared to the *hashing* strategy. As part of the future work, a possible distribution of the graph in a setting of edge- or fog-computing (which uses both end-user devices and nearby edge-resources) is a promising avenue to deploy this technology, especially using the *weighted greedy* strategy that shows very few remote edges and good insert-performance.

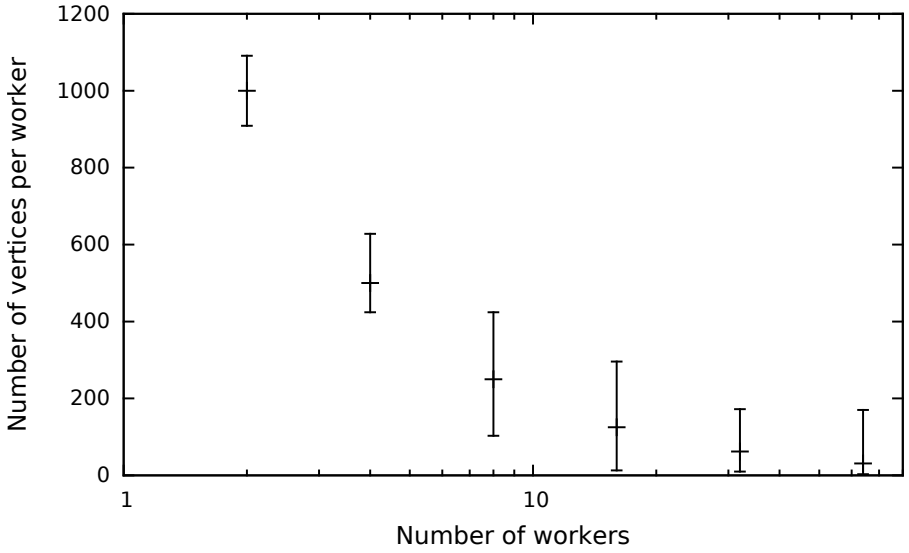


Figure 6.10: The distribution of vertices with 2 to 64 workers, showing the minimal and maximal number of vertices on one worker as well as the mean number of vertices per worker using the *weighted greedy* partitioning strategy and the Distributed Masters architecture.

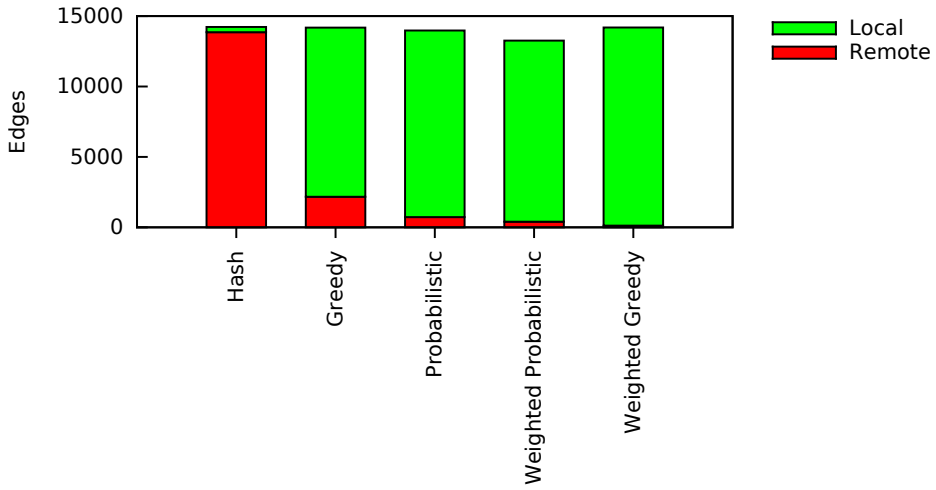


Figure 6.11: The distribution of edges, using the Distributed Masters architecture with 64 worker-nodes, after the insertion of 2000 vertices with varying partitioning strategies.

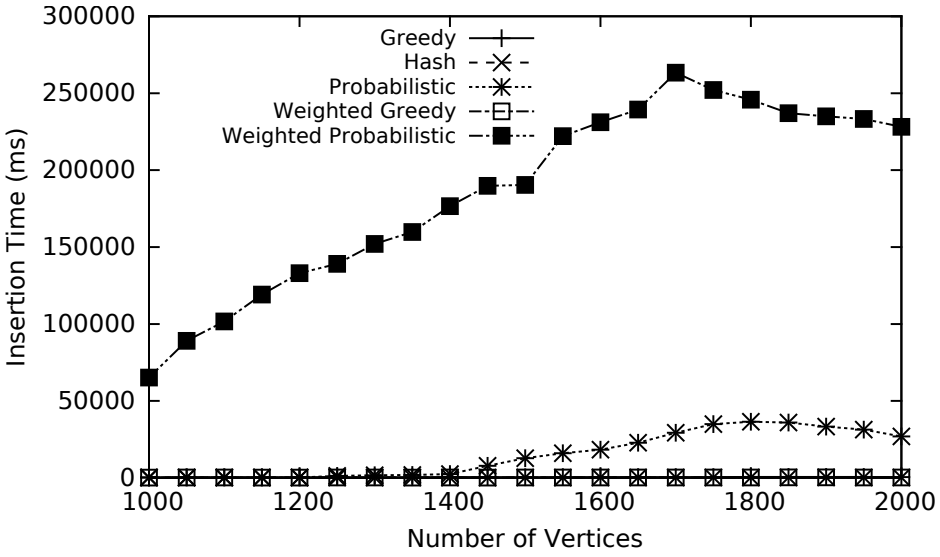


Figure 6.12: The insertion time in milliseconds for new vertices using the Distributed Masters architecture with 64 worker-nodes with five different partitioning strategies from 1000 to 2000 vertices.

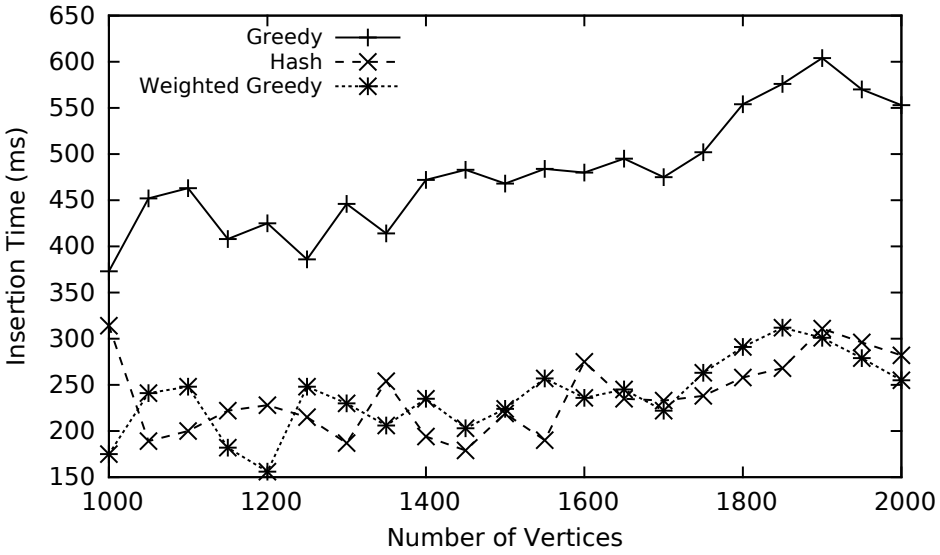


Figure 6.13: The insertion time in milliseconds for new vertices using the Distributed Masters architecture with 64 worker-nodes with three different partitioning strategies from 1000 to 2000 vertices.

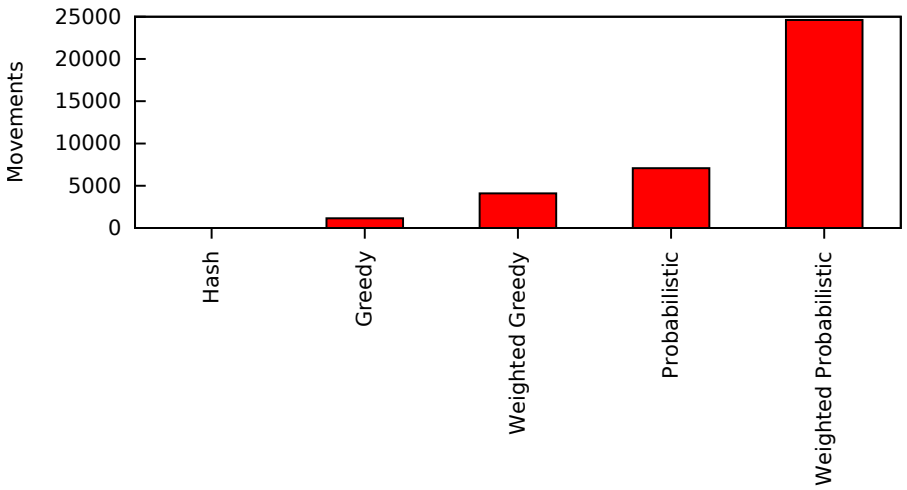


Figure 6.14: The number of movements when using the Distributed Masters architecture and five different partitioning strategies after inserting 2000 vertices using 64 worker-nodes.

6.5.5 Graph Optimization

Another concept which shall be looked at in this evaluation is the graph optimization introduced in section 3.5.

As the graph-optimization is only active when executing queries (and most notably, is not active when executing the search for edges for a newly inserted vertex to ensure correct results for these computations), the evaluation will only focus on a query-based workload for this evaluation. This workload consists of using a `TimeConeQuery`, querying for randomly picked time-cones in the area of already inserted vertices.

This query-workloads will submit five queries for every data-point shown in the graph, with the average of those five queries constituting one data-point. The setting for all evaluations in this section is the Distributed Masters architecture with 16 workers, tracking 12 persons using the *weighted greedy* partitioning strategy.

The average query-response-times up to 1500 vertices are shown in figure 6.15. First, it can be seen that these results show a response time ranging from $50\ ms$ to $200\ ms$ and mostly show a favorable behavior for the setting with the graph-optimization being active. But, it shall noted that at times the setting without the graph-optimization actually results in faster response times (e.g. at 700 vertices). This can be explained with the fact that the system is at the same time still being used for other purposes (i.e. inserting new vertices) which results in larger response times, but it is for most parts clear that the graph optimization reduces the

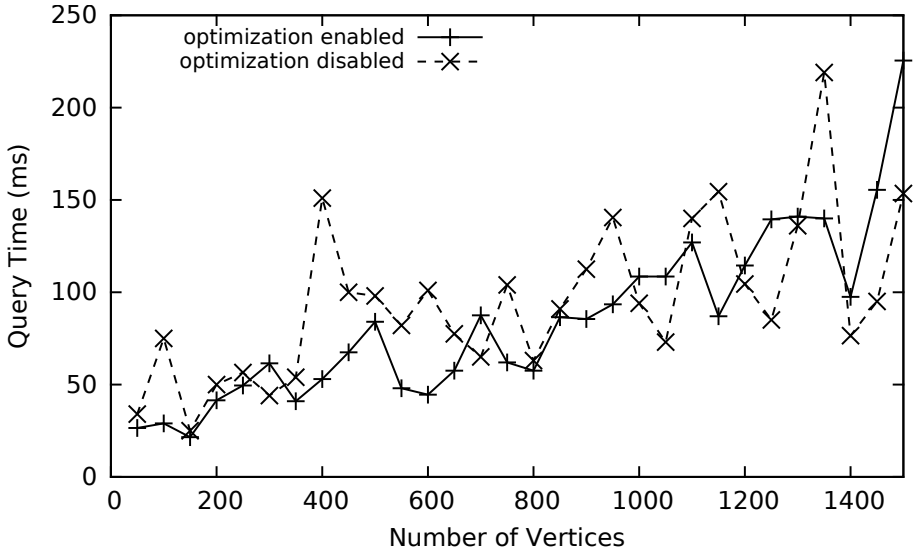


Figure 6.15: The query times for a `TimeConeQuery` in milliseconds for one query when using the Distributed Masters with 16 worker-nodes, the *weighted greedy* partitioning strategy from 50 to 1500 vertices, both with and without enabling the graph-optimization for queries.

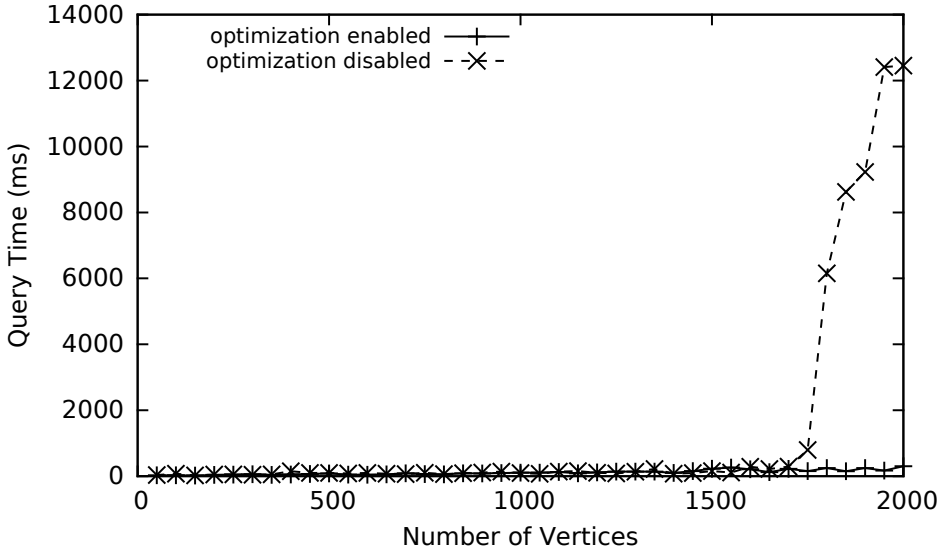


Figure 6.16: The query times for a `TimeConeQuery` in milliseconds for one query when using the Distributed Masters with 16 worker-nodes, the *weighted greedy* partitioning strategy from 50 to 2000 vertices, both with and without enabling the graph-optimization for queries.

mean response times of queries. Also, it is clear from the graph the graph-optimization doesn't show spikes in the response-times as compared to the setting without the graph optimization due to the nature of the graph-optimization of not requiring too many computational resources which might not be available at times (as discussed before).

Another evaluation will focus at the behavior up to 2000 vertices being inserted into the system. The results of this evaluation can be seen in figure 6.16. This figure shows a very interesting behavior for the setting without the graph-optimization being enabled: It becomes overloaded and back-logged, leading to response times of multiple seconds (as compared to a few hundred milliseconds as discussed before). This behavior can be explained due to the additional computational overhead of running the full queries as compared to using the graph optimization technique when the system is already at the verge of getting back-logged. This shows the actual impact of the graph-optimization as it allows for a better scalability-behavior. The setting with the graph-optimization being enabled still shows response times of only a few hundred milliseconds, supporting the claim for better scalability of the graph-optimization technique.

6.5.6 Guided Insert

The effects of the *guided insert* insertion strategy shall now be the focus of the evaluation (which was introduced in section 3.3). To evaluate this strategy, the insertion times of a new vertex with and without the *guided insert*-strategy active shall be observed, as well as the query-times, the number of moved vertices and the edge-distributions.

First, the evaluation will focus at the insertion times when using the *guided insert*-strategy as compared to using the plain, *round-robin*-based insertion strategy. The results of this experiment can be seen in figure 6.17, using 16 workers, the Distributed Masters architecture and the *weighted greedy* partitioning strategy while inserting 2000 vertices. This figure shows two interesting trends: On one hand, it is immediately apparent that the non-*guided-insert* setup is suffering from backlogging of the system and thus is showing a less fortunate scaling behavior. This can be explained with the less fortunate initial vertex-placing selections that the *round-robin* strategy will incur as compared to the *guided-insert* scenario. This leads to increased movements of vertices which on the other hand results in less actual work being done in the system (as the moving vertices is only a secondary task and not one that supports the major tasks of the system, i.e. inserting and querying vertices). A second observation is that the insertion times for both strategies (apart from the end) are mostly equal, with the *guided-insert*-strategy being a little slower than the *round-robin* strategy. This can be explained due to less-than-perfect load-balancing when using the *guided-insert*-strategy as compared to a *round-robin*-scheme, as some workers are bound to receive more vertices than others, resulting in slight contention at these workers. But, it is important to note that this is only a minor problem for these workers and it does not impact the functionality of the system as

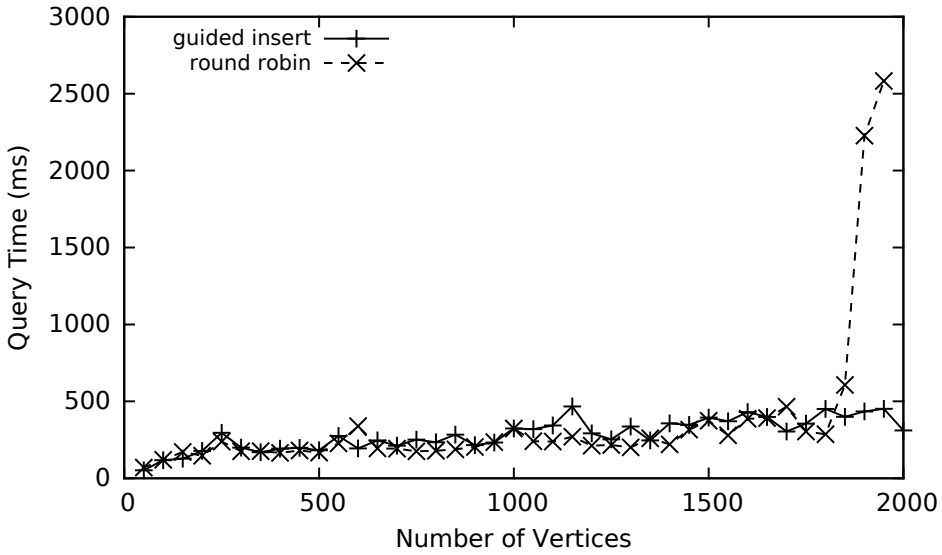


Figure 6.17: The insertion times in milliseconds for one vertex when using the Distributed Masters architecture and five different partitioning strategies after inserting 2000 vertices using 64 worker-nodes.

	Total Number of Vertex-Movements
Guided Insert	2135
Round Robin	2982

Table 6.5: The number of vertex movements (induced by running the partitioning strategy) using the Distributed Masters architecture with 16 worker-nodes, after inserting 2000 vertices into the system, either with or without using the *guided insert* insertion strategy.

such in any kind, thus one can conclude that is a good trade to sacrifice slightly on the initial load-balancing to gain more scalability.

Another evaluation actually shows the increased vertex-movements which have been identified as the main problem for scaling the round-robin-scheme in the afore-mentioned evaluation. The results of this evaluation can be seen in table 6.5 and show that the *guided-insert*-strategy incurs almost 30% less movements than the *round-robin*-strategy, due to the more fortunate initial placements of the vertices on the workers.

But, not all is good for the *guided-insert*-strategy: As discussed before, this strategy sacrifices on load-balancing and might thus result in a lot poorer vertex-balancing among the workers. Albeit this statement is true, the situation is not as bad as it might be believed to be. Table 6.6 shows the minimal and maximal number of vertices on each worker, with and without the

	Min Vertices	Max Vertices
Guided Insert	19	288
Round Robin	47	280

Table 6.6: The minimal and maximal number of vertices using the Distributed Masters architecture with 16 worker-nodes, after inserting 2000 vertices into the system, either with or without using the *guided insert* insertion strategy.

	Total Number of Edges	Local Edges	Remote Edges
Guided Insert	13990	13875	115
Round Robin	13990	13779	211

Table 6.7: The number of edges, separated into local and remote edges, using the Distributed Masters architecture with 16 worker-nodes, after inserting 2000 vertices into the system, either with or without using the *guided insert* insertion strategy.

guided-insert-strategy active using the Distributed Masters architecture with 16 workers, the *weighted greedy* partitioning scheme and 2000 vertices already inserted into the system. The results show that the *guided-insert*-strategy suffers from a slightly worse load-balancing than the *round-robin*-scheme, but these results show that both strategies do not result in a perfect load-balancing in either setup, mostly due to the partitioning scheme.

Another interesting question is whether the *guided-insert*-strategy can potentially help on reducing remote edges, due to more intelligent initial vertex-placements. The results for this experiment are shown in table 6.7, again with 16 workers, the Distributed Masters architecture, the *weighted greedy* partitioning scheme and 2000 vertices inserted into the system. These results show a reduction in remote edges for the *guided-insert*-strategy by about 100 edges (which corresponds to a rough 100% reduction!), but both setups actually result in a very good number of remote edges with the *guided-insert*-strategy contributing to an even better result.

6.6 Summary of Results

The key findings of the evaluation are:

- The Distributed Masters architecture scales well with the number of workers and the size of the problem. The timing-breakdown shows that most of the time is being spent in the actual execution of the insertion- and query-components, mainly on the image processing part while other components of the architecture are demonstrated to be effective for the given scale.

- The *weighted greedy* partitioning strategy performs equally well compared to the *hashing* strategy in terms of execution times during insertions of vertices while providing far better edge-locality and reasonable load-balancing of the vertices among the worker-nodes.
- Both probabilistic strategies perform poorly in terms of overall performance and result in a severely backlogged system due to incurring too many vertex-movements between the worker-nodes, rendering them unusable for executing the actual insertion- or query-tasks. However, the edge-locality both probabilistic strategies is better compared to the general *greedy* strategy.
- The graph-optimization has been shown to result in slightly faster query-execution but also scales better with the number of vertices and allows for faster (and most importantly non-backlogged) results as more vertices are being inserted into the system.
- The *guided-insert*-strategy results in better results (insertion-times, number of vertex-movements and number of remote edges) as compared to the plain *round-robin*-scheme by sacrificing on load-balancing to improve the initial vertex-placement. The down-side of the *guided-insert*-strategy is the slightly worse load-balancing compared to the *round-robin*-scheme which can be accepted as a trade-off for better performance.

7 Future Work

This chapter will briefly outline the potential future avenues for the work presented in this thesis.

A current project is focused on integrating the given system which allows for efficient querying and inserting of vertices with a real camera network. A potential starting point for this work is the previously developed abstraction for a camera network [HSS+11] which provides an interface for domain experts to deploy their image- and video-processing algorithms in a distributed camera network. Given this abstraction, the task for the system described in this work would be to gather key features, generate vertices out of these features and submit those to the system described in this work.

Another extension focusing of the architecture of the system would be a departure from a traditional datacenter-centric approach towards a peer-to-peer - like architecture. This would allow the use of the system in a widely distributed area, making use of nearby edge-resources or mobile devices carried by end-users and could be built on top of a previously developed abstraction [HLR+13].

7.1 Distributed Messaging Queue

Another extension to the system would be to replace the current queue-design with a distributed messaging queue, to allow for scaling to larger workloads and to remove the last non-distributed entity of the system and replace it with a distributed version. One potential avenue is to exploit the spatiolocality based partitioning of the worker-nodes and just use message-queues for specific areas. However, it is not clear how much overhead a distributed messaging queue will account for in contrast to the light-weight queue design that has been implemented in the current system.

7.2 Federated Architecture

One more avenue for further development is to enable a federated system being run using the current architecture as a building block. In this kind of system, the system presented in this work would be used as a “black-box” to handle the workload of one area, while being connected

to its neighbor-areas, handling the query- and insertion-workload on the system in coordination with nearby clusters. One of the challenges in this kind of setting is to handle queries in a widely distributed manner, without overloading any potentially centralized components. One possible avenue of doing this is to continue the usage of the message-queue and expand it to be the (per-cluster-central) communication-interface to other clusters. In this kind of scenario, one would also need a spatial index covering the wide area of clusters that are deployed to facilitate an easy and quick lookup for clusters being involved in the computation of a query or insert-operation.

8 Conclusion

This master thesis presents an architecture for distributed graph processing with an emphasis on spatiotemporal queries in the context of camera networks. The workload of inserting vertices into the distributed graph and processing the spatiotemporal queries on the graph is being handled in a fully distributed manner among the worker-nodes of the system. This work also provides an overview over a number of different graph partitioning heuristics, providing for high edge-locality while still enabling a fair vertex distribution of the graph among the worker-nodes.

The results show that the fully distributed system is a scalable solution when used with the *weighted greedy* strategy and performs well when scaling the number of workers and the problem size. Among all heuristic partitioning strategies, the *weighted greedy* strategy performs best in the number of remote edges, the distribution of vertices among the worker-nodes and, most importantly, the effect of the partitioning strategy on the execution time of tasks in the system. The *weighted greedy* strategy exhibits comparable performance as a basic *hashing* strategy in terms of inserting a vertex into the system or querying the system despite the additional work of moving vertices between worker-nodes. Additionally, the *weighted greedy* strategy also offers the added benefits of resulting in a small number of remote edges which translates to very high edge-locality when compared to the *hashing* strategy which performs very poorly in both of these metrics due to the very nature of this strategy. In addition to this, the *guided insert*-method for a smart initial placement of vertices results in another performance gain and works very well with the presented partitioning strategies.

In conclusion, the distributed architecture with the *weighted greedy* partitioning scheme offers a viable solution to the problem of efficient graph processing and online graph partitioning in the context of camera networks. The contributions of this thesis are the design of an effective architecture for distributed graph processing with online graph partitioning as well as the presentation of various optimization techniques to increase edge-locality via the initial vertex-insertion strategy (see section 3.3) and by choosing a smart vertex partitioning algorithm which can be applied in an online-fashion (see section 3.2).

A Appendix

A.1 Protobuf Messages

The following listing presents all the potential messages which are being sent in the system along with their content. As the system uses the Google Protocol Buffers [Goo08], the markup is based on the protobuf-language, which in turn compiles to C++-code for inclusion in the system.

```
package dfg_rpc;

option optimize_for = SPEED;

message PointMessage
{
      required double x = 1;
      required double y = 2;
}

message BoxAreaMessage
{
      required PointMessage minPoint = 1;
      required PointMessage maxPoint = 2;
}

message TimeDurationMessage
{
      required int64 startTime = 1;
      required int64 endTime = 2;
}

message HostMessage
{
      required string hostName = 1;
      required int64 hostPort = 2;
}

message WorkerMessage
{
      required HostMessage host = 1;
      required int64 id = 2;
}
```

```
message WorkerRegisteredMessage
{
      required WorkerMessage worker = 1;
}

message SpatioTemporalCoordinatesMessage
{
      required int64 time = 1;
      required PointMessage location = 2;
}

message CharacteristicMessage
{
      required bytes characteristic = 1;
}

message TimingMessage
{
      required string id = 1;
      required TimeDurationMessage time = 2;
}

message ExecutionStatisticsMessage
{
      required int64 id = 1;
      required TimeDurationMessage originalHostTiming = 2;
      required HostMessage originalHost = 3;
      repeated TimingMessage timings = 4;
}

message FeatureMessage
{
      required bytes feature = 1;
      required int64 label = 2;
}

message RemoteFeatureVertexMessage
{
      required SpatioTemporalCoordinatesMessage location = 1;
      required string id = 2;
      required bool remote = 3;
}

message FeatureVertexMessage
{
      required SpatioTemporalCoordinatesMessage location = 1;
      required FeatureMessage feature = 2;
      required string id = 3;
      optional ExecutionStatisticsMessage statistics = 4;
      repeated RemoteFeatureVertexMessage edges = 5;
      repeated int64 previousAssignedWorkers = 6;
```

```
}

message AddFeatureVertexMessage
{
      required FeatureVertexMessage featureVertexMessage = 1;
}

message CommonFeatureVertexMessage
{
      required int64 localID = 1;
      required HostMessage returnHost = 2;
      required int64 timestamp = 3;
}

enum RequestorType
{
      WORKER_SERVICE = 1;
      QUERY_RESPONSE_PROCESSOR = 2;
}

message CommonQueryMessage
{
      required string id = 1;
      required HostMessage returnHost = 3;
      optional int64 hostID = 4;
      required RequestorType requestorType = 5;
      optional ExecutionStatisticsMessage statistics = 6;
}

message TimeConeFeatureVertexQueryMessage
{
      required TimeDurationMessage time = 1;
      required double speed = 2;
      required PointMessage startPoint = 3;
      required double startDelta = 4;
      required FeatureVertexMessage featureVertex = 5;
}

message TimeConeQueryMessage
{
      required TimeDurationMessage time = 1;
      required double speed = 2;
      required PointMessage startPoint = 3;
      required double startDelta = 4;
      required CharacteristicMessage characteristic = 5;
}

message SpatioTemporalQueryObjectMessage
{
      required TimeDurationMessage time = 1;
      required BoxAreaMessage area = 2;
```

```
      required CharacteristicMessage characteristic = 3;
}

message InteractiveQueryMessage
{
      required string featureVertexID = 1;
}

message QueryMessage
{
      enum MessageType {
            TimeConeFeatureVertexQueryMessage = 1;
            TimeConeQueryMessage = 2;
            SpatioTemporalQueryObjectMessage = 3;
            InteractiveQueryMessage = 4;
      }

      // Identifies which field is filled in.
      required MessageType type = 1;
      required CommonQueryMessage common = 2;

      // One of the following will be filled in.
      optional TimeConeFeatureVertexQueryMessage timeConeFeatureVertexQueryMessage = 3;
      optional TimeConeQueryMessage timeConeQueryMessage = 4;
      optional SpatioTemporalQueryObjectMessage spatioTemporalQueryObjectMessage = 5;
      optional InteractiveQueryMessage interactiveQueryMessage = 6;
}

message QueryResponseMessage
{
      required CommonQueryMessage common = 1;
      optional ExecutionStatisticsMessage statistics = 5;
      repeated FeatureVertexMessage featureVertices = 2;
}

message RequestFeatureVertexMessage
{
      required CommonQueryMessage common = 1;
      required string featureVertexID = 2;
}

message RequestFeatureVertexResponseMessage
{
      required CommonQueryMessage common = 1;
      required FeatureVertexMessage featureVertex = 2;
}

message DistributedMasterQueryResponseMessage
{
      required QueryResponseMessage queryResonse = 1;
}
```

```
enum PartitionStrategy
{
      GREEDY = 1;
      HASH = 2;
      PROBABILISTIC = 3;
      GREEDY_WEIGHTED = 4;
      PROBABILISTIC_WEIGHTED = 5;
}

message SettingsMessage
{
   required bool useGuidedPlacement = 1;
   required bool useDistributedMode = 2;
   required bool useGraphOptimization = 3;
   required PartitionStrategy partitionStrategy = 4;
   required int64 runHeuristicFrequency = 5;
   required string xpubXsubHost = 6;
   required int64 xpubPort = 7;
   required int64 xsubPort = 8;
   required HostMessage mongoDSIHost = 9;
   required HostMessage queueHost = 10;
   required int64 guidedPlacementThreshold = 11;
   required int64 numberOfWorkers = 12;
   required int64 seed = 13;
}

message WorkerRegistrationResponseMessage
{
      required int64 workerID = 1;
      required SettingsMessage settings = 2;
      repeated WorkerMessage workers = 3;
}

message MoveFeatureVertexMessage
{
      required FeatureVertexMessage featureVertexMessage = 1;
}

// Queue-messages:
enum QueueIdentifierEnum
{
      ADD_FEATURE_VERTEX_QUEUE = 1;
      QUERY_QUEUE = 2;
}

message ResetQueueMessage
{
      required QueueIdentifierEnum queueIdentifier = 1;
}
```

```
message PayloadMessage
{
      enum MessageType {
            AddFeatureVertexMessage = 1;
            QueryMessage = 2;
            QueueBenchmarkMessage = 3;
      }

      // Identifies which field is filled in.
      required MessageType type = 1;

      // One of the following will be filled in.
      optional AddFeatureVertexMessage addFeatureVertexMessage = 2;
      optional QueryMessage queryMessage = 3;
      optional QueueBenchmarkMessage queueBenchmarkMessage = 4;
}

message WorkloadMessage
{
      required QueueIdentifierEnum queueIdentifier = 1;
      required PayloadMessage payload = 2;
}

message ReadyMessage
{
      required QueueIdentifierEnum queueIdentifier = 1;
      required WorkerMessage worker = 2;
}

message InsertMessage
{
      required QueueIdentifierEnum queueIdentifier = 1;
      required PayloadMessage payload = 2;
}

message FeatureVertexInsertedMessage
{
      required ExecutionStatisticsMessage executionStatisticsMessage = 1;
      required string featureVertexID = 2;
}
```

Listing A.1: Overview of all available message-types in the system

Bibliography

[Act] ActiveMQ - Performance. http://activemq.apache.org/performance.html. (Cited on page 58)

[AHC13] M. Akdere, J.-H. Hwang, U. Cetintemel. Real-time Probabilistic Data Association over Streams. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pp. 219–230. ACM, New York, NY, USA, 2013. (Cited on page 21)

[BHK97] P. N. Belhumeur, J. a. P. Hespanha, D. J. Kriegman. Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(7):711–720, 1997. (Cited on pages 49 and 52)

[BK08] D. G. R. Bradski, A. Kaehler. *Learning Opencv, 1st Edition*. O'Reilly Media, Inc., first edition, 2008. (Cited on page 49)

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008. (Cited on page 19)

[CFSV96] T. Cheatham, A. Fahmy, D. Stefanescu, L. Valiant. Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software. In A. Zaky, T. Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*, volume 2 of *The Springer International Series in Software Engineering*, pp. 61–76. Springer US, 1996. (Cited on page 19)

[CHK+12] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, E. Chen. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pp. 85–98. ACM, New York, NY, USA, 2012. (Cited on page 20)

[CLS12] Z. Cai, D. Logothetis, G. Siganos. Facilitating Real-time Graph Mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pp. 1–8. ACM, New York, NY, USA, 2012. (Cited on page 20)

[GGL03] S. Ghemawat, H. Gobioff, S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43. ACM, New York, NY, USA, 2003. (Cited on page 19)

[Gir12] Giraph. http://giraph.apache.org, 2012. (Cited on pages 15, 19 and 21)

[Goo08] Google. Protocol Buffers - Google's data interchange format. https://developers.google.com/protocol-buffers/, 2008. (Cited on pages 49 and 81)

[GV92] A. V. Gerbessiotis, L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, SWAT '92, pp. 1–18. Springer-Verlag, London, UK, UK, 1992. (Cited on page 19)

[HLR+13] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe. Mobile Fog: A Programming Model for Large-scale Applications on the Internet of Things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pp. 15–20. ACM, New York, NY, USA, 2013. (Cited on page 77)

[HML+14] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, E. Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 1:1–1:14. ACM, New York, NY, USA, 2014. (Cited on page 20)

[HRBLM07] G. B. Huang, M. Ramesh, T. Berg, E. Learned-Miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical Report 07-49, University of Massachusetts, Amherst, 2007. (Cited on page 56)

[HSS+11] K. Hong, S. Smaldone, J. Shin, D. Lillethun, L. Iftode, U. Ramachandran. Target container: A target-centric parallel programming abstraction for video-based surveillance. In *2011 Fifth ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC)*, pp. 1–8. 2011. (Cited on page 77)

[KBG12] A. Kyrola, G. Blelloch, C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 31–46. USENIX Association, Berkeley, CA, USA, 2012. (Cited on page 20)

[KL70] B. W. Kernighan, S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49:291–307, 1970. (Cited on pages 21, 23 and 25)

[LZB+13] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 775–787. 2013. (Cited on page 51)

[MAB+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146. ACM, New York, NY, USA, 2010. (Cited on pages 15, 19 and 21)

[MIM15] F. McSherry, M. Isard, D. G. Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland, 2015. (Cited on page 20)

[Mon13] D. Montag. Understanding Neo4j Scalability. http://neo4j.com/resources/wp-understanding-neo4j-scalability/, 2013. (Cited on page 20)

[Neo15] Neo4J, a Fast and Scalable Graph Database. http://neo4j.com, 2015. (Cited on page 20)

[Ori15] OrientDB - Multi-Model NoSQL Database. http://orientdb.com, 2015. (Cited on page 20)

[PS14] A. Presta, A. Shalita. Large-scale graph partitioning with Apache Giraph. https://code.facebook.com/posts/274771932683700/large-scale-graph-partitioning-with-apache-giraph/, 2014. (Cited on page 21)

[Ral58] H. Ralston. Energy-speed relation and optimal speed during level walking. *Internationale Zeitschrift für angewandte Physiologie einschließlich Arbeitsphysiologie*, 17(4):277–283, 1958. (Cited on page 56)

[RPG+13] F. Rahimian, A. Payberah, S. Girdzijauskas, M. Jelasity, S. Haridi. JA-BE-JA: A Distributed Algorithm for Balanced Graph Partitioning. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pp. 51–60. 2013. (Cited on pages 21, 23 and 25)

[SK12] I. Stanton, G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pp. 1222–1230. ACM, New York, NY, USA, 2012. (Cited on page 21)

[SKRC10] K. Shvachko, H. Kuang, S. Radia, R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pp. 1–10. IEEE Computer Society, Washington, DC, USA, 2010. (Cited on page 19)

[SW13] S. Salihoglu, J. Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pp. 22:1–22:12. ACM, New York, NY, USA, 2013. (Cited on page 19)

[TGRV14] C. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pp. 333–342. ACM, New York, NY, USA, 2014. (Cited on page 21)

[XJN+13] J. Xu, V. Jagadeesh, Z. Ni, S. Sunderrajan, B. Manjunath. Graph-Based Topic-Focused Retrieval in Distributed Camera Network. *Multimedia, IEEE Transactions on*, 15(8):2046–2057, 2013. (Cited on page 21)

[Zer12] ZeroMQ. http://zeromq.org/, 2012. (Cited on page 49)

[ZYZ+13] L. Zhuang, A. Yang, Z. Zhou, S. Sastry, Y. Ma. Single-Sample Face Recognition with Image Corruption and Misalignment via Sparse Illumination Transfer. In *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3546–3553. 2013. (Cited on page 49)

All links were last followed on August 3, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature