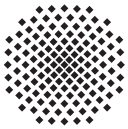


Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Concurrent Query Analytics on Distributed Graph Systems

Jonas Grunert

Course of Study:	Software Engineering
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Dipl.-Inf. Christian Mayer
Commenced:	30 November, 2016
Completed:	01 June, 2017
CR-Classification:	C.2.4,H.3.4

Abstract

Large-scale graph problems, such as shortest path finding or social media graph evaluations, are an important area in computer science. In recent years, important graph applications such as PowerGraph [GLG+12] or PowerLyra [CSCC15] lead to a shift of paradigms of distributed graph processing systems towards processing of *multiple* parallel queries rather than a single global graph algorithm.

Queries usually have locality in graphs, i.e. involve only a subset of the graphs vertices. Suitable partitioning and query synchronization approaches can minimize communication overhead and query latency by exploiting this locality. Additionally, partitioning algorithms must be dynamic as the number and locality of queries can change over time. Existing graph processing systems are not optimized to exploit query locality or to adapt graph partitioning at runtime.

In this thesis we present *Q-Graph*, an open source, multitenant graph analytics system with dynamic graph repartitioning. Q-Graph's query-aware partitioning algorithm *Q-Cut* performs adaptive graph partitioning at runtime. Compared to static partitioning strategies, Q-Cut can exploit runtime knowledge about query locality and workload to improve partitioning dynamically.

Furthermore a case study with an implementation for the shortest path problem and point search queries is presented. We present evaluations showing the performance of Q-Graph and the effectiveness of Q-Cut. Measurements show that Q-Cut improves query processing performance by up to 60% and automatically adapts partitioning on changing query workload and locality, outperforming partitioning methods using domain knowledge.

Kurzfassung

Large-scale Graph Probleme, wie beispielsweise Kürzeste-Wege Suchen oder Social Media Evaluationen, sind ein wichtiger Bereich in der Informatik. In den letzten Jahren zeigen Graph Anwendungen wie PowerGraph [GLG+12] oder PowerLyra [CSCC15] einen Paradigmenwechsel von verteilten Graph Systemen hin zu parallelen Anfragen, statt der Verarbeitung einzelner, globaler Anfragen.

Solche Anfragen besitzen üblicherweise eine Lokalität in der Graph Datenstruktur, d.h. sie betreffen nur einen Teilbereich der Knoten eines Graphs. Geeignete Ansätze zur Partitionierung können dies nutzen um den Kommunikationsaufwand zu reduzieren und die Anfragenlatenz zu minimieren. Außerdem müssen Partitionierungs Algorithmen dynamisch sein, da sich die Anzahl und Lokalität von Anfragen über die Zeit ändern kann. Existierende Graph Systeme sind nicht optimiert um Anfragen Lokalität zu berücksichtigen oder die Graph Partitionierung zur Laufzeit anzupassen.

In dieser Arbeit stellen wir *Q-Graph* vor, ein Open Source Graph System zur Verarbeitung nebenläufiger Anfragen und dynamischer Graph Partitionierung. Q-Graphs anfragenbasierter Partitionierungs Algorithmus *Q-Cut* kann die Partitionierung zur Laufzeit anpassen. Im Vergleich zu statischen Partitionierungen können hierbei Laufzeitinformationen über Anfragen Lokalität und Arbeitslast einbezogen werden.

Außerdem wird eine Implementierung für das Kürzeste-Wege Problem vorgestellt. Evaluationen zeigen die Leistungsfähigkeit von Q-Graph und die Effektivität von Q-Cut. Messungen zeigen, dass Q-Cut die Ausführungszeit von Anfragen um bis zu 60% verbessern kann und in der Lage ist, die Partitionierung an sich verändernde Anfragen Lokalität und Arbeitslast anzupassen. Q-Cut übertrifft dabei Methoden welche Domänenwissen zur Partitionierung verwenden.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Graphs are a powerful abstraction that can represent many real-world problems, such as road networks, social graphs or the web graph. Today, large-scale graph problems are processed by distributed graph processing systems such as Pregel [MAB+10], PowerGraph [GLG+12], GrapH [MTLR16] and GraphCEP [MMTR16]. For parallel graph processing, graph partitions are processed by concurrently operating workers. Each worker processes a dedicated graph partition. Programs are implemented in a vertex centric way, algorithms describe the behavior of single vertices. Problems are solved iteratively by vertices processing and exchanging messages.

Many modern graph applications require multitenant online graph processing which can process multiple queries concurrently. Typical applications are online map services such as Google Maps or OpenStreetMap, social media platforms or online artificial intelligence services. Queries usually have a localized scope and only operate on a subset of vertices [DHES16] [XYQ+14]. The number and locality of queries can change over time.
These graph applications operate on large shared graphs, partitioned across many workers. Query locality can be used to reduce communication and synchronization overhead. Therefore suitable graph partitioning is a very important factor for a good system performance and low query execution time.

We identified three major requirements for good graph partitioning in online graph processing systems: (i) *Locality* aiming to maximize the number of vertices for a query that are on the same machine. (ii) *Workload balancing* aiming to evenly distribute workload to avoid stragglers and idling workers. (iii) *Dynamic repartitioning* aiming to adapt partitioning to changing queries.

In this thesis we present *Q-Graph*, a multitenant online graph processing system and *Q-Cut*, a dynamic query-scope based graph partitioning algorithm for Q-Graph. The Q-Graph system is open-source and publicly available on GitHub[1] with additional documentation.
Q-Graph can continuously process multiple queries in parallel without queries interfering with each other, using a hybrid barrier synchronization approach that exploits local queries and reduces remote communication. Q-Cut improves graph partitioning at runtime optimizing both locality and workload distribution. It detects changes in query workload and locality and repartitions accordingly. No domain knowledge or user interaction is needed.

[1] `https://github.com/jgrunert/ConcurrentGraph`

In this thesis, we provide the following contributions:

- A fully functional distributed *open-source* system for low-latency parallel query processing.
- A novel *hybrid barrier synchronization* approach to allow both independent query synchronization and global synchronization for system wide operations.
- An adaptive partitioning algorithm denoted as *Q-Cut* to optimize vertex locality and workload distribution for better performance and lower query latency.
- An algorithmic formulation and case study of *shortest path* and *node search* queries.
- A detailed discussion of system implementation and optimization details.
- *Evaluations* showing the systems overall performance, scalability and the effectiveness of Q-Cut partitioning.

The thesis is structured as follows. At the beginning, Chapter 2 gives an introduction to graph processing systems and related work. In Chapter 3 we introduce the Q-Graph system, followed by Chapter 4 introducing Q-Cut. Chapter 5 presents a case study implementing the *shortest path* and *node search* algorithms. Evaluations based on the case study are shown in Chapter 6. Finally we give a conclusion and an outlook in Chapter 7.

Chapter 2

Background and Related Work

Q-Graph's basic computation model is based on the vertex-centric computation model, originally introduced by Google's Pregel system. Pregel is based on the Bulk synchronous parallel (BSP) [Val90] model for parallel algorithms. In this chapter, the fundamentals of BSP and vertex-centric graph processing systems are explained.

2.1 Bulk Synchronous Parallel (BSP)

BSP computations consist of a series of iterations, so-called *supersteps*. Figure 2.1 illustrates a BSP superstep. Each superstep has three phases: (i) The *local computation* phase, where workers can execute isolated compute functions in parallel. (ii) The *communication* phase, where workers exchange information for the next superstep. (iii) The *barrier synchronization* phase, after finishing communication, to ensure that all workers finished the superstep. A superstep is comparable to an iteration in the well-known MapReduce system [DG08], however a computation usually consists of a series of supersteps.

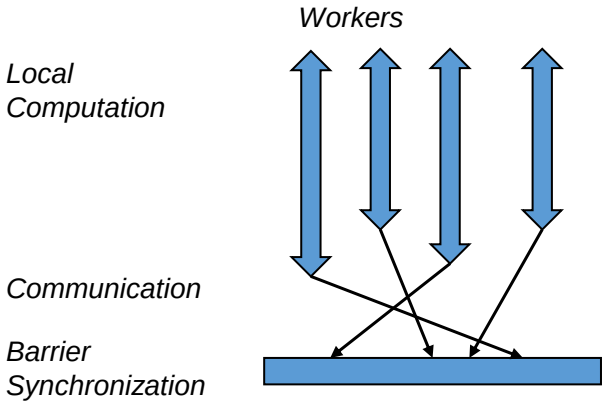


Figure 2.1: Bulk synchronous parallel computation model

2.2 Pregel

The concept of vertex-centric graph queries, which is also used by Q-Graph, was introduced by Google's Pregel framework [MAB+10]. In Pregel, algorithms are implemented using the vertex-centric API ("think-like-a-vertex"): the application programmer defines a vertex function to be executed in parallel by the graph vertices.

Vertices and edges have values, vertex values can be modified at runtime. There is one machine coordinating the system and query execution, called *master*. Query processing is done on *workers*, which process query-specific vertex functions on the vertices of their assigned graph partition. In each superstep, all active vertices execute the vertex function. When computing, a vertex can process incoming messages, modify its state and send messages to other vertices. In contrast to the BSP model, vertex messages can be sent during a workers local computation, however the destination worker will not process these messages before the next superstep. At the end of a superstep, workers perform a barrier synchronization before they start the next superstep.

Vertices are activated either explicitly by the application programmer or by receiving a message. By default, all vertices are active in the first superstep. A query terminates automatically when no more vertices are active. After termination the results are collected by the master and returned to the user.

Pregel processes queries in an offline fashion. There is only one active query at a time which has a global barrier synchronization at the end of every superstep. It is also not possible to change the graph partitioning at runtime. In contrast, Q-Graph allows processing of multiple concurrent queries, selective barrier synchronization and dynamic repartitioning.

2.3 PowerGraph

PowerGraph [GLG+12] is a large-scale graph computation framework, tailored for real-world graphs such as social networks or the web graph. It allows parallel queries using a shared-memory design. PowerLyra [CSCC15] is an extension of PowerGraph. It can exploit the locality of vertices and has a partitioning algorithm that can use different approaches for different types of vertices.

PowerLyra also uses partitioning methods exploiting locality on vertex or edge level. Q-Graph also uses locality based partitioning but based on query locality.

2.4 Seraph

Seraph [XYQ+14] [XYHD16] is a graph processing system allowing job-level parallelism. Multiple jobs can be run concurrently and share graph data. It also offers fault tolerance by using a copy-on-write semantic isolating graph mutations of queries.

2.5 GPS - Graph Processing System

Graph Processing System (GPS) [SW13] is an open-source distributed graph processing system. It is based on the networking framework Apache MINA and the distributed file system HDFS. The architecture is based on Pregel's ideas but it adds new contributions. It allows global computations for more efficient and complex algorithms. Also dynamic repartitioning at runtime on vertex level is supported.

GPS was an inspiration during the development of Q-Graph. Q-Graph adopted the idea of global computations for more control over the computation. In Q-Graph, vertices can also be moved but the repartitioning of Q-Cut uses a different approach. In GPS, repartitioning is done on the workers and on vertex level. Q-Cut uses centralized query knowledge at the master for repartitioning decisions.

Chapter 3

Q-Graph Multi-Query Graph Analytics

Q-Graph [1] is a distributed multi-query graph analytics system, implemented in Java. Similar to other graph processing systems such as Pregel or GPS, algorithms are implemented using a vertex-centric API where the application programmer defines a vertex function to be executed in parallel by the graph vertices.

In Q-Graph there is one master machine and multiple worker machines. The master starts and coordinates the query execution while workers perform the query computation. All machines are connected to each other and communicate via messages. The system can process multiple graph queries concurrently by running each queries with isolated state and messaging.

In this chapter we give an overview of the Q-Graph system. First, Section 3.1 explains assumptions we made for the system, followed by Section 3.2, giving a short overview over the system architecture. In Section 3.3 query processing and the hybrid barrier synchronization is described. Then, in Section 3.4, the communication is described in detail. Section 3.5 shows how queries are executed locally to improve the computation performance. Section 3.6 describes the handling of global query state. Finally, Section 3.7 explains statistics that are measured by the system.

3.1 Assumptions

For the system, we make some fundamental assumptions:

- **No graph modifications:** No modifications of the underlying graph are supported. No edges or vertices can be deleted or added. However the vertex state can be changed at runtime.
- **Single-threaded workers:** Each worker has a single computation thread. There are separate messaging send and receive threads which are decoupled by message queues.
- **Reliable FIFO communication:** Communication is done via reliable, First In – First Out ordered messaging using TCP sockets.

[1] `https://github.com/jgrunert/ConcurrentGraph`

3.2 Architecture Overview

The system architecture has two key components: one master and multiple workers. The master has a centralized global view and controls query processing. Workers perform the query computation and messaging.

All machines are connected with each other using TCP sockets. Figure 3.1 gives an overview over the systems communication architecture. Each machine is connected with all other machines with a dedicated channel. The TCP sockets ensure reliable and ordered communication which is important for maintaining global consistency and synchronization.
The user can start queries and get query results via the master API. There is no communication between the user and workers. The master communicates with workers to start queries, coordinate supersteps and retrieve results and statistics. Workers communicate with each other to exchange query information or to perform synchronization. Messaging protocols are explained in Section 3.4 in detail.

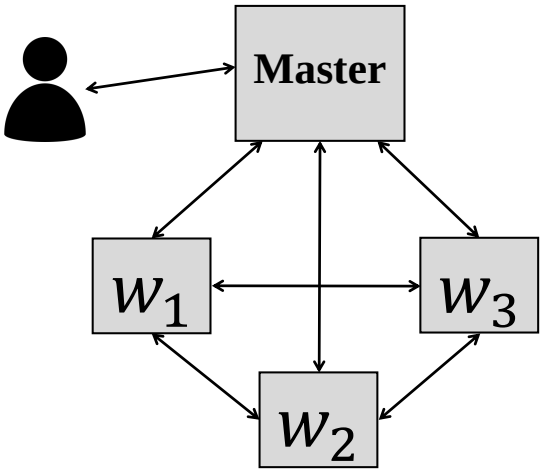


Figure 3.1: Communication architecture

The query computation is distributed across the workers. For each superstep, workers will execute the computation routine on all active vertices. A query considers a vertex as active if it received a message during the last superstep of if it was activated explicitly. Vertices can be activated inside the computation routine or by the global state management, as explained in Section 3.6.
The computation routine is user defined and implements the vertex-centric query algorithm, similar to other graph analytics systems. During the computation routine, vertex messages are sent to other vertices. If these vertices reside on the same machine, the message is transmitted directly. Otherwise a remote vertex message is sent. However not every vertex message is sent with a single network message but batched with other messages if possible, as explained in Section 3.4.
When a worker finished computing of all vertices it will barrier synchronize with the other

workers. Upon finishing the synchronization it will notify the master that it is ready for the next query superstep.

All operations are coordinated by the master. Figure 3.2 shows a simple example of a system startup with the execution of one query. First, the master and all worker machines are started. The master configures and initializes all workers and assigns the initial partitions. Workers load vertices data and finish startup, then they notify the master that they are ready.
Once all workers are ready, the master can start execution of queries. Fist it sends query data to all workers which then start calculating the first superstep. After each superstep, workers perform a barrier synchronization and notify the master once a superstep is finished. The master explicitly starts the next superstep by sending a control message to all workers. Query computation and barrier synchronization is described more detailed in Section 3.3. When the worker received superstep finished from all workers, it starts the next superstep. A worker informs the master when it has no more vertices active for a query.
When all workers have no more active vertices, the query is finished. Then the master orders all workers to send query output and statistics. The master does the final query evaluation and returns all query results to the user.

At runtime the master can use its global view to coordinate the system. It can limit the number of active queries or establish global barriers for repartitioning. Repartitioning is described in detail in Chapter 4.

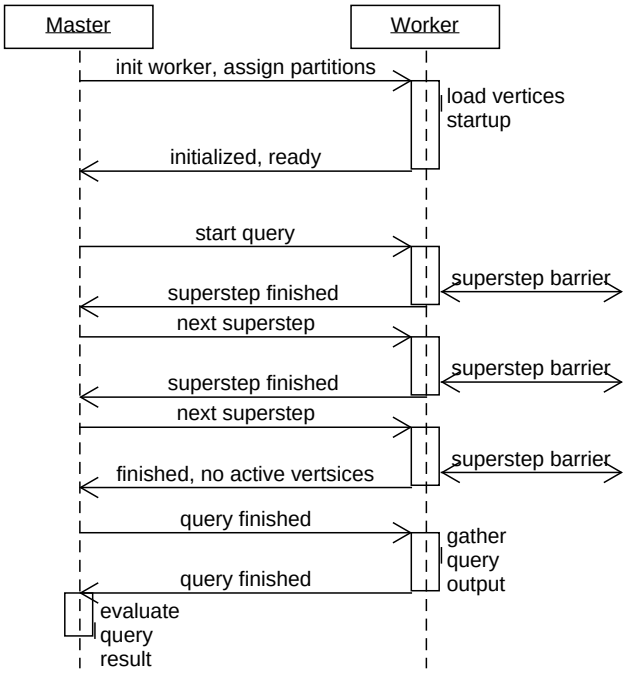


Figure 3.2: Query execution example

3.3 Multi-Query Analytics

Graph processing systems usually allow the execution of one query at a time. In Q-Graph it is possible to process multiple queries concurrently. To achieve this, all query states and messages are isolated for each query. An active vertex has an isolated state for each query it is active in. Vertex messages and superstep barriers are independent as well.

Independent query barrier synchronizations allow efficient processing of heterogeneous queries. Workers will perform a query local barrier sync as soon as all workers are ready for the next query superstep. When all workers have finished the computation of a superstep they will send a barrier sync message to all other workers through the connecting FIFO channels. Once a worker received all barrier syncs it will notify the master that its barrier sync is finished. When the master received from all workers, that they finished barrier, it will start the next superstep.
We use global barriers to perform operations that need global consistency and synchronization. This is used for performing vertex move operations in order to implement Q-Cut repartitioning. To start a global barrier, the master postpones the start of query superstep until all queries have finished their current superstep. Then it sends all workers a message to start a global barrier.
Figure 3.3 shows this concept of hybrid barrier synchronization. First, queries have independent query local barriers which allows different superstep lengths. Then the master establishes a global barrier to perform repartitioning.

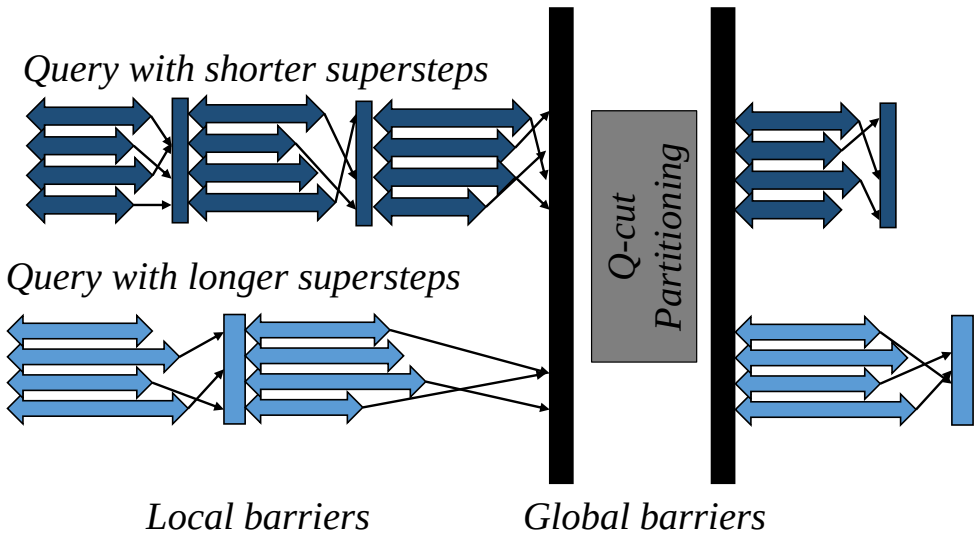


Figure 3.3: Hybrid Barrier Synchronization

It is also possible to skip barrier synchronization with workers that have no active vertices for a query. If there is only one worker active for a query, which is the goal of Q-Cut partitioning, a query can run completely locally on a worker. This concept of local query execution is explained in Section 3.5.

There are various advantages arising from multi-query processing. The system can continuously start new queries, no restart of the system is required to start additional queries. Furthermore parallel queries can improve worker utilization. Additionally it can further improve the effect of increased partitioning locality, as done by Q-Cut. Q-Cut tries to keep queries local on worker machines. However if there is only one query at a time, keeping this query local would make the system compute on only one worker. In a perfect scenario with n workers and n perfectly local queries, each worker would execute a query completely local without any remote communication or synchronization.

3.4 Messaging

Efficient messaging is a key component of Q-Graph. It is crucial for efficient query execution and low latency. All machines, i.e. master and workers are connected to each other with reliable, ordered TCP channels. The messaging layer of Q-Graph directly uses Java TCP sockets without any additional layer between to minimize messaging overhead.
Messages are sent asynchronously to further improve performance. There is a dedicated sender and receiver thread for each channel connecting two machines. Messaging threads are decoupled from the compute thread by message queues. All message types are sent through the same channels. However there are differences in the message protocol which are described in Section 3.4.2

3.4.1 Messaging Architecture

Figure 3.4 shows the messaging architecture of a worker machine. All messages are sent from the compute thread - control messages as well as vertex messages. Instead of sending messages directly, they are inserted into the message queue of the sender thread for the destination machine. The compute thread can then continue with its execution while the sender thread handles serialization and sending of the message.
Receiver threads are continuously listening on incoming TCP sockets. When a message is received, it is deserialized and inserted into the compute threads message queue. The compute thread periodically processes its message queue, usually between the execution of two supersteps.
The master has a similar architecture but the compute thread is replaced by the master thread, executing master tasks such as query starting, coordination or output evaluation.

3.4.2 Message Types

There are three fundamental message types with different requirements. *Control messages* are used for coordination between machines. Control messages are relatively small and are sent in smaller numbers but there are many different control message types. *Vertex messages* are sent by vertices during communication to transmit information to other vertices. A vertex message can be directed to a local vertex or a vertex on another machine. Vertex messages

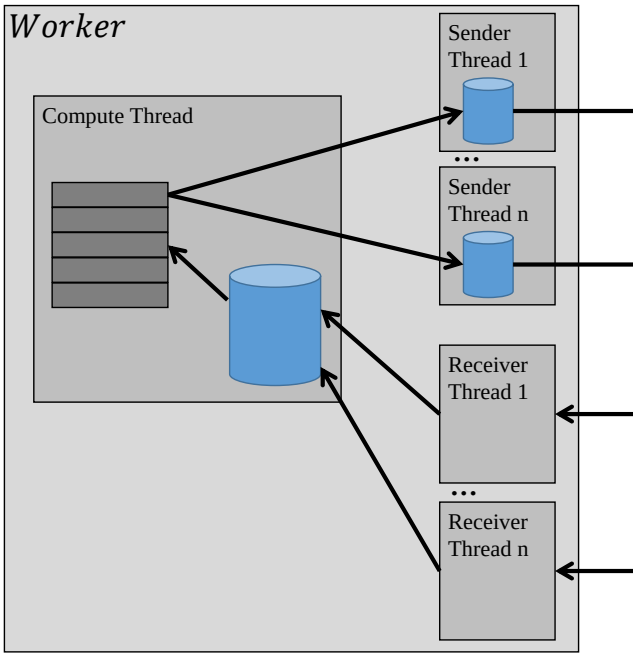


Figure 3.4: Worker Messaging

are relatively small but there are large numbers of vertex messages sent in each superstep. *Vertex move messages* are sent for repartitioning. After the master decided to move vertices for Q-Cut repartitioning, workers send vertices to their new partition machine. This happens less frequently but usually a large number of vertices is sent at once.

Control Messages

Control messages are used for all kinds of coordination and synchronization. They are exchanged between all types of machines, master and workers. As there are a lot of different control message types and the importance of extending or modifying the control message protocol, control messages are serialized using Google Protocol Buffers [2]. Protocol Buffers is a language and platform-neutral serialization library. Message structures are defined in a special language and then automatically compiled to the target language.

In the main thread, control messages are built using the utility class *ControlMessageBuildUtil*. This packs the information into the so-called *MessageEnvelope*, a Protocol Buffers message containing the actual information. This packed message can then be inserted to the outgoing message queue of the destination machines sender thread from where it is sent to the destination machine. There it is unpacked and later processed.

[2] `https://developers.google.com/protocol-buffers/`

There are three types of control messages: Master-Worker, Worker-Master and Worker-Worker control messages.

Master-Worker Messages:

- **MasterWorkerInitialize:** Assigns the initial partition to a worker and triggers workers initialization.
- **MasterQueryStart:** Starts a given query on this worker.
- **MasterQueryNextSuperstep:** Signals a worker to start the next superstep of a query. It also contains information about how to process a superstep and whether the next superstep runs locally or in a distributed fashion, on this or on another worker.
- **MasterQueryFinished:** Sent when all workers are finished with a query. Requests works to send all information of the finished query for the final evaluation.
- **MasterStartBarrier:** Commands the worker to establish a global barrier as soon as possible. Also contains information about the barrier tasks, such as vertex moving.
- **MasterShutdown:** Signals the worker to shutdown.

Worker-Master Messages:

- **WorkerInitialized:** Signals master that the worker is initialized and ready for query processing.
- **WorkerQuerySuperstepFinished:** Tells master that a superstep is completed on this worker. All vertices are computed, all barrier syncs were received.
- **WorkerQueryFinished:** Sent when a query was finished on a worker after receiving a *MasterQueryFinished* message. Also transmits information about the query results.
- **WorkerQueryVertexChunks:** Sends information about calculated query intersections, used for Q-Cut partitioning.

Worker-Worker Messages:

- **WorkerQuerySuperstepBarrier:** Message sent through a channel to signal the end of a superstep. After this message, no other message of the finished superstep will be sent through the channel.
- **WorkerBarrierStarted:** Informs other workers that this worker started establishing a global barrier.
- **WorkerBarrierReceiveFinished:** Notifies other workers that the information sending during the global barrier is finished.
- **WorkerBarrierFinished:** Sent when this worker finished the global barrier.

Vertex Messages

Vertex messages are simple messages for fast exchange of information between vertices during supersteps, as defined by the *Bulk synchronous processing model* [Val90]. Naturally vertex messages are only sent between workers. Active vertices send information to other vertices during their computation. Vertex messages from one superstep will be processed in the next superstep. All vertices receiving messages will be activated in the next superstep.
If the messages destination vertex is on the same machine, it can be delivered directly. No serialization and network communication is needed. In contrast to the sending of a remote vertex message, this is much cheaper in terms of performance.

In each superstep, a large number of vertex messages is sent. Depending on the algorithm and graph, this can be multiple messages per active vertex and superstep which can lead to thousands of messages.
Therefore vertex messaging is a crucial part of the systems performance and can easily become the performance bottleneck. Q-Cut aims to reduce the number of remote vertex messages to a minimum. However an efficient vertex message transmission is still very important.

There are several optimizations to reduce the overhead of remote messages. Vertex messages don't use any additional serialization or messaging layer. Protocol Buffers offers good performance for smaller number of messages but for a large number of small messages it causes additional CPU and network overhead. Vertex message data is written directly into byte buffers.
Furthermore individual vertex messages are not sent separately. Workers try to collect larger vertex message batches. For a query, the worker collects messages in lists, for each other worker individually. A batch is sent as soon as it reaches the lists maximum capacity. Batches are also flushed when a superstep is finished.

Vertex Move Messages

Vertex move messages are custom messages to send vertices from one worker to another. This is done during a global barrier when the master decided to repartition. At that point, all workers finished the last superstep of all queries and did not start any next supersteps.

Vertices are not sent separately but batched to larger messages, similar to vertex messages. A vertex move message contains all information needed by the destination worker work with a vertex, starting from the next superstep. This includes:

- ID of the vertex
- Static vertex information.
- All vertex edges.
- State of the vertex in all non-finished queries it was active in.
- All queries where the vertex will be active in the next superstep.

- All vertex messages received for the next superstep.

3.5 Local Query Execution

Remote communication and synchronization can reduce the overall performance. Therefore it is desirable to reduce it to a minimum by increasing the locality of vertices and exploiting the locality. Q-Cut tries to optimize the locality by repartitioning the graph over worker partitions. When vertices are on the same machine, vertex messages can be exchanged much cheaper by local message transmission.
When the system is well-partitioned, all vertices active in a query superstep are on one or few worker machines. The best-case partitioning is when all supersteps of a query can be executed on a single machine.
In a traditional graph processing system, query supersteps are synchronized on all workers, regardless of any locality. In Q-Graph, the master uses its global knowledge to detect possible optimizations. There are two techniques to exploit the locality of vertices.

Barrier skip allows to skip the synchronization with inactive workers in the next superstep. This technique is used if more than one but not all workers are active in the next superstep. The master notifies all active workers as usually, that they can start the next superstep. Together with the next superstep message, it sends the list of workers to synchronize with. Workers will ignore inactive workers and only synchronize with other active workers.
For the next superstep the master will determine again, which workers are active and have to synchronize with each other. Worker-Worker synchronization can be reduced while Master-Worker synchronization is still needed at the end of a superstep.

Local execution is used when only one worker has active vertices for a query. This is the best-case as supersteps can now be executed with no remote communication and synchronization at all. Similar to *barrier skip*, the master uses its global knowledge to determine the number of active workers. If only one worker is active, it will tell this worker to execute the query locally, as long as possible. All other workers will go on standby for this query.
The single active worker will execute the query locally as long as possible. When a superstep is finished it can directly start the next superstep without any waiting. A worker will automatically stop local execution when a remote vertex message was sent. In that case, another worker will be active in the next superstep as well. After finishing the last local superstep, the worker will notify the master and the next superstep will be started as usually. This next superstep can be either a normal superstep, a superstep with barrier skip or again a local execution but on another machine.
It is still important to frequently interrupt local execution to process other queries. Otherwise other workers would have to wait with their queries until the local query is finished or local execution would stop.

Both techniques can improve the system performance significantly, when combined with good partitioning. The combination of Q-Cuts partitioning and local query execution can lead to major improvements, as shown in the evaluations in Chapter 6.

3.6 Query Global State and Logic

Besides the vertex state, there can also be data shared globally by all workers and the master. Usually this includes data about the query itself, such as start and end point of a shortest path query. It can also include additional shared data, depending on the use case.

The shared query data can be modified by workers during a superstep. After a superstep, the shared data values are sent to the master, where they are combined. The combined values are then sent back to all workers for the next superstep.

Activation of vertices can also be implemented depending on the global state. It is possible to activate certain vertices, all vertices or by default all active vertices. This can reduce the number of active vertices significantly.

All global data and data combiner behavior can be defined by the developer, similar to the definition of the vertex compute function. Thereby it is possible to design more sophisticated and optimized algorithms which can share information and coordinate globally. The shortest path case study, presented in Chapter 5 demonstrates this.

3.7 Statistics

Q-Graph collects a large number of statistics for two main reasons: Statistics are used at runtime for monitoring and measurements, such as Q-Cut optimizing the partitioning based on measured values. Secondly for system evaluation and debugging purposes. All evaluations presented in Chapter 6 are based on these statistics. Furthermore the statistics have proven very useful during development to identify performance bottlenecks or to find bugs.

There are three levels of statistics: Query level, worker level and global level. For each query, individual statistics are recorded for all supersteps. This includes statistics such as local and remote messages transmitted, time measurements, supersteps computed and query locality.
Worker statistics are recorded in a configurable interval. A worker statistics sample consists of aggregated query statistics of all supersteps during the sample interval as well as additional worker statistics. Worker specific statistics are worker times, active vertices and system measurements such as CPU load.
Global statistics are aggregated worker statistics samples. This can be averages, minimum or maximum values over all workers in the system.

Statistics can be plotted automatically using the library JFreeChart [3]. Plotting is configurable, statistics can be plotted on all three levels.

[3] `http://www.jfree.org/jfreechart/`

Chapter 4

Q-Cut Graph Partitioning

Q-Graph uses a dynamic repartitioning algorithm to optimize graph partitioning over workers, denoted as *Q-Cut*. Combined with Q-Graph's approaches to exploit query locality, suitable partitioning can increase the overall performance significantly.

There are three major partitioning optimization goals: *query locality*, *workload balancing* and *adaptivity*. Both are needed for a good system performance. If the query locality is low, a large number of remote messages and barrier synchronizations is needed. Especially for larger systems with many workers, communication and synchronization can become a bottleneck. We define the partitioning problem as "Minimize the summed number of vertices that are not assigned to the worker with largest query subset". This function represents the number of vertices communicating remotely and the number of partitions to synchronize. For example if all vertices of a query are on one machine it is zero.

However even when there is a high locality, if *workload balancing* is poor, the system can be slowed down drastically. An extreme case would be, if all vertices are on one machine. The locality would be perfect in this case but only one worker would be actively processing queries. In all cases with poor workload balancing, workers with fewer workload will idle while waiting for stragglers with higher workload.

Adaptivity is needed to react on changing query workload and locality. It is necessary to detect when query characteristics changed and the current partitioning is suboptimal. In that case a suitable repartitioning is needed.

In this chapter we will present different partitioning approaches used in *Q-Graph*. Section 4.1 shows static partitioning strategies which are available in Q-Graph as well. In Section 4.2 we introduce the Q-Cut for dynamic partitioning.

4.1 Static Partitioning Strategies

Q-Graph offers different static partitioning strategies. Regardless of the usage of Q-Cut, the system will always start with an initial static partitioning. There are three strategies available: *Default partitioning* uses directly the partitioning of the given input file, vertices are assigned to workers in the order as loaded. *Hashed partitioning* is a strategy where vertices are assigned to workers based on their hashed ID. *Clustered partitioning* assigns vertices to the closest point of user-defined set of points.

All these strategies have their individual advantages but they are a trade-off between locality and workload balance. Furthermore they are static and there is no possibility to react on changes at runtime.

4.1.1 Hashed Partitioning

is one of the most commonly used graph partitioning strategies where vertices are assigned to workers based on their hashed ID. It is also the default partitioning strategy of Pregel [MAB+10]. For N partitions, vertices are assigned to the partition *hash(ID) mod N*.

When using a good hashing function, this partitioning leads to perfect workload distribution for larger numbers of vertices, as all workers get the same number of vertices. However locality is not taken into account.

4.1.2 Clustered Partitioning

Clustered partitioning has two prerequisites: The user must select a set of cluster centers and the vertex data must support a distance function in order to compare vertex and cluster center distances. The partitioner will assign each vertex to the next cluster with the shortest distance between cluster center and vertex.

This approach offers a high locality as colocated vertices are usually on the same machine. Figure 4.1 shows the comparison between hashed partitioning and clustered partitioning. For localized queries, hashed partitioning cannot exploit this locality. When using clustered partitioning, the locality is much higher. However it is important to ensure good workload balancing. When clusters are defined only based on vertex locations, this can lead to poor balancing.

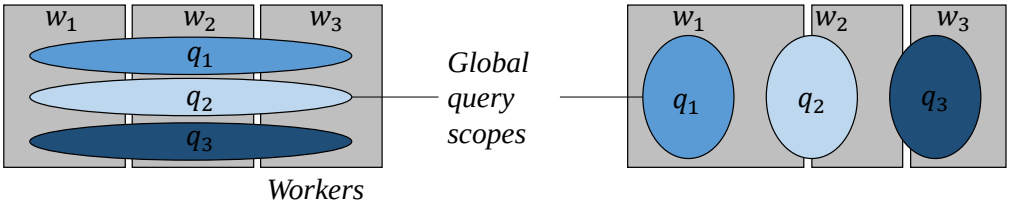


Figure 4.1: Query locality: Hashed(L): poor locality, Clustering(R): good locality

Clustered partitioning is especially useful in combination with domain knowledge. In the *shortest path* case study in Chapter 5, we present a clustering algorithm based on vertex GPS positions.

4.2 Q-Cut Partitioning

Q-Cut is a graph partitioning algorithm for Q-Graph. It uses real-time query knowledge for graph partitioning. In Q-Graph the master uses global knowledge about query scopes and workload for partitioning decisions. Q-Cut will operate on top of an initial static partitioning and repartition if necessary.

While a centralized view helps to make global decisions, centralized low-level knowledge is not scalable. Decisions on vertex or edge level would require Gigabytes of data. The graph portioning problem is NP-complete and impossible to solve for large-scale graphs in acceptable runtime. Therefore Q-Cut operates on query scopes. Instead of single vertices it analyzes sizes and overlaps of query scopes.

Figure 4.2 shows the basic principle of query based partitioning. Similar to location aware partitioning, colocated vertices are stored together, leading to higher locality compared to hashed partitioning. In contrast to clustered partitioning, query scopes are used to determine graph partitions. Q-Cut also enforces workload balancing.

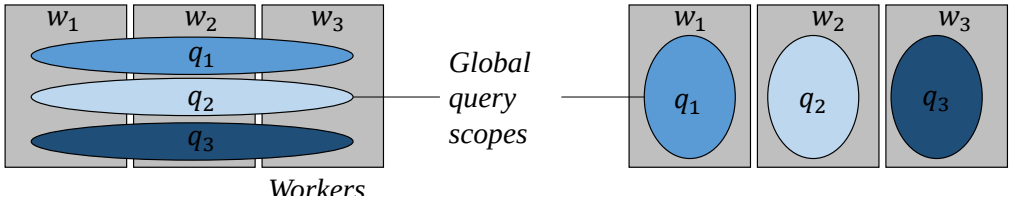


Figure 4.2: Query locality: Hashed(L): poor, Q-Cut(R): good locality and balance

In Figure 4.4 the results of Q-Cut partitioning are visualized. It shows the partitioning of a road network graph from the shortest path case study that is presented in Chapter 5. Colors represent the worker a vertex is assigned to. Initially the graph is partitioned using hashed partitioning with no locality at all. Step by step Q-Cut improves the locality of the partitioning while ensuring sufficient workload balance.

The repartitioning process consists of three steps, as illustrated in Figure 4.3. (i) Workers periodically collect statistics about query scopes and workload and send it to the master. This process is explained in Section 4.2.1. (ii) The master combines the received information to make a decision about repartitioning, as explained in Section 4.2.1. If repartitioning is required, (iii) the master instructs workers to implement the new partitioning by moving vertices, as shown in Section 4.2.3.

4.2.1 Query Statistics Retrieval

In order to make query scope partitioning decisions, the master needs to receive and combine local knowledge from workers and construct a global knowledge model. Each worker periodically collects various statistics and sends it to the master. Three statistics are especially important for Q-Cut: *Workload*, *current query locality* and *query scopes*.

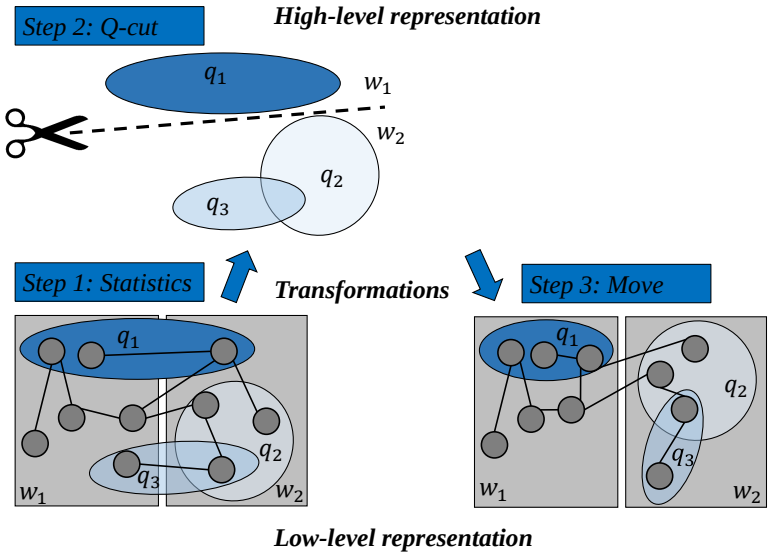


Figure 4.3: Q-Cut repartitioning steps

Workload is important to ensure evenly distributed query workloads across workers to avoid stragglers and idling workers. However it is difficult to have a workload metric that is up-to-date but robust against short-term oscillations. Therefore it is not possible to use the vertices active at a certain point in time. On the other hand, using the total number partition vertices could lead to imprecise results as some vertices could be more active while others are not activated at all.
In Q-Graph workers measure the workload by counting the number of vertices that were active in a sliding window with a configurable size, by default 60 seconds. This parameter is a trade off between stability and reaction speed and can be adjusted for different setups.

Query locality is used to measure the quality of the current partitioning. It is measured by counting the ratio of supersteps that have been executed locally. As explained in Section 3.5, queries are executed in local mode if all active vertices are on one machine. If the locality of the current partitioning is good, most of the supersteps will be executed locally. Query locality is also measured using a sliding window to avoid oscillations.

The most complex statistic are *query scopes*. For scalability reasons it is not possible to send alls vertices and their active queries to the master. However it is important to have detailed knowledge about query sizes and overlaps, on and across workers.
Figure 4.5 shows the basic principle of query scope evaluation. Each worker determines for its active vertices, in which queries a vertex is active in. From this low-level information it generates a set of query scopes. A query scope can include vertices that are active in one query or a set of queries. One vertex can be only in one query scope, the scope of all queries it is active in. The master receives the query scopes from all workers and rebuilds a model representing the actual query scopes as good as possible.
Determining the query scope for all active vertices can be expensive as it scales with the

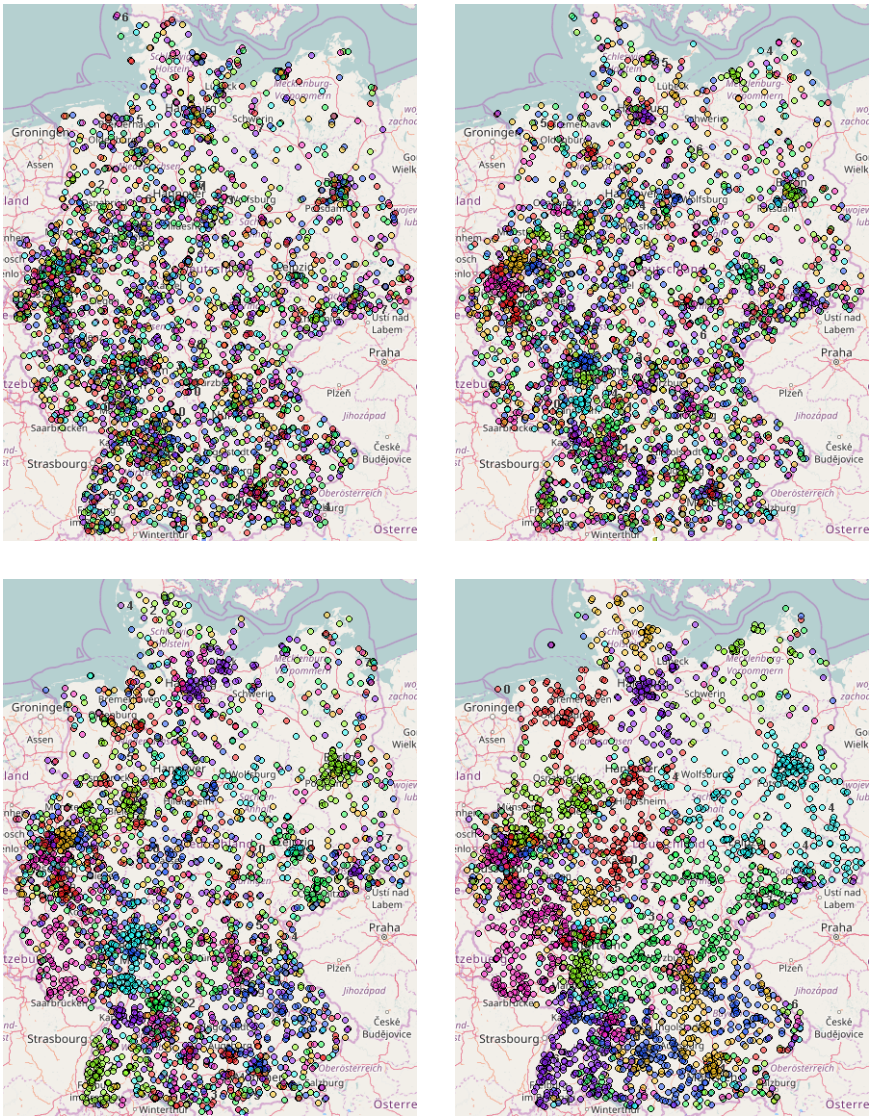


Figure 4.4: Increasing locality through partitioning over time

number of active vertices. To avoid this to become a performance bottleneck, workers use sampling. Instead of analyzing all active vertices, only a fraction is tested, by default 1/100. In the end all scope sizes are scaled up by the sampling factor.
After calculating the scopes, the worker sends the set of query scopes to the master. For each scope it sends the set of queries involved and the number of vertices in this scope. However for large numbers of queries with many partial overlaps, the number of scopes can be very high, with many small scopes. The number of scopes can quickly become too large for efficient sending via network or processing in later steps. To avoid this, smaller scopes are merged with other scopes they have a high overlap with. Scopes are considered to have a high overlap if they share a large fraction of their queries.

By now, only finished queries are used for query scope calculation. Tests have shown, scopes of active queries are highly volatile and can lead to unwanted oscillations. As Q-Cut aims to have a good long-term partitioning, it uses a history of finished queries. However finished queries are removed after a configurable time.

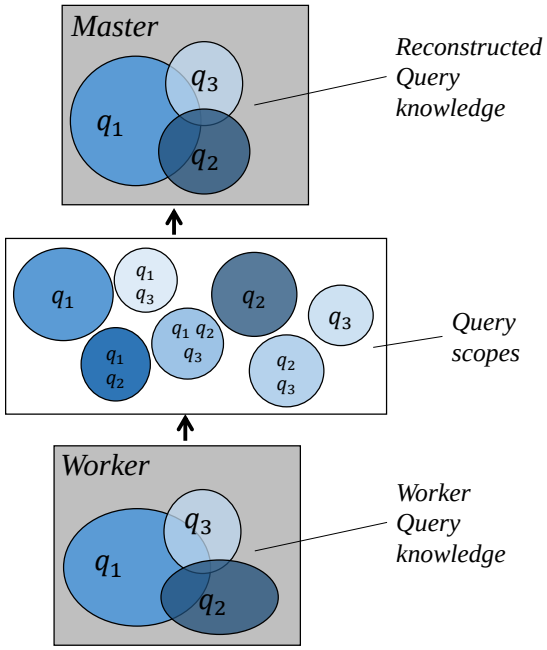


Figure 4.5: Global query knowledge: Master collects and reconstructs query information

All three statistics use configurable time windows to ensure stable measurements. Experiments have shown, that depending on the use case, jittering and oscillations can occur. Depending on the query type and user behavior different time windows may be appropriate. In general it is more important for partitioning decisions to have stable values and a larger history with more information than reacting on short-term changes.

4.2.2 Query-aware Partitioning

Based on the statistics sent by the workers, the master can make partitioning decisions. The goal of Q-Cut is to maximize query locality and acceptable workload balance. We define the locality cost function c_s as the sum over all queries, the number of vertices that are not on the worker partition with the largest query subset. Q-Cut aims to minimize c_s.
For workload balancing we define two functions: the maximum imbalance of a worker c_{imax} which is the maximum deviation from the average of a workers workload and c_{iavg}, the average of all workers workload deviations.

There are two decisions for the master to make: *If* repartitioning is needed and if yes, *how* to repartition. To determine *if* repartitioning is required, the master uses the workload balance

functions c_{imax} and c_{iavg} as well as the *query locality* metric. Both workload imbalance and locality can trigger a repartitioning. There are configurable thresholds for c_{imax} and c_{iavg}. If one value is above the given threshold, repartitioning is triggered.
For *query locality* based triggering, a threshold is not sufficient. Depending on the type of queries, the maximum possible locality can vary. Therefore the master uses the difference between the current locality and the locality before the last repartitioning. As long as the locality is improving, it is assumed that partitioning can be further improved. Once the metric converges, partitioning is stopped to avoid the additional overhead for repartitioning. When the locality drops this indicates a change in the systems usage and triggers repartitioning again.

If the master decided that repartitioning is needed, the actual *Q-Cut* algorithm is started in order to find a better way *how* to partition the graph across workers. Two of the previously calculated statistics are used by Q-Cut: *Workload* and *query scopes*. While the workload metric is only used to ensure, that a partition decision will not increase the imbalance over a threshold, query scopes are used to find a new partitioning with best possible locality.

The heart of Q-Cut is the *query locality model (QLM)*. This model represents the query scopes on all worker machines. For the initial state it is constructed from the *query scopes* that were sent by the workers. Figure 4.6 illustrates this process. For each worker, the model stores which query partitions are present and how many vertices are active.
The QLM can give an estimate about query locality and workload distribution. It also offers information about overlapping queries and how many vertices of a query are active on which worker. It provides the estimated cost functions for locality and workload balance: c_s, c_{imax} and c_{iavg}.
When Q-Cut tries to optimize the partitioning, any move operation is simulated by the QLM. After a simulated move the model is updated and all metrics are recalculated. Thereby a suitable search algorithm can estimate if a move operation improves the partitioning quality.

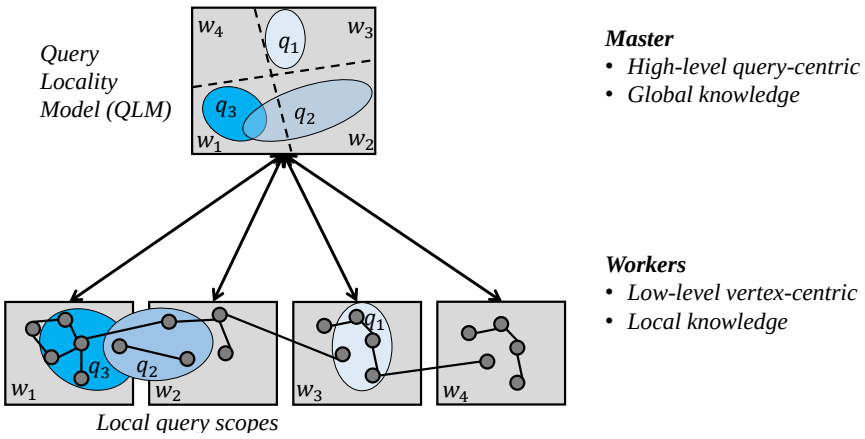


Figure 4.6: Query Locality Model from combined worker query scopes

Q-Cut uses the well-proven search method *iterated local search (ILS)*. ILS is a modification of traditional local search, where a perturbation method is used to escape local minima in the

cost function. We use ILS to find a solution for the partitioning problem minimizing the cost function c_s. Algorithm 1 shows the basic structure of the Q-Cut algorithm.

Algorithm 1 Iterated local search algorithm for Q-Cut partitioning.

```
1: state ŝ ← InitialSolution()
2: ŝ ← Balance(ŝ)
3: ŝ ← LocalSearch(ŝ)
4: while not Terminate() do
5:     s ← Pertubation(ŝ)
6:     s ← LocalSearch(s)
7:     if c_s < c_ŝ then
8:         ŝ ← s
9:     end if
10: end while
```

The initial state is given by the QLM built from the workers query scopes. Before the actual ILS, the *Balance* function is used to satisfy the balancing criteria, as shown in Algorithm 2. For the balancing criteria, c_{imax} and c_{iavg} must be below a configurable limit. While the balancing criteria is not satisfied, the smallest query scope is moved from the worker with highest workload to the worker with lowest workload.

Algorithm 2 Balancing function of Q-Cut

```
1: function Balance(State s)
2:     while not Balanced(s) do
3:         w_min ← MinLoadedWorker
4:         w_max ← MaxLoadedWorker
5:         q ← SmallestQuery(w_max)
6:         s.MoveQuery(q, w_max, w_min)
7:     end while
8: end function
```

After balancing the initial state, local search is applied for the first time. Algorithm 3 shows the functions pseudocode. The local search iteratively moves query partitions and evaluates the cost of the resulting partitioning using the QLM. If the resulting partitioning is better than the previous and the balancing criteria is satisfied, this partitioning is remembered. In each iteration, the result of the best move operation is chosen. Every partitioning that violates the balancing criteria is discarded If there is no improvement in an iteration, the local minimum is reached and the function returns.

In the first version the local search moved single queries. However this turned out to be hard to simulate in the QLM for overlapping queries. When query scopes with overlapping queries are moved this way, colocated queries can be split up onto different machines, leading to poor results. Therefore we introduced query clusters C. Queries are in one cluster if they have a large overlap in query scopes. Instead of moving single queries between workers, local search moves query clusters. Overlapping queries are kept together forming query scope clusters on

worker partitions. Experiments showed, that this approach leads to much better results.

Algorithm 3 Local search function of Q-Cut

```
1:  function LocalSearch(State s)
2:      terminated ← False
3:      while not Terminate() do
4:          l ←Successors(s)
5:          s' ← argmin_{s''∈l} c_{s''}
6:          if c_{s'} < c_s then
7:              s ← s'
8:          else
9:              terminated ← True
10:         end if
11:     end whilereturn s
12: end function
13:
14: function Successors(State s)
15:     best ← s
16:     for w_1, w_2 ∈ W, c ∈ C do
17:         if w_1 ≠ w_2 then
18:             s' ← Copy(s)
19:             s'.MoveCluster(c, w_1, w_2)
20:             if Balanced(s) ∧ c_{s'} < c_{best} then
21:                 best ← s'
22:             end if
23:         end if
24:     end for
25: return best
26: end function
```

When initial balancing and local search is finished, the actual ILS loop is started. In each iteration, the perturbation function is called. Afterwards local search is performed on the perturbed state. If the resulting state is not better than the previous state, it is discarded. Algorithm 4 shows the perturbation function. Similar to the local search, query clusters are used for move operations. The function picks a random cluster to move. Then it moves all query scopes of this cluster to the worker with the largest partition. As this step can result in an imbalanced state, the *Balance* function is called afterwards.

Although local search does not use workload balancing in its cost function, every single step of Q-Cut ensures that the workload balancing criteria is not violated. Before starting the ILS, a workload balance is established. In every following function, steps causing too high imbalance are discarded.

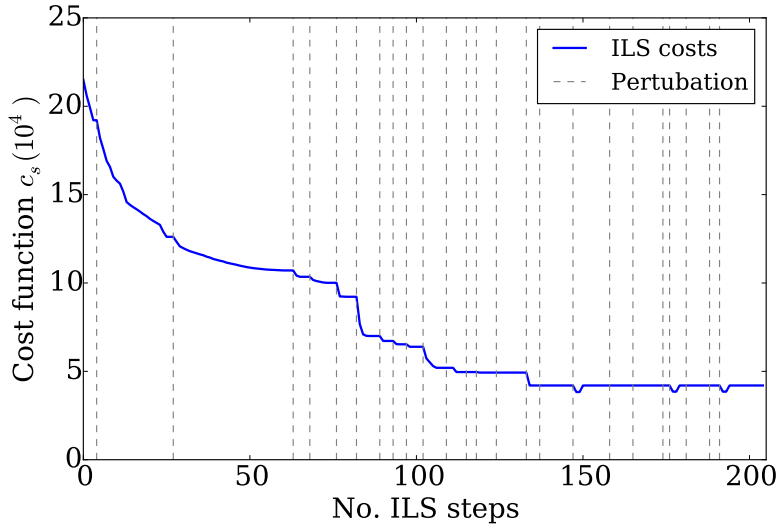
Section 4.2.2 shows an example for an ILS run on initial hashed partitioning. At first, the workload balance is very good but the locality cost function is high, as expected for hashed

Algorithm 4 Perturbation function of Q-Cut

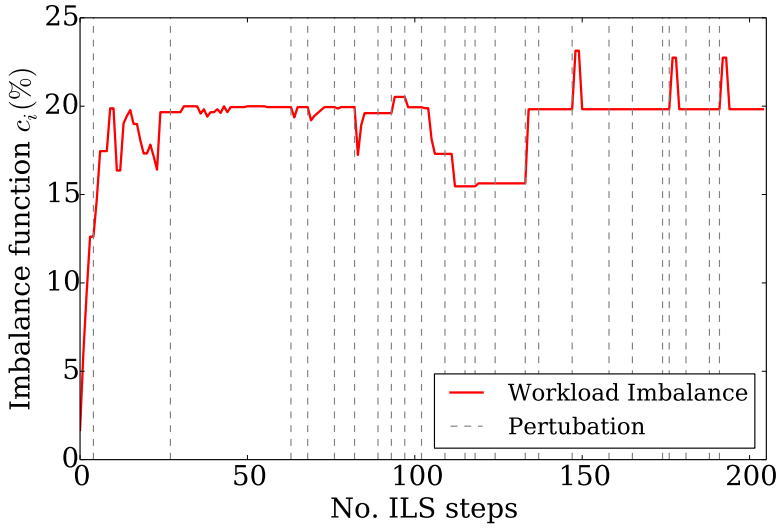
```
1: function PERTURBATION(State s)
2:     c_rand ← RANDOM(C)
3:     w_max ← WORKERWITHLARGESTPARTITION(c_rand)
4:     for w ∈ W do
5:         if w ≠ w_max then
6:             s.MOVECLUSTER(c, w, w_max)
7:         end if
8:     end for
9:     s ← BALANCE(s)
10: end function
```

partitioning. Then local search is started, quickly ending in a local minimum. After alternating perturbation and local search reduce the cost function significantly. However the average imbalance reaches the limit of 20% and limits the options for move operations. At a certain point, the cost function converged, it is not possible to perform move operations without exceeding the imbalance limit.

The algorithm terminates after a configurable amount of time or iterations. It runs asynchronously in a separate thread on the master machine and can be interrupted if necessary. When a new ILS partitioning decision is finished, the master will start a global barrier as soon as possible. The repartitioning decision is translated to a set of move operations that are then executed by the workers during the global barrier.



(a) Cost function during ILS iterations



(b) Workload imbalance function during ILS iterations

4.2.3 Vertex Moving

After a Q-Cut decision, the new partitioning must be implemented. First the new partitioning must be translated on the master to a series of move operations that are then performed by workers. The translation algorithm uses the *query locality model* to calculate all partitioning differences.
Figure 4.8 shows a simple example of this translation. Q-Cut decided to move the query scope q_2 from w_1 to w_2 and q_1 to w_1. This can be translated directly to move operations

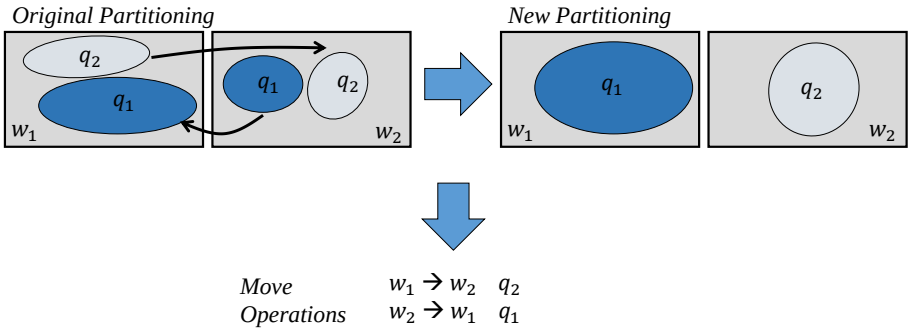


Figure 4.8: Simple move operation translation

However there are ambiguous cases where moved query scopes cannot be mapped to single queries. Figure 4.9 shows an example. Q-Cut moved the query scopes q_2 and $q_2 \cap q_3$ from w_1 to w_2. The remainder of q_3 and $q_2 \cap q_4$ should not be moved because they would not improve the locality or violate workload balancing.
Therefore we define a move operation as a set of queries Q_i for which all vertices are moved and a set Q_t with queries, that are tolerated for vertices to move. In this example, w_1 would move all vertices active in q_2 if they are not active in any other query except q_3. This method allows fine granular move decisions while having an acceptable run time.

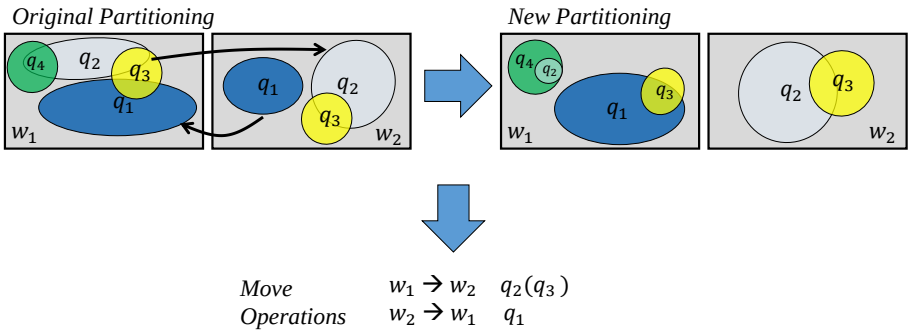


Figure 4.9: Complex move operation translation

After the translation to move operations, the master will establish a global barrier for the execution of vertex moving as soon as possible. When the next supersteps are finished by workers, the master postpones the next superstep instead of starting it immediately. When

all workers finished all supersteps, the system is ready for the global barrier. No supersteps are active at this point. This ensures consistency during and after moving vertices.

Figure 4.10 shows the sequence of a global with vertex move. The sequence has the following phases:

1. The master requests workers to start a global barrier and perform move operations during barrier this barrier.
2. After receiving a *barrier request*, a worker sends to all other workers a message that it started with a global barrier.
3. When a worker received *barrier started* from all other workers, it knows that the other workers are ready to receive vertices to move. If it has any vertices to send, it will collect all vertices to send to another worker, based on the move commands received from the master. All vertices to move are removed locally and then sent using a vertex move message, as explained in Section 3.4.2.
4. Once a worker received all vertices from all workers, it notifies the other workers that it finished receiving.
5. When all workers finished receiving, each worker processes the received vertices. They are added to the local graph partition and registered at the queries they are active in.
6. After all vertices are received and processed, a worker finishes the barrier. It sends a *barrier finished* message to all other workers.
7. Finally, when a worker received *barrier finished* messages from all other workers, it knows that all other workers and communication channels are finished with the global barrier. It sends a *barrier finished* to the master.
8. When the master received *barrier finished* messages from all workers, the global barrier ends. The normal execution continues, new supersteps are started by the master.

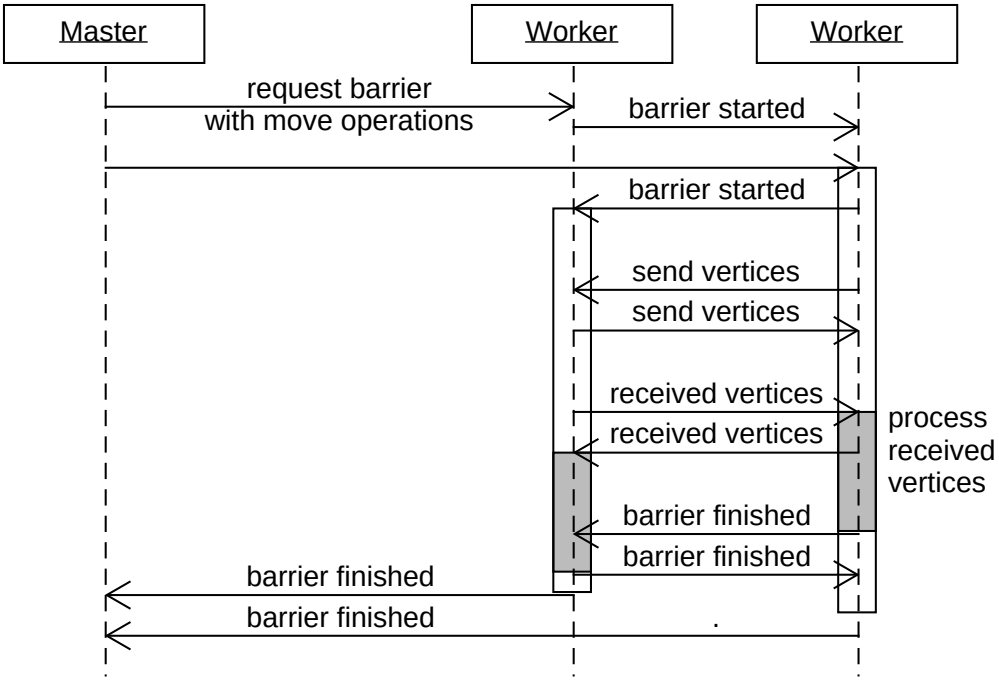


Figure 4.10: Sequence of global barrier with vertex move

Chapter 5

Case Study: Shortest Path Finding

The shortest path problem is a popular graph problem with a wide range of uses. Shortest path algorithms are a class of graph algorithms to find the shortest path between two points. However the most popular pathfinding algorithms such as the *Dijkstra algorithm* or *A** are single threaded algorithms. In this chapter we present a parallel shortest path algorithm for parallel computation on a distributed graph system. It is based on an algorithm presented in [RBVM15], which is inspired by the delta-stepping algorithm of Meyer et al. [MS03].

Section 5.1 explains the fundamentals of distributed and parallel pathfinding and presents our delta stepping based shortest path algorithm. Section 5.2 introduces a variant of this algorithm to find the next node matching a search criteria. In Section 5.3 it is shown how the algorithm is implemented in Q-Graph.

5.1 Distributed Shortest Path Algorithms

Traditional shortest path algorithms, such as the Dijkstra algorithm [Ski90] use a shared data structure like a priority queue to determine the next vertex to visit. This is problematic in a distributed graph system system with parallel workers and vertex-centric data. When using a global priority queue, it would be necessary to synchronize a shared data structure among all workers. Therefore it is necessary to use different approaches better suited for parallel, vertex-centric computation models.

5.1.1 Simple Parallel Shortest Path Algorithm

In the Pregel paper [MAB+10] a simple *Single Source Shortest Path* algorithm was proposed. It discovers all vertices and edges in a breadth first search, beginning at the start vertex. Initially all vertices are initialized with the distance *infinity*, the start vertex is initialized with 0. Vertices send on all outgoing edges the sum of their value and the edges weight. When a vertex receives a message it is activated. If it receives a message with a smaller distance than its current value, the value is updated. In each superstep, all active vertices send the sum of their own distance plus the outgoings edge weight to all neighbors. The algorithm terminates once all vertex distances have converged and no more vertices are active. An example for this algorithm is shown in Figure 5.1.

This simple algorithm can reliably find the shortest path in a parallel graph system but it will visit a large number of nodes, as there is no limit which nodes to visit and re-visit.

5.1.2 Delta Stepping based Shortest Path Algorithm

Several algorithms for parallel shortest path algorithms have been proposed. One proven technique is the delta-stepping algorithm [MS03]. It can find shortest paths on arbitrary graphs in parallel setups with $O(n + m + d * L)$ total average-case time. At delta stepping vertices discover their neighboring vertices in parallel. A distance limit, stepwise increased by the delta parameter, limits the search space to avoid discovery of unnecessary vertices.
The original algorithm was designed for the single source shortest path problem, however we use the same principle for the single-pair shortest path (SPSP) problem, finding the shortest path between two points. In [RBVM15] a delta stepping based SPSP algorithm to run on the BSP computational model of Apache Giraph was proposed.

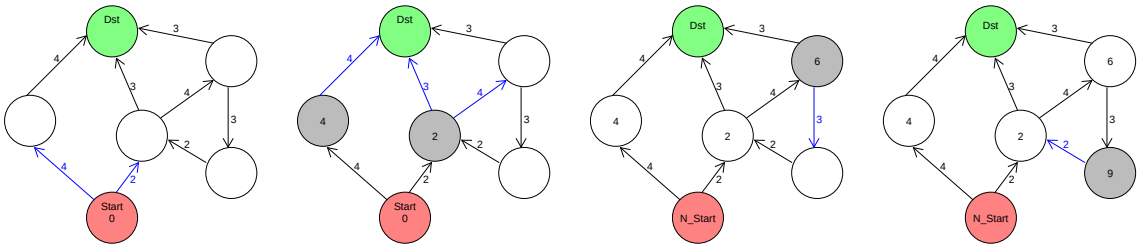


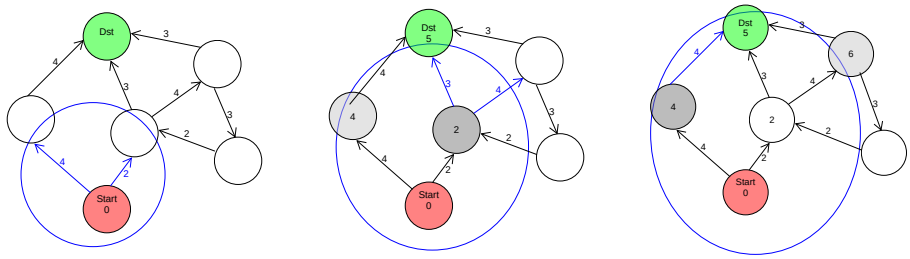
Figure 5.1: Simple shortest path algorithm

Our algorithm uses the same basic principle as the simple algorithm presented in Section 5.1.1. Beginning from a start vertex, vertices propagate their distance plus the outgoing edges length. However instead of discovering all neighbor vertices in a in a breadth first search fashion, a distance limiter prevents discovery of unnecessary vertices.
Figure 5.2 illustrates this principle. The distance limiter is increased stepwise. Once the algorithm discovered the destination vertex the distance limit is fixed to the current distance. The algorithm terminates when no more vertices are active, when all distances of all discovered vertices are minimal. Compared to Figure 5.1, which shows the same use-case, this algorithm discovers less vertices and terminates faster.

In detail, the algorithm has the following steps:

1. Activating the source vertex, start discovering neighbor vertices. Activated vertices send their current distance to neighbor vertices.

2. Continuously increasing the distance limiter and discover new vertices. When a vertex is discovered but has a distance larger than the limiter, it is suspended until the distance limit is larger. When a vertex receives a neighbors distance it checks if the sum of the distance and the neighbor edges length is smaller than its current distance. If it is smaller, it updates its own distance and activates itself.



(a) Start vertex (b) Distance limit 3 (c) Distance limit 5

Figure 5.2: Delta-stepping based shortest path algorithm

3. When the destination vertex is discovered, the distance limiter is fixed to the current value. No more vertices with larger distance will be discovered after this point. However the algorithm continues sending distances until all discovered vertices have the minimal distance and no more messages are sent.

4. After the search is finished, path reconstruction is started. Starting from the destination vertex, all vertices on the shortest path send a message to their predecessor. When the source vertex is reached again, all points of the shortest path were visited and the path can be outputted.

5.2 Point of Interest Search

The algorithm presented in Section 5.1.2 can be easily adapted to the problem to find the next point matching a given search criteria, starting from a source vertex. This problem reflects many real world problems such as "Finding the next Point of Interest (POI)", for example finding the next gas station, supermarket or a touristic attraction.

The *POI search* shares most of the logic with the shortest path algorithm. It has the same discovery and delta stepping logic. Instead of searching for a single destination point, the algorithm searches for a point matching the criteria, for example a POI tag. When a point matching the criteria is found, it is treated similar to the destination vertex in the original shortest path algorithm. The distance limiter is fixed and once the search is finished, the shortest path is reconstructed and returned.

5.3 Implementation in Q-Graph

In this section, the implementations of the shortest path algorithm from Section 5.1.2 and the POI search in *Q-Graph* are described. Both algorithms were designed to fit the vertex-centric execution model of Q-Graph. The basic implementation of the algorithms is similar to other

distributed graph processing systems. In addition the implementation makes use of more advanced features in Q-Graph such as shared global data and selective vertex activation.

As for every Q-Cut algorithm implementation, the logic is defined by two classes: The *vertex class*, defining the vertex state model and vertex-centric logic. The *query class* defines information about a requested query and global logic, as explained in Section 3.6.

Algorithm 5 illustrates the compute function, defining the vertex-centric logic. In the first superstep, only the source vertex is activated and broadcasts the distance which is 0 at this point. In contrast to most other implementations, where all vertices are activated in the first superstep, the global query logic only activates the source vertex. This can reduce the query execution time significantly, especially on large graphs.
During the search phase, all activated vertices check if a received message has a shorter distance than the current distance. If yes, it updates its own distance and predecessor and broadcasts its own distance. When a vertices distance is above the distance limit, it does not broadcast its distance. In the next superstep, it will be activated again until the distance limit is above the own distance.
As soon as the destination vertex is found, the global distance limit is fixed. This prevents that more vertices further away are discovered. The search phase will continue until no more vertices are active. Then it is guaranteed that the shortest path to the destination vertex was found.
After the search phase is finished, the global query logic starts the reconstruction phase and activates the destination vertex explicitly. Then all vertices along the shortest path send a message to their predecessor until the source vertex is reached and the algorithm is finished.

This algorithm can be easily adapted to the *POI search* problem. For the search phase, only Algorithm 5 Line 30 has to be modified. Instead of checking if the ID is the destination vertex ID, the algorithm checks if the vertex matches the criteria, using a vertex *tag*. The reconstruction phase has to start reconstructing from the closest matching point instead of the destination vertex. Besides these changes, no major modifications are necessary.

5.3.1 Graph Data Generation

For this case study we used publicly available OpenStreetMap (OSM) data[1]. OSM offers world wide data for waypoints and roads with GPS positions. Roads include additional information such as distances and speed limits.

We developed a converter[2] which transforms the raw OSM data into a minimal road network graph. Vertices represent junctions and edges represent connecting roads. The edge weight is the sum of all road pieces between two junctions divided by the speed limit.

This graph data was used during the development of Q-Graph and for the evaluations presented in Chapter 6.

[1] `http://download.geofabrik.de/`
[2] `https://github.com/jgrunert/SimpleOSM2Graph/`

Algorithm 5 Vertex compute function for shortest path algorithm

```
1: function Compute(superstepNo, messages, query)
2:     if superstepNo = 0 then
3:         if ID = query.src then
4:             BroadcastDistance(state.dist)
5:         end if
6:         VoteHalt()
7:         return
8:     end if
9:     if query.ReconstructionPhaseActive then
10:        if ID = query.src then
11:            query.Finish()
12:        else
13:            SendReconstructMessage(state.pre)
14:        end if
15:        VoteHalt()
16:        return
17:    end if
18:    minDist ← state.dist
19:    minPre ← state.pre
20:    for msg ∈ messages do
21:        if msg.dist < minDist then
22:            minDist ← msg.dist
23:            minPre ← msg.pre
24:        end if
25:    end for
26:    if minDist > query.DistLimiter then
27:        SuspendActivation(minDist, minPre)
28:        return
29:    end if
30:    if ID = query.dst then
31:        query.FixDistanceLimit(state.dist)
32:        VoteHalt()
33:        return
34:    end if
35:    if minDist < state.dist then
36:        state.dist ← minDist
37:        state.pre ← minPre
38:        BroadcastDistance(state.dist)
39:    end if
40:    VoteHalt()
41: end function
```

5.3.2 Domain Graph Partitioning

Based on the domain knowledge that can be extracted from given map data we developed a static partitioning method called *hotspot partitioner*. The hotspot partitioner is a clustered partitioner generating clusters around larger towns, assuming that more and localized queries will be started in these areas.
In addition to the road network graph, the user supplies a list of k largest cities on the given input data. The partitioner will then assign vertices to the closest hotspot cluster. These clusters can then be assigned to worker as their graph partitions.

Evaluations in Chapter 6 show that this partitioning can reduce query execution time in many scenarios. However it does not offer the generality and adaptivity of *Q-Cut*.

Chapter 6

Evaluation

In this chapter we show evaluations of the Q-Cut system showing the systems performance and scalability. First we explain the experimental setup in Section 6.1. It is based on the shortest path case study presented in Chapter 5. Afterwards we show the systems overall performance and the efficiency of the Q-Cut algorithm in scenario with constant query characteristics in Section 6.2 and for a changing query characteristics in Section 6.3. Finally we show in Section 6.4 the efficiency of the hybrid barrier approach and Q-Graph's scalability in Section 6.5.

The evaluations demonstrate that Q-Graph with the Q-Cut algorithm, combined with the hybrid barrier synchronization can improve query latency by up to 60%.

6.1 Experimental Setup

We used different scenarios for the *shortest path case study* on different hardware setups. Graph data was generated from OpenStreetMap data. A query generator was used to simulate different usage scenarios.

6.1.1 Computing Hardware

In the experiments we used three different computing setups:

- *compute cluster* with 12 nodes × 8 cores (3.0GHz) and 32GB RAM per node
- *notebook* Asus N53SV with 8 cores, i7 2630QM with 8GB RAM
- Amazon *EC2 cloud* instance m4.2xlarge with 8 × 1vCPU (Intel Xeon E5-2676 v3), 32 GB RAM and 2.4 GHz clock speed.

6.1.2 Graph Data

All evaluations were performed on road network graphs, generated from publicly available OpenStreetMap data. Road network graphs were generated by using the *OSM graph data generator* presented in Section 5.3.1.

Two different graphs were used for the evaluations:

- Road network graph of *Germany (GY)* with 11,805,883 vertices and 30,804,741 edges
- Road network graph of the German state *Baden-Wuerttemberg (BW)* with 1,802,728 vertices and 4,770,566 edges.

6.1.3 Queries and Computation Algorithms

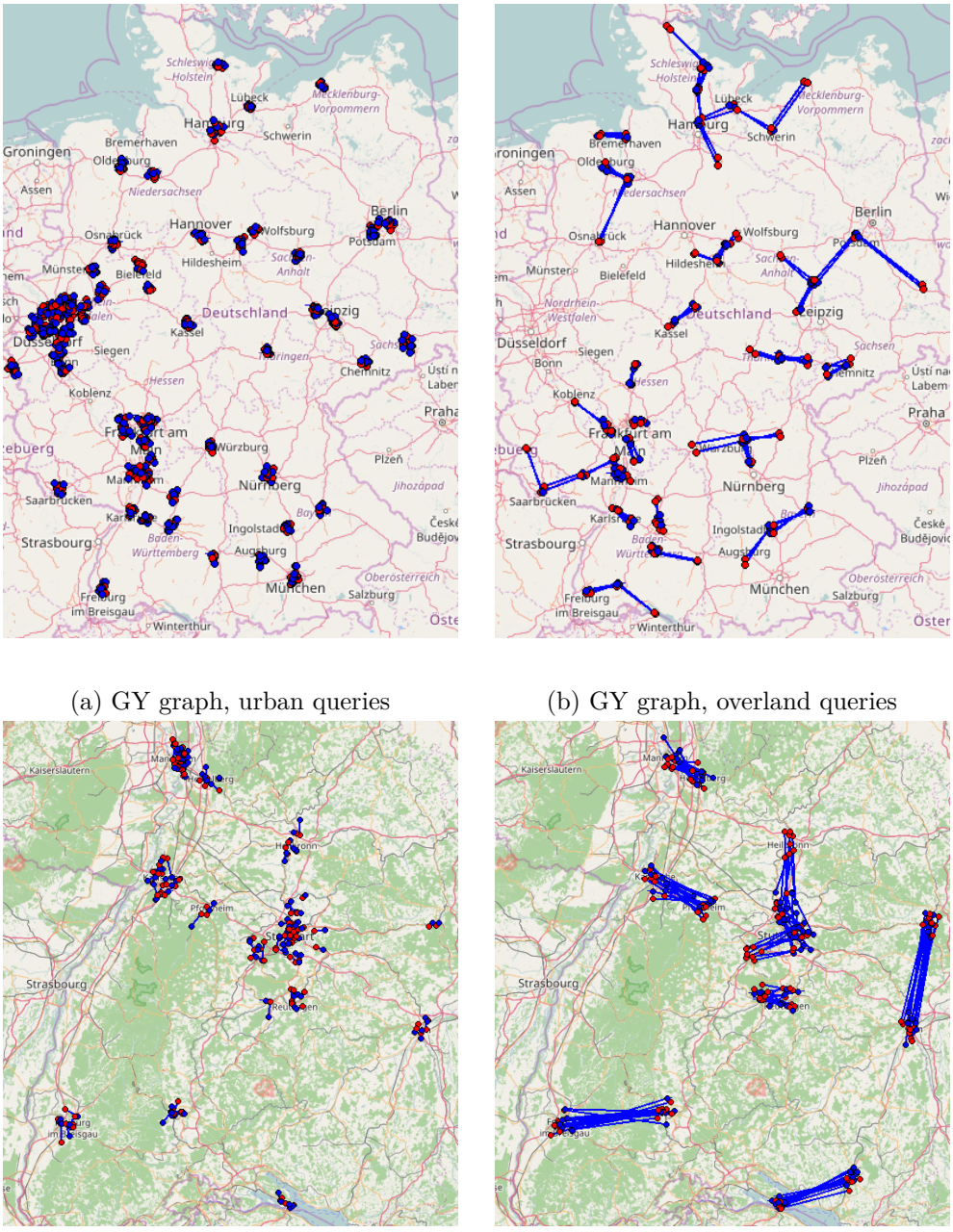
For the experiments we used both algorithms from the shortest path case study: *shortest path* and *POI search*. Queries for shortest path defined a start and end point, queries for POI search a start point and a search criteria.

Four types of queries simulating real world requests were generated, two for the GY graph, two for BW. All query types are visualized on an OSM map in Figure 6.1.

- *GY urban*: Queries in the 64 largest cities on the Germany OSM road network graph.
- *GY overland*: Overland queries connecting cities with a shift of focus towards east, compared to GY urban.
- *BW urban*: Urban queries for the 16 largest cities on the BW graph.
- *BW overland*: City connecting queries, similar to GY overland but on the BW graph.

6.1.4 Benchmark

In all evaluations we compare two initial static partitioning with and without Q-Cut: *hashed partitioning* and *domain partitioning*. In hashed partitioning the vertices are distributed across workers without considering colocation.
Domain partitioning is represented by the hotspot partitioner presented in Section 5.3.2. Vertices are assigned to partitions based on proximity to the next larger city. This partitioning method offers better locality but worse workload balancing.



(a) GY graph, urban queries

(b) GY graph, overland queries

(c) BW graph, urban queries

(d) BW graph, overland queries

Figure 6.1: Generated queries on OSM map

6.2 Q-Cut Partitioning

In this experiment we show the effectiveness of Q-Cut in scenarios with constant query characteristics. Two scenarios with different query types were tested: *Shortest path* queries for the *BW urban* dataset and *POI search* for *BW urban*. Both experiments were performed on the *EC2 cloud* setup.

For *shortest path* with *BW urban* queries, both domain partitioning and Q-Cut offer improvements over hashed partitioning. A combination of domain portioning and Q-Cut offers the best results. As Figure 6.2 shows, Q-Cut offers a large reduction of query processing time of 40% over an initial hashed partitioning. The combination of domain and Q-Cut partitioning reduces the processing time by 53%.

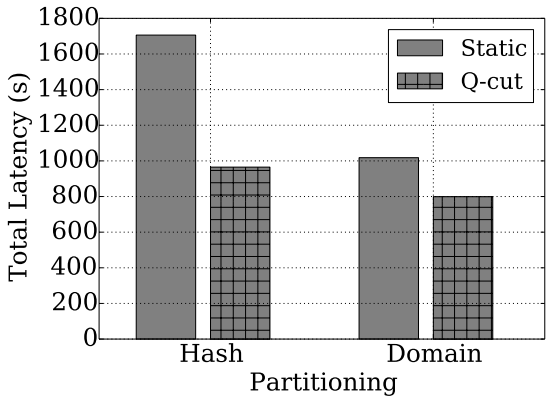
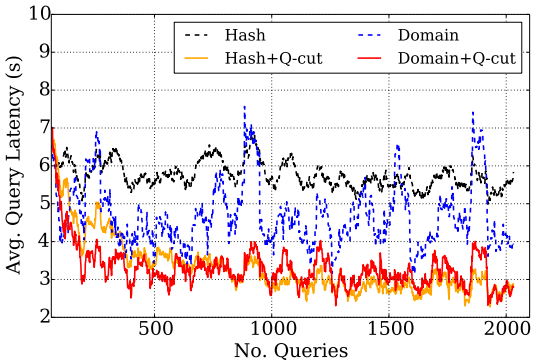
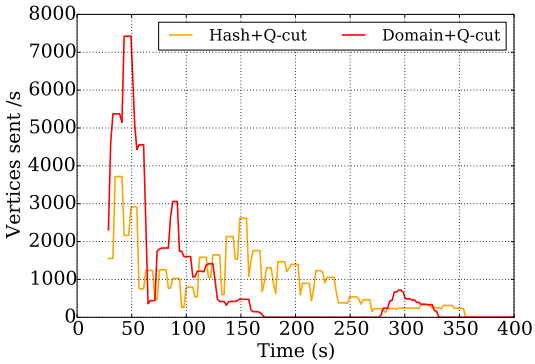


Figure 6.2: BW shortest path queries: Total query latency

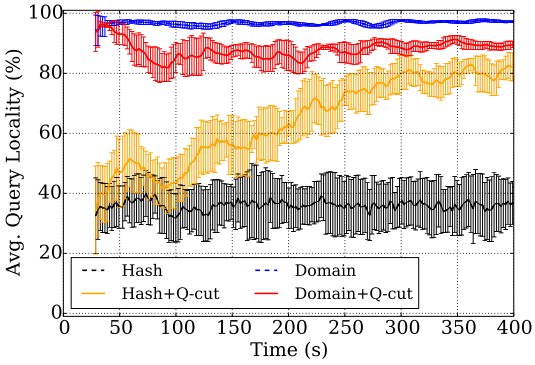
Figure 6.3 shows a selection of Q-Graph statistics, demonstrating how partitioning impacts the system. (a) clearly shows how the individual query latency is lowered by a good partitioning. After a short repartitioning phase, Q-Cut reduced the query latency drastically. (b) shows that there is an initial peak of vertices moving, decreasing to zero over time. This happens as the partitioning converges against its optimum and repartitioning is stopped when convergence is detected.
(c) and (d) show how repartitioning optimizes different factors for initial hashed and domain partitioning. For hashed partitioning it increases the initially low locality while maintaining workload balance. Domain partitioning is improved by decreasing the workload imbalance without lowering locality much. In both scenarios it can be observed that locality and workload balance converge against similar values, independent from the initial partitioning.
Repartitioning does not only increase the percentage of local supersteps, it also decreases the number of remote messages significantly, as shown in (e). Together with synchronization, this is a major bottleneck. While the number of remote vertex messages decreases, the amount of local messages increases. Compared to remote messages these are much cheaper as they don't need serialization and network communication.



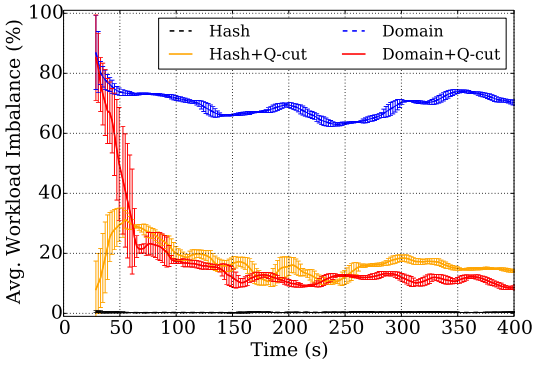
(a) Average Query Latency



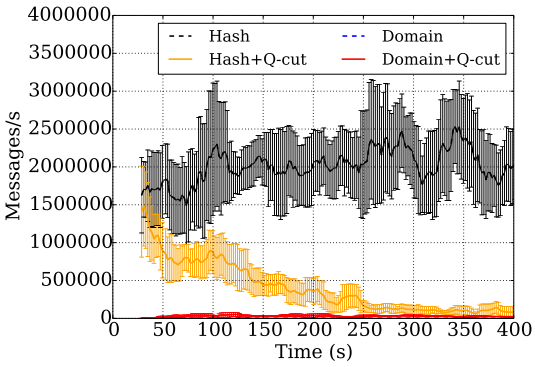
(b) Vertices moved



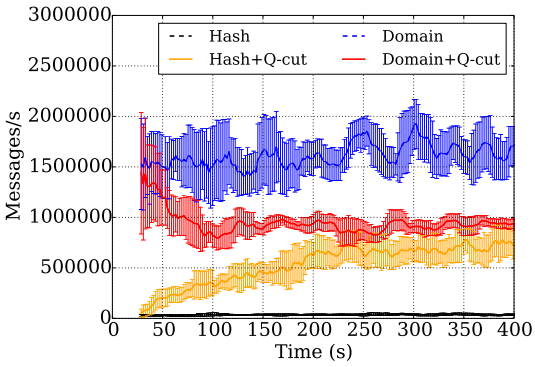
(c) Local supersteps ratio



(d) Average workload imbalance



(e) Remote messages sent



(f) Local messages

Figure 6.3: Statistics for BW shortest path queries

The second evaluation tests *POI search* queries for *BW urban*. In this scenario the improvement of Q-Cut is even bigger. As Figure 6.4 shows, both domain partitioning and Q-Cut improve the total query latency. The combination of domain partitioning and Q-Cut reduces the total query latency by 60%. Other experiments have also shown that longer queries increase the effect of Q-Cut.

Similar to the first evaluation, we can see in Figure 6.5 clearly how Q-Cut improves the initial partitioning. Both Q-Cut curves are close together, the curve with initial domain knowledge slightly lower.

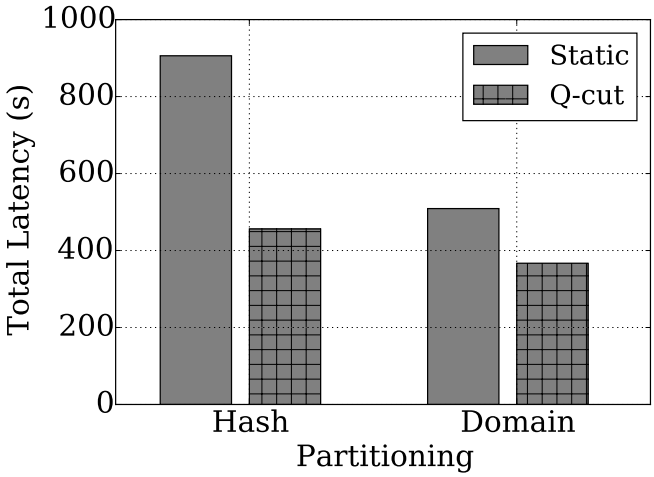


Figure 6.4: BW POI queries: Total query latency

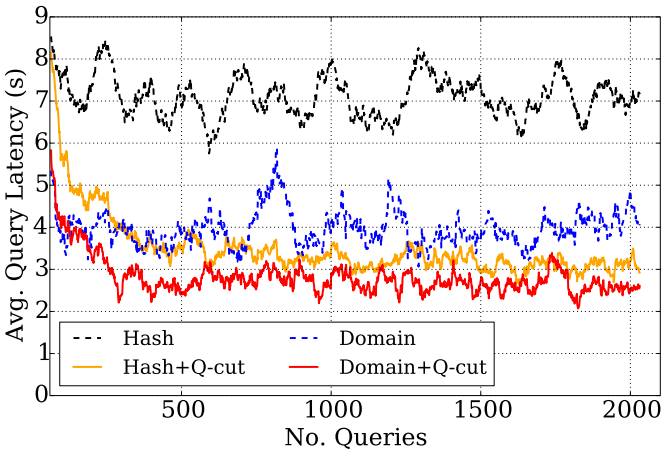


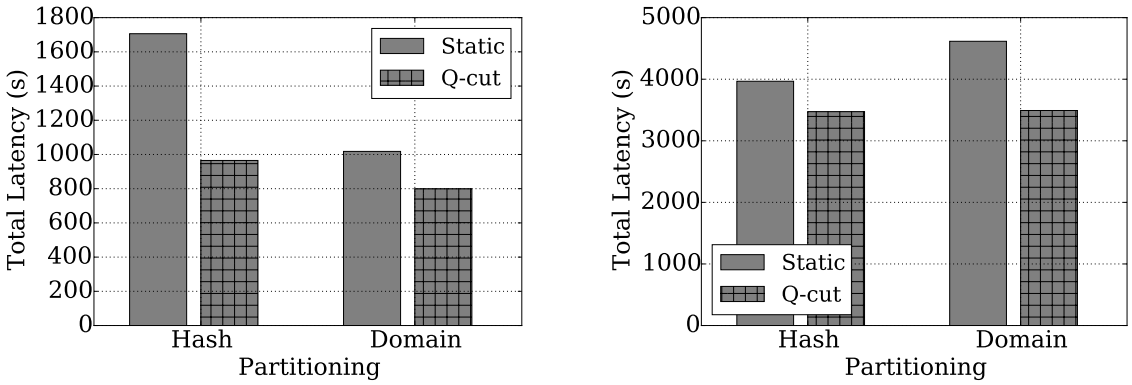
Figure 6.5: BW POI queries: Average query latency

6.3 Adaptive Q-Cut Partitioning

In this section we present evaluations showing that Q-Cut can also detect changes of query characteristics and adapt partitioning dynamically. Two scenarios were evaluated: *Shortest path* queries for the *GY graph* and for the *BW graph*. In both cases query characteristics change at a certain point. The first 2048 queries are *urban queries* with shorter range and higher locality. After a *disturbance* the queries change to 512 *overland queries*. These queries have a longer range, different locality and in general longer processing times. The experiments were performed on the *EC2 cloud* setup.

In Figure 6.6 it is shown that Q-Cut can improve the total query latency in both scenarios. For *BW* the improvement is 53% over hashed partitioning. In *GY* Q-Cut gives an improvement of 12% over hashed partitioning and 25% over domain partitioning, which is worse in this scenario.

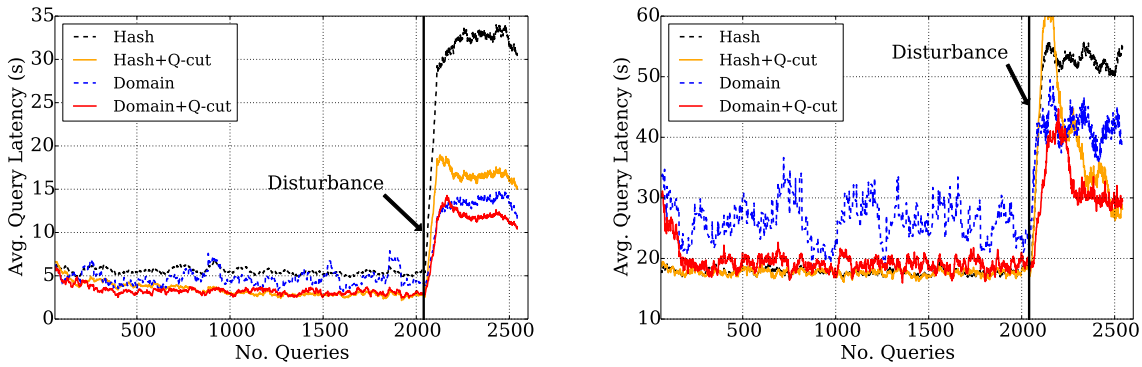
The average query latencies in Figure 6.7 give a more detailed picture. Before the disturbance there is the convergence similar to results in Section 6.2. Q-Cut quickly reaches the minimum, independent from the initial partitioning
After the disturbance the query latencies for the different partitioning strategies change drastically. In both scenarios hashed performs much worse on the long queries. For the BW graph Q-Cut quickly reach a good latency again. This explains the low total query latency.
However for the GY graph it takes a much larger repartitioning effort but ultimately Q-Cut reaches a low query latency as well. The total query latency for this experiment is higher as Q-Cut did not have enough time to benefit from the repartitioning investments. For a larger amount of queries the total latency improvement of Q-Cut would be significantly higher.



(a) Shortest path queries on BW graph

(b) Shortest path queries on GY graph

Figure 6.6: Total Query Latency for shortest path queries



(a) Shortest path queries on BW graph

(b) Shortest path queries on GY graph

Figure 6.7: Average Query Latency for shortest path queries

6.4 Hybrid Barrier

Both evaluations in Section 6.2 and Section 6.3 used the *hybrid barrier* optimization. In this section, an evaluation to show the impact of hybrid barrier is presented. The test was performed on the *notebook* setup using 64 shortest path on the *BW* dataset. We compared traditional BSP-like barrier synchronization where all queries perform a global barrier to the hybrid barrier synchronization with 16 parallel queries.

Figure 6.8 shows the comparison of BSP synchronization and hybrid barrier for hashed partitioning and domain partitioning. When using hashed partitioning, hybrid barrier improves the total query latency by 20%. For domain partitioning the gain is much larger with 43%. Partitioning with high locality supports hybrid barrier and vice versa. With a better locality the number of local supersteps is higher. Local supersteps can increase the performance more if hybrid barrier allows parallel execution of queries with different barriers.

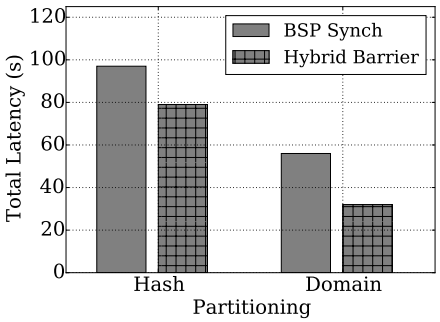


Figure 6.8: Latency improvement with domain partitioning and hybrid barrier synchronization

6.5 Scalability

Finally we show the scalability of the Q-Graph system in combination with Q-Cut and hybrid barrier. For this test, the *compute cluster* setup with 8 compute nodes was used. A series of 1024 queries on the BW graph was tested for both *shortest path* and *POI* queries. In both tests the hybrid barrier optimization with 16 parallel queries was used.

For shortest path, as shown in Figure 6.9, the query total latency decreases contentiously for up to 8 workers. For hashed partitioning it decreases from 927s to 474s but increases for more than 8 workers. Hashed partitioning shows worse scalability behavior as the large number of messages and synchronization has more impact for a larger number of workers. Domain partitioning has the disadvantage of poorer workload balance. This has a bigger impact for a lower number of workers. In combination with Q-Cut, domain partitioning performs best, reducing latency to 283s. POI queries show the same scalability behavior with Q-Cut partitioning leading to the lowest total latency.

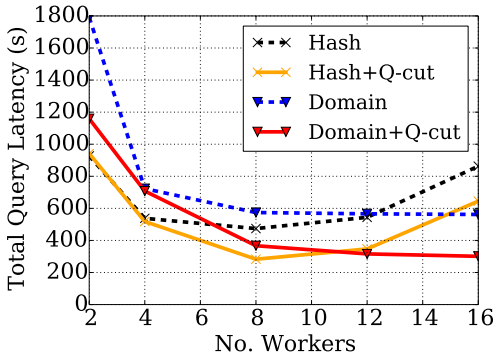


Figure 6.9: Scalability of Q-Graph for shortest path with increasing worker count

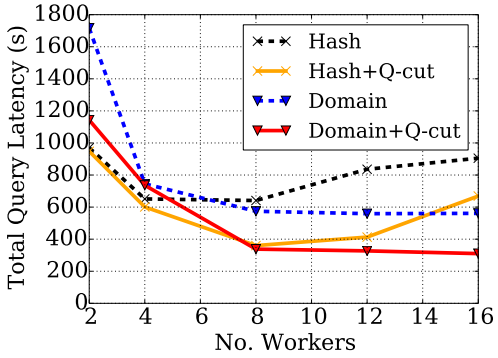


Figure 6.10: Scalability of Q-Graph for POI with increasing worker count

Chapter 7

Conclusion and Future Work

Graph processing systems are used in a wide range of scenarios. This thesis presented *Q-Graph*, multitenant graph processing system and *Q-Cut*, an adaptive and dynamic graph partitioning method. Additional optimizations for multi-query graph processing were introduced: *Hybrid barrier synchronization* and *local query execution*, both reducing synchronization overhead and improving query latency and scalability. A case study for road network graphs was presented. Two algorithms were implemented: *shortest path* and *point of interest (POI) search*.
Evaluations showed that when used in combination, these techniques reduce the query latency by up to 60%. Q-Cut partitioning was superior in all tests over hashed and domain partitioning as it increases both query locality and workload balance. In contrast to static partitioning techniques, Q-Cut was able to detect changes in query characteristics and to adapt the partitioning accordingly.
Each single optimization improves the performance and query latency but a combination offers best results. A good partitioning reduces the number of remote messages and increases the locality of queries. Hybrid barrier synchronization and local query execution removes the need for unnecessary query synchronizations and enables workers to quickly process a local query on one machine.

Q-Graph's centralized system architecture gives opportunities for further optimizations. Global knowledge could be used more efficiently to detect query characteristics and adapt the system faster to changes. Pattern detection could recognize query characteristics and partition accordingly. Furthermore, the Q-Graph system can be further optimized in general. Combiners aggregating information of vertex messages could reduce the total number of messages.
The algorithm implementations presented in the case study could also be improved. Modern implementations of the shortest path implementations use hierarchies and other optimizations which could be adapted for this usage. The constant delta stepping method could also be extended by a dynamic, adaptive stepping method. Finally, more graph algorithms could be implemented for *Q-Cut* for a broader range of uses.

Bibliography

[Chi13] A. Ching. Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, p. 25, 2013.

[CSCC15] R. Chen, J. Shi, Y. Chen, H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*. 2015.

[DG08] J. Dean, S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DHES16] A. Dubey, G. D. Hill, R. Escriva, E. G. Sirer. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *VLDB*, 2016.

[GLG+12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. 2012.

[Had09] A. Hadoop. Hadoop, 2009.

[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 *SIGMOD*. 2010.

[MMTR16] R. Mayer, C. Mayer, M. A. Tariq, K. Rothermel. GraphCEP: real-time data analytics using parallel complex event and graph processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 309–316. ACM, 2016.

[MS03] U. Meyer, P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[MTLR16] C. Mayer, M. A. Tariq, C. Li, K. Rothermel. GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning. In *ICDCS*. 2016.

[RBVM15] P. Rutgers, P. Boncz, S. Voulgaris, C. Martella. *Extending the Lighthouse graph engine for shortest path queries*. Ph.D. thesis, Master's thesis, Vrije Universiteit, 2015. http://homepages. cwi. nl/boncz/msc/2015-Rutgers. pdf. 5, 2015.

[Ski90] S. Skiena. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.

[SW13] S. Salihoglu, J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, p. 22. ACM, 2013.

[SYK+10] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 721–726. IEEE, 2010.

[Val90] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990. doi:10.1145/79173.79181. URL `http://doi.acm.org/10.1145/79173.79181`.

[XYHD16] J. Xue, Z. Yang, S. Hou, Y. Dai. Processing Concurrent Graph Analytics with Decoupled Computation Model. *IEEE Transactions on Computers*, 2016.

[XYQ+14] J. Xue, Z. Yang, Z. Qu, S. Hou, Y. Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 227–238. ACM, 2014.